A Constraint-based Partial Evaluator for Functional Logic Programs and its Application

Laura Lafave

A thesis submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science

October, 1998

Abstract

The aim of this work is the development and application of a partial evaluation procedure for rewriting-based functional logic programs. Functional logic programming languages unite the two main declarative programming paradigms. The rewriting-based computational model extends traditional functional programming languages by incorporating logical features, including logical variables and built-in search, into its framework.

This work is the first to address the automatic specialisation of these functional logic programs. In particular, a theoretical framework for the partial evaluation of rewriting-based functional logic programs is defined and its correctness is established. Then, an algorithm is formalised which incorporates the theoretical framework for the procedure in a fully automatic technique. Constraint solving is used to represent additional information about the terms encountered during the transformation in order to improve the efficiency and size of the residual programs. Experiments using an implementation of the algorithm for the partial evaluation of Escher programs show that the specialiser not only passes the "KMP-test", but also can perform the elimination of data structures and obtains notable speed-up for McCarthy's 91-function.

Circuit simulation lends itself to optimisation by partial evaluation. A general interpreted-code circuit simulator can be specialised with respect to a particular design in order to improve the simulation speed. In this work, a simulator is implemented for behavioural and register-transfer level designs written in the Verilog hardware description language. Testing and verification of high-level designs using interpreted-code simulators is notoriously inefficient. In this thesis, it is shown that the efficiency of an event-driven simulator for behavioural or register-transfer level designs can be improved automatically by partial evaluation.

Declaration

The work in this thesis is the independent and original work of the author, except where explicit reference to the contrary has been made. No portion of the thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of education.

Laura Lafave

Acknowledgements

Firstly, I would like to thank my supervisor, John Gallagher. Although I will probably never forgive him for the 91-function example (see Section 5.5), I could never fault his supervision throughout these four years. He allowed me to explore the area, while offering the guidance that only his expertise could provide. Of course, his unfailing optimism cannot go without mention. He introduced me to the research community and made opportunities available to me that I would have otherwise not had.

I would like to thank my viva committee, Neil Jones and Henk Muller, for their helpful comments and interesting discussion of this work.

I would like to express my appreciation to Julio Peralta for the many discussions on imperative program transformation, program slicing, semantics, and homeomorphic embeddings; Kerstin Eder and John Lloyd for the Escher details and implementation; Tony Bowers, for still supporting Gödel; and, Brian Stonebridge and Henk Muller for their collaboration.

I found my visit to DIKU this year had a profound effect on my research. I would like to thank Neil Jones for finding the extra funding for the trip. In addition, I would like to thank Neil, Morten Sørensen, Robert Glück, Jens Peter Secher, Arne Glenstrup, Torben Mogensen, and Henning Niss for many interesting discussions. Extra thanks to Michael Leuschel for the party, Henning for the cake, and Morten and Mette for their hospitality.

There are many people in the research community who have influenced my work, in particular, Bern Martens, Michael Leuschel, Jesper Jørgensen, Hüseyin Sağlam, Andre de Waal, German Puebla, Jonathan Martin, Andy King and Pat Hill.

On a personal level, I would like to thank Adam Worrall and David Hedley for their friendship from the first day I stepped into the department. I would also like to thank the person who chose to put the Declarative Systems group in the same office as the Safety Systems Research Centre. But since I don't know who this bureaucrat is, I will instead just thank John Napier and Ilesh Dattani, particularly Ilesh for all of his proofreading during the last two months. Furthermore, thanks to Stefan Kruger, particularly for the LaTeXtemplates.

I would like to thank Pete for his patience, support, and friendship. Removing infix notation from Escher code is no way to spend a holiday, but Pete has always been helpful beyond the traditional call of duty. I also think that Ru Morris and Stephanie Hollington should be acknowledged here, both for their friendship and for feeding Pete while I was at DIKU.

And finally, they may be two thousand miles away, but it's not *really* that far. Thanks to my parents, Grandma, Grandpa, David, and Daniel.

I was supported during my PhD study by grants from the National Science Foundation (USA) and from the British Marshall Aid Commemoration Commission.

Contents

Chapter 1

Introduction

As the modern computer reaches its first half century of existence, it is possible to reflect on general trends during its development. While the physical size of the machines has rapidly decreased over the years, computing power has greatly increased. The increased computing power has permitted the development of larger and more complex software systems offering greater functionality and applicability. As a result of the increasing complexity of the software systems, high-level programming languages have become more popular, and data abstraction techniques are commonplace in software development. Furthermore, structured software development methods and CASE tools have transformed the development lifecycle into one having a closer resemblance to science rather than art.

Software developers aim to release products that have high quality and maintainability standards, while being efficient and timely. As the complexity of software increases, techniques that allow rapid development of quality software, including modular programming and programming in high-level languages, have gained popularity. Unfortunately, the use of these methodologies tends to have an adverse effect on the efficiency of the resulting products. For example, in modular programming, the off-the-shelf components used in the development of the system usually have greater functionality, and therefore complexity, than is required. Programming in high-level languages such as declarative languages allows developers to express *what* the program should compute in a smaller, more readable form than in low-level languages, but the advanced features of these languages also suffer from efficiency problems. Generally, increasing the generality or abstraction level has an adverse effect on the efficiency of the software.

The ideal trend for the future of computing would allow software developers to adopt any of the tools for rapid development of quality software, while still guaranteeing that the overall efficiency of the resulting program is optimised. So far, the responsibility of ensuring the efficiency of software has been assigned to the compiler [ASU86]. However, basic compilers are difficult enough to construct without having to include many specialised operations.

An advanced optimisation that is not included in traditional compilation operations is the specialisation of a program to solve a particular problem. In general, programs are designed to solve a class of problems. For example, a digital circuit simulator exhibits the behaviour for a class of circuit designs. On the other hand, designers often repeatedly simulate a particular circuit design for a set of environment vectors, requiring many runs of the original program with the circuit's design as input. This is very inefficient, as the structure of the circuit has to be recomputed by the program with each new environment vector. It would be advantageous to be able to generate a more efficient version of the digital circuit simulator that simulates *only* the circuit in question.

This advanced specialisation operation is possible by a program transformation technique called *partial evaluation*.

1.1 Partial Evaluation

Programs are transformed by partial evaluation in order to obtain a more efficient version of the program for a given subset of the domain of input data [Ers78]. That is, a program is *specialised* with respect to a fixed subset of the input data. The efficiency of the original program is improved by evaluating the expressions in the program that depend on the fixed input and generating specialised code for those expressions that depend on the run-time input data.

This program transformation is performed by a partial evaluator, a meta-program which takes as input a program and the fixed data and generates a specialised program, called a *residual* program, which is operationally equivalent to the original program with respect to the fixed data (Figure 1.1) [JGS93, Jon96a]. In this way, the computation of a program is divided into two stages by the program transformation, a compile-time computation and a run-time computation. The residual program is the intermediate program of this transformation, much like intermediate functions that are formed by the partial application of curried functions.

Partial evaluation is a source-to-source transformation; the residual programs generated by the partial evaluator are programs in the same language as the original program. Therefore, partial evaluation differs from strict compilation, since the efficiency is not gained by compiling into a lower level language, but by performing optimisations on the code. Although a loss of readability is typically incurred by the transformation, the user has access to the program for further development or

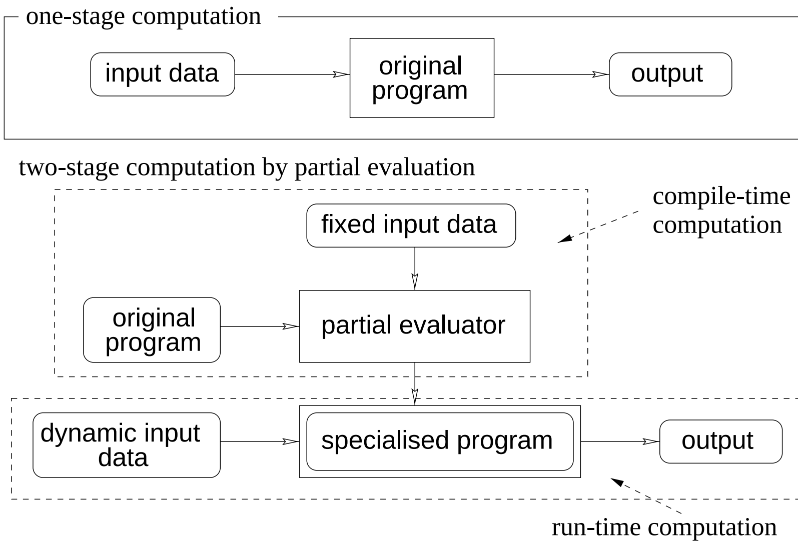


Figure 1.1: Using a partial evaluator to specialise a program with respect to some fixed data.

documentation.

In order to be truly useful, the gain in efficiency of the residual program must be greater than the cost of performing the specialisation of the original program. Typically, any application in which a program is run many times with a fixed subset of input data will benefit by partial evaluation.

For example, considering the digital circuit simulator example again, if the simulator is specialised with respect to a circuit design, the resulting residual program will be the original program specialised to simulate only one circuit: the circuit that was passed as input to the partial evaluator (Figure 1.2). Experiments specialising a gate-level circuit simulator with respect to a given circuit design demonstrated that the residual programs in this case could be up to 91 times faster than the original simulator [BW90]. Of course, a similar result is obtainable by compiled hardware simulation. A compiled simulator takes a circuit design as input and generates a program to exhibit its functional behaviour [Mic86]. However, these compiled hardware simulation programs are much more complex, and therefore much more difficult and costly to write, than a basic (interpreted-code) digital circuit simulator (§ 6.1.2).

However, partial evaluation has not been applied widely in real-world applications. In general, the concentration of research in the area of program transformation has been on obtaining an optimal technique, rather than investigating practical applications of the technology. Furthermore, low-level programming languages are notoriously difficult to transform automatically, because of their lack

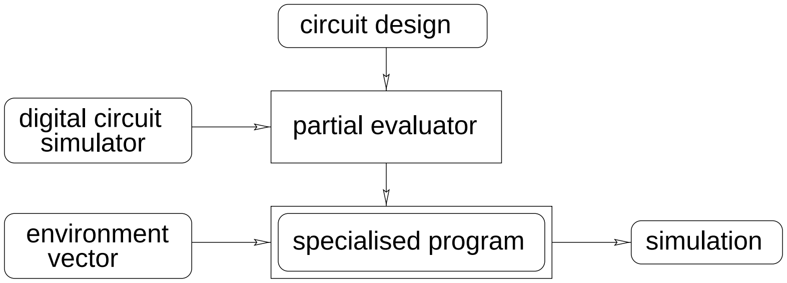


Figure 1.2: Applying partial evaluation to a digital circuit simulator.

of semantic foundations. On the other hand, the industry is often reluctant to incorporate tools that require user intervention, since the training and staff requirements are often too costly.

Therefore, the goal of this work is to develop an *automatic* optimisation tool for a high-level language that obtains noticeable increases in efficiency for a real application. The specialisation achieved by online partial evaluation is improved by extending the information representation in the procedure. This advanced technique for information propagation is incorporated into an automatic procedure for the partial evaluation of a new class of high-level declarative programs. Then, experiments using the automatic partial evaluator and a general interpreted-code digital circuit simulator show that the transformation can generate efficient simulators, on par with those only available currently by compiled simulation.

1.2 Contributions of the Thesis

The partial evaluation procedure described in this thesis transforms rewriting-based functional logic programs, a new declarative programming paradigm which incorporates features of both functional and logic programming languages. These declarative languages are very flexible and expressive. However, as discussed earlier, this flexibility has a price: efficiency. Applying partial evaluation to these programs reduces this cost by automatically improving their efficiency.

Therefore, the main contributions of this thesis are as follows:

Theoretical Framework The work outlined in this thesis is the first study of the partial evaluation of rewriting-based functional logic programs. This new declarative programming language paradigm incorporates aspects from both functional programming languages and logic programming languages. Similarly, the theoretical framework for the partial evaluation procedure shares features from many declarative partial evaluation procedures, including the partial evaluation of logic programs, deforestation, and supercompilation.

Proof of Correctness of the Procedure The procedure for the partial evaluation of rewriting-based functional logic programs is proved to be sound and complete for a large class of programs and terms. In each case, a single closedness condition is required, in a manner similar to the partial evaluation of logic programs [LS91]. In addition, the theoretical framework also covers the specialisation of lazy higher-order functional languages.

Algorithm Design The algorithm for the online partial evaluation of functional logic programs describes a fully automatic technique [LG98a, LG97]. The control of the algorithm ensures that for any program, the partial evaluator terminates and returns an optimised program. The procedure described by the algorithm is shown to be correct in terms of the theoretical framework. This establishes the algorithm as a correct transformation for rewriting-based functional logic languages.

Constraint Solving Integration The algorithm for the partial evaluation of rewriting-based functional logic programs is extended to represent information about expressions in the program using constraints. This allows the advanced representation and exploitation of information by the partial evaluator, while maintaining the termination qualities and automation of the specialiser. The online partial evaluator uses this extra information about the expressions to make better decisions during specialisation, resulting in better quality residual programs. The benefits of representing information by constraints are explored, using the 91-function as an independent gauge to compare various techniques [LG98b].

Implementation and Experiments An implementation of the algorithm for the partial evaluation of Escher programs is described in this thesis. The Escher language is an extension of the Haskell language providing logical variables, non-determinism, and set abstractions [Llo95, Llo]. Experiments on benchmark programs illustrate the efficiency of the residual programs in comparison with the results of existing program specialisation procedures.

Application: Circuit Simulation As discussed earlier, specialising a digital circuit simulator with respect to a circuit's design results in a specialised version of the simulator with the interpretive overhead removed. In this case, the circuit simulator is based on the Verilog semantics defined by Gordon [Gor95]. The partial evaluator specialises the interpreted-code digital circuit simulator with respect to the representation of a Verilog design to generate simulators that are up to four times faster than the original tool.

1.3 Overview

The thesis is organised as follows. The first part of this thesis concentrates on the development of a constraint-based partial evaluation procedure for rewriting-based functional logic programs. The second part of the thesis concerns the application of the constraint-based partial evaluator.

Chapter 2: Technical Background This chapter begins with an overview of partial evaluation, including an introduction to established specialisation techniques for functional and logic programs. The chapter also includes a brief introduction to constraint solving and to the features of functional logic programming languages. Then, the syntax and semantics for the rewriting-based functional logic language *E* are defined. This subset of the Escher language [Llo95] is the example target language of this thesis.

Chapter 3: Partial Evaluation of Functional Logic Programs The theoretical framework of the partial evaluation procedure for rewriting-based functional logic languages is presented in this chapter. The total correctness of the procedure is established with respect to several domains of programs. The chapter concludes with a comparison of the features of the procedure with the program specialisation techniques for other declarative languages.

Chapter 4: Constraint-based Partial Evaluation Algorithm The algorithm for the partial evaluation of rewriting-based functional logic programs is presented in this chapter. The algorithm incorporates the theoretical framework of Chapter 3 with constraint solving technology to improve the representation of information in the procedure. The control of the partial evaluation is defined, as well as the functions for constructing the residual program. The algorithm is proved to terminate for any program and term and satisfies the correctness requirements of the previous chapter. In addition, several extensions of the algorithm which improve the quality of the residual programs are presented at the end of the chapter.

Chapter 5: Experiments with Constraint-based Partial Evaluation The partial evaluator described in Chapter 4 has been implemented to specialise Escher programs. The implementation of the partial evaluator is briefly surveyed. Then, the results of specialising several benchmark programs, including the naive pattern matching program, McCarthy’s 91-function, the Ackermann function, and a general interpreter for a simple imperative language, are evaluated. In particular, the performance of the constraint-based program specialiser is compared with programs resulting from other well-established program transformers, including positive supercompilation [SGJ96, GS96], deforestation [Wad90], and generalized partial computation [FN88].

Chapter 6: Compiled Simulation by Program Specialisation Generating compiled simulations using a semantic-based interpreter and a hardware design described in a hardware description language (HDL) is covered in Chapter 6. This chapter begins with an introduction to the hardware design cycle and digital circuit simulation technology. Then, the semantics for the Verilog HDL are discussed, first in terms of the general IEEE semantics, and then in terms of the defined semantics for a subset of the Verilog language as defined in [Gor95]. A interpreted-code simulator for the subset of Verilog written in Escher is specialised using the constraint-based partial evaluator with respect to several basic circuits. The results of the experiments and a comparison with earlier work [AWS91] concludes the chapter.

Chapter 7: Conclusions and Further Work A review of the procedures and applications of this work is presented in Chapter 7. Several areas of this work which could be extended by future research are discussed.

The benchmark programs are included in the Appendix.

The work presented in this thesis has been published previously in the refereed papers [LG98b, LG98a, LG97, GL98, GL96].

Chapter 2

Technical Background

In this chapter, the necessary technical background is presented. The chapter begins with a review of the established program specialisation techniques for declarative programs in Section 2.1. In particular, Sections 2.1.3 and 2.1.4 introduce established procedures that will be used in Chapter 5 for comparing the performance of the constraint-based partial evaluator described in this work.

Then, a brief introduction to constraint solving is covered in Section 2.2. This section contains an introduction to constraint domains (§ 2.2.1) and the constraint solving operations on these domains (§ 2.2.2).

Finally, the chapter concludes with an introduction to functional logic programming and a definition of the example language of this thesis in Section 2.3. Section 2.3.1 relates rewriting-based functional logic languages to narrowing-based languages. A brief introduction to the Escher programming language is provided in Section 2.3.3. The syntax and semantics of the *E* rewriting-based functional logic language, a subset of the Escher language, is formalised in Sections 2.3.4 and 2.3.5.

2.1 Program Specialisation by Partial Evaluation

Partial evaluation is a source-to-source program transformation which optimises programs by performing some computation at compile-time. A partial evaluator, given a program and some fixed input to that program, generates a *residual program*, which is the original program specialised with respect to that fixed subset of input data. Given the rest of the input data, the residual program computes the same result as the original program run with the entire set of input data. Figure 1.1 from the previous chapter illustrates graphically the process of specialising a program by partial evaluation. A partial evaluator, PE, converts a one-stage program, p, into one that computes in two stages. For a program p which takes two sets of data $d1$ and $d2$ as input, assuming the set $d1$ is fixed, partial evaluation is described by the following equation:

$$\forall p, d1, d2 : [\![p]\!][d1, d2] = [\![p']\!][d2] \text{ where } p' = [\![PE]\!][p, d1]$$

where $[\![\_]\!]$ is the program meaning function for a given implementation language [JGS93]. That is, $[\![p]\!][d1, d2]$ is the result of running the two-input program p using input data $d1$ and $d2$.

The residual program is the intermediate program constructed in the two stage computation, similar to intermediate functions constructed during partial application of curried functions. A partial evaluator is a *meta-program*; it takes a program, called an *object* or *source program*, as data. If the implementation language of the partial evaluator is the same as the language of the object programs, also called the *source language*, the partial evaluator can be applied to itself. Self-application of partial evaluators results in compilers and compiler-generators, known as the second and third Futamura projections [Fut71].

In order to generate the residual program, a partial evaluator must perform computation of the program code that depends on the fixed input data, also called the *static* data, while generating code for the parts of the original program that depend on the unknown input data, the *dynamic* input data. In other words, the partial evaluator must decide which program expressions to *reduce* and which to *residualise*. Both the efficiency and size of the residual program depend directly on the quality of these decisions.

There are two main frameworks for handling the reduce/residualise decision making of a partial evaluator. *Offline* partial evaluators take *annotated* programs as input. The annotations indicate which expressions should be reduced by the partial evaluator; otherwise, code is generated for the unannotated expressions by the partial evaluator. That is, the partial evaluator simply performs the actions indicated by the annotations in order to generate the residual program. On the other hand, *online* partial evaluators perform the decision-making during program specialisation. Based on the static values of the data, the partial evaluator decides whether to reduce or residualise the expressions of the program. The advantage of offline partial evaluators is their smaller size and reduced complexity as compared to an online partial evaluator. The smaller specialisers respond better to self-application [JGS93]. On the other hand, since the online specialisers have more information about the static data, the residual programs tend to be more efficient than those generated by offline techniques [Ruf93].

	$T[\![$ **if** $u = v$ **then** t **else** $s]\!] =$	information propagation
(a)	**if** $u = v$ **then** $T[\![t]\!]$ **else** $T[\![s]\!]$	constant propagation
(b)	**if** $u = v$ **then** $T[\![t\{u := v\}]\!]$ **else** $T[\![s]\!]$	unification-based
(c)	**if** $u = v$ **then** $T[\![t\{u = v\}]\!]$ **else** $T[\![s\{u \neq v\}]\!]$	constraint-based

Table 2.1: The types of information propagation for partial evaluation. The double brackets identify the syntactic argument of the transformation function *T*.

Therefore, partial evaluators can be categorised according to their reduce/residualise decision making approach. A further distinguishing feature of specialisers is the type of information representation and exploitation employed. In general, the quality (size and efficiency) of the residual programs is positively related to the extent of information representation in the specialiser. The additional information improves the reduce/residualise decision making. Furthermore, unreachable expressions can be removed from the object program, if information from the conditions of conditional expressions or case statements is extracted.

Glück and Sørensen identified three different methods for transforming a conditional statement, as shown in Table 2.1 from [GS96]. In constant propagation, the information in the condition is not used by the specialiser when evaluating the subterms of the expression. In unification-based propagation, the positive information from the condition is represented as a positive binding and applied to the positive branch of the conditional. A constraint-based transformer represents the positive and negative information in the conditions as constraints associated with each sub-expression.

2.1.1 Development History

Historically, the theoretical foundation for partial evaluation is attributed to Kleene's s-m-n theorem [Kle52]. This theorem states, for a recursive function f of $m + n$ arguments, the n-ary function $f_{d_1 \ldots d_m}$, which maps $d_{m+1} \ldots d_{m+n}$ to $f\ \ d_1 \ldots d_m\, d_{m+1} \ldots d_{m+n}$ is recursive. Kleene's proof of the s-m-n theorem outlines the specialisation of a Turing machine, although the construction described in the proof does not guarantee improved efficiency.

The origins of modern partial evaluation can be traced to early work on the specialisation of LISP programs by Lombardi and Raphael in 1964 [Lom64, LR67]. In 1971, Futamura published the first main application of the early work in partial evaluation: self-application and the generation of compilers [Fut71]. Ershov, in cooperation with Turchin and Romanenko of the supercompiler project, independently discovered the three rules for the generation of compilers and compiler-generators, which he called the "Futamura projections" [Sør96, Ers78]. He is also responsible for the terminology “mixed computation” for what is also called partial evaluation [Ers78]. Since then, program specialisation by partial evaluation has been applied extensively. For this reason, a full history of its development is outside the scope of this thesis. An overview of the history of partial evaluation research can be found in [JGS93]. Instead, several program specialisation procedures for declarative languages, in particular, functional and logic programming languages, are reviewed in this section. A brief introduction to the operation of each procedure is presented and each technique is categorised in terms of the two features of specialisers discussed earlier: their decision making approach and information representation and handling.

Many program specialisation procedures can be related to the unfold/fold program transformation framework. The review of the existing program specialisation technology begins with an introduction to this system.

2.1.2 The Unfold/Fold Transformation

Burstall and Darlington introduced the unfold/fold program transformation framework in their 1977 paper as a program synthesis system with the original intention to obtain programs from specifications [BD77]. Since then, several program transformation procedures, such as the partial evaluation of logic programs and deforestation, have been formalised in terms of the unfold/fold transformation rules [PP94, Mar96].

The transformation process can be defined as a sequence of programs $P_0, P_1, \ldots, P_k$ such that a program P_i is obtained from P_{i-1} by an application of either the *unfolding* rule or the *folding* rule.

Unfolding: Given a program P containing an equation with term t in the right-hand side, let t be an instance of some e_1, $t = e_1\theta$, where $e_1 = e_2$ is in P and θ is a substitution. Then t can be replaced in the right hand side of the equation with the term $e_2\theta$.

Folding: Given a program P containing an equation with term t in the right-hand side, let t be an instance of some e_2, $t = e_2\theta$, where $e_1 = e_2$ is in P and θ is a substitution. Then t can be replaced in the right hand side of the equation with the term $e_1\theta$.

These two basic rules are supplemented with ones for *defining* new equations and *instantiating* existing equations to form the basic transformation system. These four rules are typically used in the following order during the transformation: define, instantiate, unfold, fold.

- New *definitions* are created in the program. This step is called the “eureka” step. This cannot always be performed automatically; a certain level of ingenuity is needed in some cases

[PP94, PP90].

- Some equations are *instantiated*. These instantiated definitions automatically overlap the existing definitions, but are necessary for the unfolding step.
- Some terms in the right-hand sides of equations are *unfolded*, as defined above.
- Then, in some cases, the new right-hand sides are *folded* using existing equations in the program.

The unfold/fold rules can be deceptive; their application preserves the partial correctness of a program, but their use must be closely controlled in order to ensure the operational equivalence of the transformed program [San95b, San95c]. Folding can alter the recursive structure of a program. Applying the folding step carelessly may result in a loss of termination in the resulting program; for example, consider folding the equation $F(x) = 42$ to $F(x) = F(x)$. Restrictions for the application of the transformation rules above have been identified which preserve the total correctness of the transformed programs [Kot85].

2.1.3 Partial Evaluation of Functional Programs

This section contains surveys of five specialisation frameworks for functional programs: four online techniques, and one offline technique based on Jones' Mix system. Each technique exhibits particular features of specialisers which will be used to evaluate the overall performance of the constraint-based partial evaluator in Chapter 5.

Partial Evaluation

The first self-applicable partial evaluator was Jones' Mix system [JSS85, JSS88]. Since it is an offline specialiser, the partial evaluator simply follows the annotations of the object program to generate the residual program (§ 2.1). In the original self-application experiments, these annotations were added to the object program manually. Later, a technique for the automatic annotation of programs, called Binding Time Analysis (BTA), was developed [Bon90, NN88, GJ96].

Offline partial evaluators do not have access to the static values, but only know *which* arguments are fixed (static). Based on this information, the expressions in the program are annotated with their binding times, based on the binding times of their arguments. For example, an expression may be annotated to be reduced by the partial evaluator if all of its arguments have a "static" binding time. BTA has been defined for first-order and higher-order programs.

Partial evaluators based on the Mix system (as defined in [JSS85]) have limited specialisation "power". One reason for this is the use of constant-based information propagation [Ruf93]. In addition, the offline partial evaluator suffers from the conservative annotations of the binding time analysis. In order to ensure a gain in efficiency, the binding time analysis must often annotate expressions as "residualise" when they could safely be reduced. The structure of expressions in the object program has a great effect on the quality of the residual program for these specialisers. Two operationally equivalent programs with different structure may specialise very differently. Binding time *improvements* are semi-automatic alterations of the structure of a program intended to improve its specialisation for a given partial evaluator. For example, converting a program to continuation passing style improves its specialisation potential by isolating static subterms from a dynamic expression [CD91, LD94, Dus97]. This conversion of a "bad" program structure to a "better" structure for specialisation is not limited to specialisers based on Mix; preprocessing transformations based on grammar analysis (see deforestation below) and consumption analysis [Thi94, CK96] have been developed for online specialisers as well.

Deforestation

Deforestation is an online, unfold/fold-based program transformation technique aimed at eliminating intermediate data structures ("trees") from functional programs. As defined originally in [Wad88, Wad90, FW88], deforestation can be applied to first-order functional programs containing only *treeless* functions, that is, functions guaranteed not to generate intermediate data structures when evaluated. Given a non-treeless term and a treeless program, deforestation generates an operationally equivalent treeless term and program. This is achieved by unfolding a sub-expression until an outermost constructor is *produced* and then *consuming* the constructor using the surrounding function call [Chi90]. The intermediate data structure is either eliminated or forced to the outermost position in the term. In addition to the treeless function restriction, there is an additional linearity condition imposed on the terms to ensure the resulting program is more efficient than the original program.

Termination of the original deforestation algorithm is ensured by a basic folding step for generating recursive functions in the transformed program. This folding step is sufficient for guaranteeing termination of the algorithm because of the restrictive nature of the treeless form [FW88].

Recent research in deforestation has concentrated on extending its application to both arbitrary first-order and higher-order functional programs. Wadler developed two methods for extending the domain of deforestation: blazing and higher-order macros [Wad90]. Since then, a number of static analyses to annotate unsafe expressions in first-order functional programs have been developed by

Chin [Chi90, Chi91, Chi94], Hamilton [Ham93, HJ91, Ham91], Sørensen [Sør94, Sør93], and Seidl [Sei96]. Approaches to higher-order deforestation problem include extending the rules of Wadler's deforestation [MW92, Mar96, Ham96] and adding a higher-order removal transformation [Chi92].

Deforestation provides no information propagation. The rule for evaluating a case statement by deforestation is shown below [Wad90]:

$$T[\![\, \textbf{case}\ v\ \textbf{of}\ p_1 : t_1 \mid \ldots \mid p_n : t_n]\!] = \textbf{case}\ v\ \textbf{of}\ p_1 : T[\![t_1]\!] \mid \ldots \mid p_n : T[\![t_n]\!]$$

The treeless form required by the procedure does not permit unification-based information propagation. In fact, it is possible to consider positive supercompilation, presented later in this section, as a unification-based deforestation technique for first-order programs.

Supercompilation

The supercompilation project was started in the 1960s with the aim of developing a system for artificial intelligence and program synthesis [Tur86, TNT82]. The Refal language, a complex pattern-based, call-by-value first-order functional language, was the source language for the supercompiler. Supercompilation, *supervised compilation*, is an abstract technique composed of three actions modelled on human thought:

- Deduction: In problem solving, one applies some pre-established rules for thought.
- Generalisation: In order to construct a solution from observations/deductions, one generalises the observable patterns.
- Metasystem Transition: If the problem is not solvable, one needs to evaluate the current rules and generate new rules for deduction.

It has been shown that supercompilation can perform program specialisation, language translation, theorem proving, and specialiser generation [TNT82]. The deduction operation of a supercompiler is performed by *driving*. By driving, a term is unfolded and a tree of possible computations, called a *process tree* is produced [GK93]. Driving is a constraint-based information propagation technique; positive and negative information is passed during supercompilation by environments [Tur86]. The generalisation step ensures termination of supercompilation by folding and generalising patterns in the process tree [Tur88].

Recent work in supercompilation includes experiments in self-application of an implemented supercompiler [GT89, NPT96], the application of driving to first-order functional languages [GK93], and continued research in metasystem transitions [Glü96, GK95, GS96]. Supercompilation remains one of the first specialisation techniques to incorporate full constraint-based information propagation.

Positive Supercompilation

Positive supercompilation is an online program specialisation technique incorporating two components of supercompilation: driving and generalisation [SGJ96, GS96, Sør96, SG95, SGJ94]. Unlike supercompilation, positive supercompilation propagates information by unification; for this reason, negative information is not represented. The source language of positive supercompilation is a first-order, call-by-name functional language.

As in supercompilation, driving in positive supercompilation results in the generation of a possibly infinite *process tree*, which contains all possible reductions of a term in a program [SGJ96]. The term is typically a partially instantiated term; therefore, it is possible to consider the positive supercompiler as a non-standard interpreter, evaluating partially instantiated terms in a functional context. Sørensen and Glück formalised an algorithm for the generalisation step of positive supercompilation in [SG95]. The generalisation step ensures the resulting process tree is finite, using a well-quasi ordering over the terms and replacing terms in the process tree with their most specific generalisation when non-termination is a possibility.

The relation between positive supercompilation and the partial evaluation of logic programs (Section 2.1.4) was studied by Glück and Sørensen in [GS94]. The authors erroneously claimed that partial deduction and driving are equivalent processes. Actually, they related the deduction mechanism of logic programming with the unfolding operation of driving. Structures such as the m-trees of Gallagher and Martens [MG95] also have a strong relationship with the process trees of positive supercompilation.

As noted earlier, positive supercompilation is a unification-based program transformation technique. In fact, it is possible to consider positive supercompilation as an extension of deforestation with unification-based information propagation. The generalisation operation allows positive supercompilation to avoid the syntactic restrictions of deforestation, thus ensuring termination of positive supercompilation for arbitrary first-order functional programs.

Generalized Partial Computation

Generalized partial computation (GPC) is an online program specialisation technique which allows *full* information propagation [FN88, FNT91]. Environment information is represented by a set of predicates. Upon evaluation of a conditional expression, the information in the condition is added to the set of predicates and this set is passed to a theorem prover in order to decide if a branch of the conditional can be eliminated.

Generalized partial computation was originally developed in order to improve the results of the Futamura projections. As formalised by Futamura and Nogi in [FN88], this technique assumes the existence of a powerful theorem prover and interaction from the user. Recent implementations of GPC have used constraints and a disunification-based solving procedure instead of a theorem proving mechanism in order to obtain an automatic procedure [Tak92, Tak91]. Libraries of rules particular to given domains are used to simplify and extend the set of constraints generated during the transformation.

GPC can be considered a constraint-based program specialisation technique. The rules for specialising a conditional expression in GPC are as follows [FN88]:

$G[\![$ **if** p **then** t **else** $s]\!]_i = G[\![t]\!]_i$ if $i \vdash p$

$G[\![$ **if** p **then** t **else** $s]\!]_i = G[\![s]\!]_i$ if $i \vdash \neg p$

$G[\![$ **if** p **then** t **else** $s]\!]_i =$ **if** p **then** $G[\![t]\!]_{i \wedge p}$ **else** $G[\![s]\!]_{i \wedge \neg p}$ if it is not "easy to decide" if $i \vdash p$ or $i \vdash \neg p$

In the above equations, the subscripted i represents the current set of information and $i \vdash p$ indicates that the predicate p is provable from the set of information i, using some defined proof theory.

2.1.4 Partial Evaluation of Logic Programs

In this section, the partial evaluation procedure for logic programs, called "partial deduction", is outlined. An introduction to a relevant extension of partial deduction, conjunctive partial evaluation [LSdW96], is also covered in the following review.

Partial Deduction

In logic programming, a goal G is computed in a program P by constructing an SLDNF-tree for $P \cup \{G\}$[1]. In the partial evaluation of logic programs, the partially instantiated goal G' contains the static data, and clauses for the residual program are generated from the SLDNF-tree for $P \cup \{G'\}$. Since the set of data represented in the goal is incomplete, the construction of the SLDNF-tree usually does not terminate. For this reason, in the framework of partial deduction, the concepts of *incomplete* SLDNF-trees and SLDNF-derivations are identified [Kom81]. An incomplete SLDNF-

[1] An introduction to logic programming is beyond the scope of this thesis; see [Llo93] for a description of the procedural semantics of logic programming.

derivation is one in which a literal may not be selected from the final goal of the derivation.

An incomplete, non-failing SLDNF-derivation uniquely defines a *resultant* in the following way:

Definition 2.1.1 (resultant)
Let P be a logic program, $\leftarrow Q$ a goal, and G a finite SLDNF-derivation of $P \cup \{\leftarrow Q\}$ with resolvent $\leftarrow B$ and computed answer θ. Then, $Q\theta \leftarrow B$ is the *resultant* of G.

These resultants form the Horn clauses of the residual logic program. The partial deduction of a program P is defined with respect to a set of atoms **A** by generating incomplete SLDNF-trees for each element of **A** and extracting the resultants from the non-failing derivations of the trees.

Definition 2.1.2 (partial deduction)
Given a program P, atom A, and an SLDNF-tree T for $P \cup \{\leftarrow A\}$, let $G_1, \ldots G_k$ be the non-failing derivations of T, with associated resultants $R_1, \ldots R_k$. The *partial deduction of A in P* is the set of clauses $R_1, \ldots R_k$.

Given a finite set of terms **A**, the partial deduction of P with respect to **A** is the union of the partial evaluations of the elements of **A** in P.

The correctness of partial deduction was established by Lloyd and Shepherdson [LS91]. Two additional conditions were identified: *closedness* and *independence*. Independence ensures the resulting program does not compute additional answers. Closedness guarantees that the literals in the bodies of the clauses are covered by a definition in the residual program.

Definition 2.1.3 (independence, closedness)
A finite set of atoms **A** is *independent* if there are no two elements of **A** with a common instance. Given a set of first-order formulas S, and set of atoms **A**, S is **A**-*closed* if every atom of S with a predicate symbol occurring in **A** is an instance of an element of **A**.

The correctness of the partial deduction of a program P with respect to a goal G follows by requiring the set of atoms **A** to be independent and requiring the literals in the residual program and goal to be **A**-closed.

Partial deduction is naturally a unification-based technique, although a technique using basic binding constraints to propagate negative information was developed by [LS97].

Ensuring the termination of partial deduction is typically divided into *local* control and *global* control. Local control is concerned with the generation of finite incomplete SLDNF-trees during specialisation. On the other hand, the global control guarantees that the set of atoms **A** is finite and independent. Many techniques exist for the control of partial deduction, varying from basic depth-bounds to advanced techniques using characteristic atoms or trees [Leu97a, GB91].

Conjunctive Partial Deduction

A notable extension of partial deduction in relation to this thesis is conjunctive partial deduction [LSdW96, Leu97a]. Partial deduction is a limited unfold/fold transformation, where only the unfolding rule is permitted. One transformation not possible by partial deduction is deforestation, the elimination of intermediate data structures (§ 2.1.3). This results from restricting the generation of resultants to atoms in partial deduction, thus ignoring the variable interdependencies among atoms in the goal G'.

Conjunctive partial deduction is an incremental improvement of partial deduction that incorporates much more of the power of the unfold/fold program transformation paradigm in a controlled context. Like partial deduction, it is a unification-based program transformation technique. Conjunctive partial deduction achieves the same specialisation of logic programs as partial deduction while being able to remove intermediate data structures as in deforestation.

In conjunctive partial deduction, a program is no longer specialised with respect to a set of atoms, but instead with respect to a set of *conjunctions* of atoms. For each element of this set, resultants are generated as in partial deduction. These resultants may not be clauses, since the heads of the resultants may contain conjunctions. A renaming operation is required to convert the resultants to Horn clauses. An algorithm to ensure the termination of conjunctive partial evaluation [GJMS96] shares many of its features with the generalisation procedure of positive supercompilation (§ 2.1.3).

2.1.5 Summary of the Features

A summary of the techniques in light of the features above is shown in Table 2.2.

2.2 Constraint Solving

The previous section contained a survey of the established specialisation techniques for declarative programs, focusing on the information propagation employed by the procedures. As discussed in Chapter 1, an aim of this work is to study the effect of improved information propagation on the

Transformation	Language	Decisions	Information	Automatic
Partial evaluation (Mix)	functional	offline	constant	yes
Deforestation	functional	online/offline	none	yes
Supercompilation	functional	online	constraint	no
Positive supercompilation	functional	online	unification	yes
Generalized partial computation	functional	online	constraint	no
Partial deduction	logic	online	unification	yes
Conjunctive partial deduction	logic	online	unification	yes

Table 2.2: A summary of the features of the program specialisation techniques.

quality of the residual program while maintaining an automatic procedure. This can be achieved by representing information as constraints and using constraint solving to exploit the additional information during specialisation.

Constraint solvers are based on well-established, efficient algorithms for determining whether a set of constraints is satisfiable. Incorporating constraint solving into the partial evaluation algorithm results in an efficient, decidable method for removing unreachable computations from the resulting residual program. This should have a positive effect on the size and efficiency of the specialised program, since the specialiser then has more information with which to make the reduce/residualise decision.

Constraint solving has been used previously in program specialisation. For example, a method for the BTA of the lambda calculus was developed using type constraints to represent the two-level expressions [Hen91]. In deforestation, an analysis to detect unsafe expressions approximates the set of terms encountered during the transformation by set constraints [Sei96]. This technique was refined to ensure the termination of higher-order deforestation [SS97].

Constraint solving has also been applied to the problem of propagating information during the transformation. For example, in supercompilation, negative information is propagated by restrictions, which can be represented by Herbrand (term) constraints [Tur86, GK93]. The ability to propagate negative binding constraints during partial evaluation was explored by Leuschel et al. in [LS97]. In addition, generalized partial computation has been defined using constraints and libraries of manipulation functions [Tak91]. On the other hand, the integration of modern constraint solving in an algorithm for partial evaluation has been under-exploited; exploring the use of advanced constraint solving in partial evaluation is one of the aims of this work.

In this section, the theoretical foundations of constraint domains and the operations over constraint domains are presented. Three constraint domains, Herbrand, linear arithmetic, and Boolean, are introduced and constraint solving operations over each domain and their union are defined.

2.2.1 Domains

Given some signature Σ, let $\mathcal{D}$ be a Σ-structure, the domain of the computation, and $\mathcal{L}$ be a class of Σ-formulae, the constraints. Let $(\mathcal{D}, \mathcal{L})$ be called a constraint domain [JM94].

As an example of a constraint domain, let Σ contain the constants 0 and 1, the function symbol $+$, and the predicate symbols $=$, $<$, and $\leq$. Let D be the set of Reals, and let $\mathcal{D}$ interpret the elements of Σ as usual, for example, $+$ means addition. Then $\mathcal{R}$ is the constraint domain of linear arithmetic over the Real numbers. The structure $\mathcal{R}$ maps ground constraints of $\mathcal{L}$ to the truth values **true** and **false**, based on their fixed interpretation in the structure.

The constraints of three specific domains are the focus of this work. The Herbrand constraint domain has two predicates, one for equality and one for disequality.

$t_1 == t_2$	syntactic equality
$t_1 \sim== t_2$	syntactic disequality

For any terms t_1 and t_2, the ground constraint $t_1 == t_2$ is true in the Herbrand constraint domain if the terms are syntactically equivalent up to renaming.

The Boolean constraint domain, $\mathcal{B}$, has two constants *tt* and *ff*. The primitive functions of the Boolean constraint domain are conjunction (&) disjunction ($\vee$) and negation ($\sim$). The predicates for the Boolean constraint domain are:

$t_1 =:= t_2$	Boolean equality
$t_1 =\sim= t_2$	Boolean disequality

The linear arithmetic constraint domain, $\mathcal{Q}$, has the primitive functions for addition ($+$), subtraction ($-$), multiplication ($*$) and division ($/$). At least one of the arguments to the primitive functions multiplication and division must be non-variable. The predicates for the linear arithmetic constraint domain are shown below.

$t_1 = t_2$	arithmetic equality
$t_1 < t_2$	arithmetic strict inequality
$t_1 \leq t_2$	arithmetic inequality
$t_1 \sim= t_2$	arithmetic disequality

Finally, given these three constraint domains, it is possible to form the union of the three constraint domains, the structure $f(\mathcal{Q}, \mathcal{B})$ [Stu91]. In this structure, it is possible to embed linear arithmetic or Boolean constraints in term constraints and have the embedded constraints evaluated in their particular domains. For example, the constraint $F(tt\ \&\ x) == F(x)$ evaluates to **false** if interpreted strictly in the Herbrand constraint domain, but considered in the union of the above constraint domains, it evaluates to the truth value **true** since the embedded Boolean constraint is extracted and evaluated in its own domain. That is, in the domain $f(\mathcal{Q}, \mathcal{B})$, the above constraint is separated into the two constraints $\{F(y) == F(z), tt\ \&\ x =:= x\}$. Since both of these constraints are true in their respective domains, the union of the constraints is true in $f(\mathcal{Q}, \mathcal{B})$.

2.2.2 Operations on Constraint Domains

Constraint domains, such as the Herbrand, Boolean, and linear arithmetic constraint domains, support the following operations [JM94]:

- Testing constraints for satisfiability. That is, for each constraint domain $(\mathcal{D}, \mathcal{L})$, $\mathcal{D} \models \tilde{\exists} c$, where $\tilde{\exists} c$ represents the constraint resulting from the existential quantification of all free variables of the constraint c.
- Simplifying constraints. Given a constraint c_0, the solver returns a simpler constraint c_1 such that $\mathcal{D} \models c_0 \leftrightarrow c_1$.
- Testing entailment of constraints. For example, testing if a constraint c_0 entails a constraint c_1 in the constraint domain $(\mathcal{D}, \mathcal{L})$, $\mathcal{D} \models c_0 \rightarrow c_1$, is supported.
- Projecting a constraint onto variables $x_1, \ldots, x_n$. It is intended that the solver returns the simplest constraint c_1 such that $\mathcal{D} \models c_1 \rightarrow \exists \bar{y}\ c_0$, where the variables $\bar{y}$ are the free variables of c_0 without the variables $x_1, \ldots, x_n$.
- Handling negated constraints, i.e. supporting the testing of satisfiability and entailment, simplification, and projection operations defined above for negated constraints.

For the union of the constraint domains, integrated operations must be performed for all the constraints in each of the domains. For example, the Herbrand constraint $F(x) \sim== F(y)$ and linear arithmetic constraint $x = y$ are each satisfiable in their domains, but the set of constraints $\{F(x) \sim== F(y), x = y\}$ in the union of the constraint domains is unsatisfiable. Therefore, the union of the constraint domains also supports a satisfiability check operation and a simplification operation.

- $sat_{HRB}(C) =$ **false** if the conjunction of Herbrand, Boolean, and linear arithmetic constraints C is unsatisfiable in $f(\mathcal{Q}, \mathcal{B})$.
- $sat_{HRB}(C) = C'$, otherwise, where C' is a simplification of the conjunction C.

The three domains have well-established algorithms for checking the satisfiability of constraints and projecting constraints onto free variables. For the Herbrand constraint domain, satisfiability is tested by the unification algorithm [JM94, Llo93]. Binary decision diagrams (or ordered Boolean decision diagrams) provide an efficient representation of Boolean expressions, which can be manipulated in order to test satisfiability or project the constraints onto free variables [Bry86, Bry92]. Recently, the satisfiability test for Boolean constraints has been implemented by applying Gröbner bases to Boolean algebras [SS88]. Typically, systems of linear equations, inequalities, and disequalities are simplified and shown to be unsatisfiable by the Simplex algorithm [Chv83]. Projection of linear arithmetic constraints onto free variables can be performed by Gaussian elimination [JM94]. It should be noted that both the Simplex algorithms and the Boolean constraint satisfiability tests have exponential worst case complexity.

For the Herbrand, linear arithmetic, and Boolean constraint domains, it is possible to handle negated constraints by computing its *admissible closure* [Stu91]. The domains described above are admissible closed, meaning that a negated constraint $\sim c(\bar{x}, \bar{y})$ has an associated disjunction of conjunctions of constraints $d(\bar{x}, \bar{z})$ such that $\mathcal{D} \models \exists \bar{x}\ (\sim \exists \bar{y}\ c(\bar{x}, \bar{y})\ \leftrightarrow\ \exists \bar{z}\ d(\bar{x}, \bar{z}))$.

2.3 The Rewriting-based Functional Logic Language

In this section, an overview of the development of functional logic programming languages is presented. Section 2.3.1 contains a brief review of functional logic programming. Section 2.3.3 presents an example of a rewriting-based functional logic language, the Escher language [Llo95]. The algorithm and examples in this thesis are presented in terms of a subset of Escher for simplicity. The syntax of this restricted language, called *E*, is presented in Section 2.3.4, and the semantics of the *E* language is formalised in Section 2.3.5.

2.3.1 Functional Logic Programming

Functional logic programming languages combine the features of the two main declarative programming paradigms: functional programming and logic programming [Han94, Han97]. This integration can be accomplished in two ways. In some cases, logic programming languages have been extended with function declarations [BE86, Fri85, Han90, MNRA92]. The well-established *narrowing* semantics provides a computational mechanism for these languages [Red85, BGM88]. On the other hand, functional languages can be extended by permitting logical variables in the terms of a computation [Llo95] or by adding set abstraction [DG89, RS82]. Rewriting-based functional logic languages are an example of the latter kind of integration.

The main difference between the rewriting and narrowing operational semantics is that rewriting simplifies a term by *matching* its redexes to statements in the program, while narrowing uses *unification* in order to compute answer substitutions for a term. That is, one-step rewriting ($\xrightarrow{r}$) can be defined as follows, from [Red85]:

1. If the program contains an equation $f\ t = d$ and there exists a substitution θ such that $t\theta = e$, then $f\ e \xrightarrow{r} d\theta$.
2. If $e \xrightarrow{r} d$ and e occurs in term c, then $c \xrightarrow{r} c[d/e]$, where $c[d/e]$ is the term c with the subterm e replaced by d.

On the other hand, one-step narrowing ($\xrightarrow{n}$) uses the most general unifier to compute an answer substitution.

1. If the program contains an equation $f\ t = d$ and there exists most general unifier of t and e $\theta \cup \varphi$ such that $t\theta = e\varphi$, then $f\ e \xrightarrow{n} d\theta$ with computed answer substitution φ.
2. If $e \xrightarrow{n} d$ with computed answer substitution φ and e occurs in term c, then $c \xrightarrow{n} c[d/e]\varphi$, where $c[d/e]$ is the term c with the subterm e replaced by d and φ is the computed answer substitution.

Therefore, narrowing more closely resembles the computational mechanism of logic programming. A set of answer substitutions is computed as a result of a narrowing reduction. On the other hand, rewriting deterministically reduces a term to its normal form. The following example illustrates the difference between the two operational semantics for a basic computation.

Example 2.3.1 Consider the following program P defining the list concatenation function.

```
Concat([],y)    = y.
Concat([h|t],y) = [h|Concat(t,y)].
```

The expression `Concat(x,[])` cannot be reduced using the rewriting computational mechanism. It will be delayed until it is instantiated sufficiently to be rewritten. On the other hand, the narrowing semantics computes the variable assignments for which the term reduces; using the terminology of [Red85], it is *narrowed* through a substitution so that it can be reduced in the program.

2.3.2 Partial Evaluation of Functional Logic Programs

To the best of the author's knowledge, this thesis documents the first study of the partial evaluation of rewriting-based functional logic programs. Concurrent with this work, a group of researchers have established the foundations for the partial evaluation of narrowing-based functional logic programs [AFV96a, AFV96b, AFJV97]. The procedure for the partial evaluation of narrowing-based functional logic programs shares many features with partial deduction and conjunctive partial deduction (§ 2.1.4). Incomplete narrowing trees are generated for the terms and subterms, and resultants are extracted from these trees. Local and global control of the algorithm ensures that the procedure terminates for all terms and programs. The global control for this technique is adapted from the approach of [GB91] for partial deduction. Local control is ensured by imposing an ordering on the terms in the narrowing tree [AFV96b].

2.3.3 The Escher Language

The Escher language, a rewriting-based functional logic language [Llo95, Llo], originally motivated this study. At first the Escher language was derived from the Gödel logic programming language [HL94], which was extended with higher-order features, function definitions, and equations instead of statements. Recently, the focus of Escher has changed; the language is now an extension of Haskell, sharing its syntax and structure.

The Escher language offers powerful features in a declarative setting, including higher-order capabilities, monadic I/O, set processing, logical operators, and concurrency. Unlike functional programming, partially instantiated terms can be computed in the hybrid language, while the deterministic computations set these languages apart from logic programming languages. Rewriting-based functional logic languages cannot perform function inversion; the computational mechanism is less powerful than that of logic programming languages. The language is polymorphic and strongly-typed.

The rewriting computational model simplifies terms by replacing subterms (redexes) in a term with their equivalent terms, as indicated by the equations in the program. Therefore, the value of a complex expression can be obtained by successive rewrites. Since Escher has no understanding of the intended interpretation of the program, an evaluation of the term is not possible by computation;

a term is *simplified* by rewriting. The soundness of Escher is guaranteed by the following property: if a term s rewrites to term t by an Escher computation, then s and t have the same value in the intended interpretation of the program.

The main extension of Escher over Haskell is the introduction of logical variables. In addition, the Escher language also provides a data type for sets and offers Boolean functions as primitive built-in functions of the language. These extensions are briefly discussed below.

Booleans

The Booleans module of the Escher language provides built-in functions for truth values `True` and `False`, logical connectives conjunction, `&&`, disjunction `||`, and negation `not`, and quantifiers `exists` and `forall`. The existential and universal quantifiers require a lambda expression immediately following the identifiers; that is, the expression `exists \x -> t` indicates the variable `x` is existentially quantified in the term `t`.

All of these elements of the Booleans module have associated rewrite rules. For example, consider the rewrite rules defining the equality (==) operator. The equation

```
 f x1 ... xn == f y1 ... yn = (x1 == y1) && ... && (xn == yn);
```

simplifies an equality to a conjunction of equalities of the arguments of the terms if the data constructors are the same. Otherwise, if the data constructors are different, the term rewrites to the truth value `False`:

```
f x1 ... xn == g y1 ... yn = False;.
```

Similarly, disequality simplifies to a negated equality expression:

```
x /= y = not (x == y);.
```

Equations exist for the other Boolean functions; details can be found in [Llo].

Sets

A set is implemented in Escher by associating it with a predicate. A term is a member of a set if the associated predicate maps the element to the truth value `True`. The notation $\{t_1, \ldots t_n\}$ is shorthand for the set abstraction $\{x \mid (x = t_1) \vee \ldots \vee (x = t_n)\}$ where x is not free in any t_i; likewise, $\{\}$ is equivalent to $\{x \mid False\}$ and $\{t\}$ means $\{x \mid (x = t)\}$. Basic set operations, such as subset, intersection, union, and cardinality are defined as higher-order functions which take a predicate as input.

```
module SportsDB(Person(...), Sport(...), likes) where {

data Person = Mary | Pete | Joe | Fred;

data Sport = Snooker | Football | Ultimate;

likes :: (Person, Sport) -> Bool;
likes =
      {(Mary, Snooker),
       (Mary, Ultimate),
       (Pete, Snooker),
       (Pete, Football),
       (Joe, Ultimate),
       (Joe, Football)};
```

Figure 2.1: Example Escher program using the sets data type.

Example Program

In order to demonstrate the simplification of terms that is performed by the rewriting mechanism in these languages, consider a small example Escher program from [Llo]. The program shown in Figure 2.1 demonstrates the use of the sets data type of the Escher language.

Example computations using the program in Figure 2.1 are shown below. Firstly, the term

`likes(Mary,x)`

simplifies to the term

`(x == Snooker) || (x == Ultimate)`.

The term

`forall \y -> y ‘in‘ {Snooker, Football} --> likes(x,y)`

requires the Boolean functions to be available by importing the Booleans module of the Escher system. The result of the computation of the term is

`(x == Pete)`.

Finally, for the set abstraction {`p | likes(Fred,s)`}, the result of the computation is the empty set, {}, since `Fred` is not defined in the `likes` function in the program (Figure 2.1).

2.3.4 Syntax of *E*

This section describes the syntax of a functional logic language called *E* that will be used throughout the description of this work. The syntax of the higher-order rewriting-based functional-logic language is based on the original definition of the Escher language as described in [Llo95]. The language is assumed to be polymorphic strongly-typed with a module structure. Furthermore, it is assumed that there is a library of built-in functions for *E*.

Definition 2.3.2 (Language *E* syntax)
Let D_V, D_C, D_P, D_F, D_G be finite, disjoint sets of variable names, constructor names, primitive function names, f-function names, and g-function names. Let $x \in D_V$, $c \in D_C$, $p \in D_P$, $f \in D_F$, and $g \in D_G$. Let n range over *non-variable terms*, t range over *terms* and s range over *statements* of *E*.

$$
\begin{array}{lcll}
n & ::= & c(t_1, \ldots, t_m) & \text{(constructors)} \\
 & | & p(t_1, \ldots, t_m) & \text{(primitive functions)} \\
 & | & f(t_1, \ldots, t_m) & \text{(f-functions)} \\
 & | & g(t_0, t_1, \ldots, t_m) & \text{(g-functions)} \\
 & | & \texttt{LAMBDA}[x](t) & \text{(lambda abstraction)} \\
 & | & t_1\ t_2 & \text{(application)} \\
 & | & \texttt{ITE}(t_1, t_2, t_3) & \text{(conditional)} \\
 & | & \{x \mid t\} & \text{(set abstraction)} \\
t & ::= & x \mid n & \\
s & ::= & f(x_1, \ldots, x_m) \Rightarrow t & \\
 & | & g(n_0, x_1, \ldots, x_m) \Rightarrow t'_0 & \\
 & & \vdots & \\
 & & g(n_k, x_1, \ldots, x_m) \Rightarrow t'_k &
\end{array}
$$

Variables in the language are denoted by identifiers beginning with a lowercase letter. All other identifiers begin with a uppercase letter. A constant is a 0-ary constructor function. The term $\texttt{ITE}(t_1, t_2, t_3)$ represents the conditional expression: *if* t_1 *then* t_2 *else* t_3. The term $\texttt{LAMBDA}[x](t)$ represents the lambda expression $\lambda x.t$.

A term with no variables is *ground*. A term containing only constructors, constants, and variables is called a *pattern*. Otherwise, the term is an *active term*. The sequence of free variables of a term t is $\mathcal{FV}(t)$; the free variables occur in the sequence $\mathcal{FV}(t)$ in the same order as they occur in the term t.

A *program* is a set of rewrite rules, in the form of statements $h \Rightarrow b$, where the term h is the head of the statement, and the term b is the body. The *definition* Def_P^H of a n-ary function *H* in a program *P* is the set of statements in *P* with head $H\ t$, where t is a tuple of length n.

The f-functions and g-functions differ in the form of their definitions in a program P. There can only be one rewrite rule in the definition of an f-function, since all arguments in the head of the schema statement for a f-function are variables. If there were more than one schema statement defining f in a program P, non-determinism would result. On the other hand, the schema statements in the definition of a g-function each have a term in the head of the statements, called the *term argument*, that determines which of the schema statements should be applied during the rewriting step. The g-functions are *strict* in their first argument, and this argument is *demanded* by the computation [Han97].

For a g-function, $TA(g, P)$ is the set of all term arguments in the definition of g in P. The term arguments of a g-function must be *pattern* and *instance* non-overlapping to ensure deterministic computations in *E*. It is assumed that the f-functions and g-functions appear with all of their arguments; partial application is represented by lambda abstraction.

The existential operator $\mathtt{SOME}[x](t)$ occurs in some example programs in this work. The representation of an existential quantifier is syntactic sugar for $\Sigma(\mathtt{LAMBDA}[x](t))$, where Σ is a primitive (built-in) function. Furthermore, the logical operators &, \/, and ~ are also built-in functions with defined rewrite rules (as in [Llo95]). The set of statements defining the primitive functions are assumed to form a confluent system.

The obvious restriction of this higher-order rewriting functional logic language is limiting the heads of statements defining g-functions to one term argument. This is a customary simplification of functional languages, and there are methods for translating definitions into this restricted form [Aug85]. However, in some examples, there will be more than one term argument in the definition of a function. Given a fixed computation rule for a rewriting-based language, the partial evaluation algorithm can be modified to handle g-functions with arbitrarily many term arguments.

Given a schema statement $h \Rightarrow b$, define $rhs(h \Rightarrow b) = b$. In terms of a set of statements A, $rhs(A)$ is the set of terms $\{b_i \mid h_i \Rightarrow b_i \in A\}$.

Definition 2.3.3 (standard term, $\mathcal{U}_P(t)$)
Given a program P, a term t is a *standard term* if $t = f(t_1, \ldots, t_n)$ or $t = g(t_0, t_1, \ldots, t_n)$, where f and g are defined in P. The set $\mathcal{U}_P(t)$ is the set containing all the standard subterms of t (may contain t if it is a standard term) wrt the program P.

The symbol ϵ denotes a "dummy term", a term that does not exist in any program.

Definition 2.3.4 (term metric)
Given a term t, let $|t|$ be the size of t, computed as follows:

$$\begin{array}{lcl}
|x| & = & 1 \\
|c(t_1, \ldots, t_n)| & = & 1 + \sum_{i=1}^{n} |t_i| \\
|p(t_1, \ldots, t_n)| & = & 1 + \sum_{i=1}^{n} |t_i| \\
|f(t_1, \ldots, t_n)| & = & 1 + \sum_{i=1}^{n} |t_i| \\
|g(t_0, t_1, \ldots, t_n)| & = & 1 + \sum_{i=0}^{n} |t_i| \\
|\mathtt{LAMBDA}[x](t)| & = & 1 + |x| + |t| \\
|t_1\ t_2| & = & 1 + |t_1| + |t_2| \\
|\mathtt{ITE}(t_1, t_2, t_3)| & = & 1 + |t_1| + |t_2| + |t_3| \\
|\{x \mid t\}| & = & 1 + |x| + |t|
\end{array}$$

The following naming conventions will apply in this thesis. Variables will be designated by subscripted terms x, y, w, z, v, function variables by f, g, terms by s, t, r, h, b, types by α, β, γ, substitutions by θ, ρ, ψ, sets by A, B, C, and function names by F, G, H.

Types

The higher-order logic which forms the foundation for the language *E* is based on Church's type theory. The *E* language is polymorphic strongly-typed.

Given a set S of primitive types, the set of types in the language is the smallest set containing the primitive types and closed under the type constructors $\rightarrow$ and $\times$. The primitive types 1 and o are always contained in S. A denumerable set of parameters $a, b, c, \ldots$ (type variables) is associated with the set S. The set of *types* can be defined inductively as follows, using the symbols $\rightarrow$ and $\times$.

Definition 2.3.5 (types)
Given a set of type constructors S, a type is defined as follows.

1. A parameter is a type.
2. For a type constructor c of arity n in S, if $\alpha_1, \ldots, \alpha_n$ are types, then $c(\alpha_1, \ldots, \alpha_n)$ is a type.
3. If α, β are types, then $\alpha \rightarrow \beta$ is a type.
4. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \ldots \times \alpha_n$ is a type.

A further formalisation of the type system of *E* is not required for the presentation of the partial evaluation algorithm. Typical methods for generating type declarations in the residual program will be applied to the programs of *E* [Gur93].

Modules

The module system of *E* shares its structure with the module system of Haskell. A module consists of a set of definitions of types and statements and a declaration that specifies which definitions are *exported*, that is, the definitions which are available for use by other modules. These statements are called *visible*. Modules using these definitions must *import* the module (via an import declaration).

A further discussion of the module structure of *E* program is outside the scope of this work. A method for handling the partial evaluation of programs with a modular structure, called flattening, is described in [Gur93]. The modular structure of the program is altered by promoting and demoting symbols in order to make their declarations accessible to other modules in the program. In this work, it is assumed that all programs are flattened in a similar manner before partial evaluation; all example programs will have only a single module.

2.3.5 Semantics of *E*

In this section, the operational semantics of the *E* functional logic language is defined.

Informally, a term t in *E* is rewritten by instantiating one of the statements of the program so that the head of the statement is identical to the redex in t. At this point, the redex in t is replaced by the instantiated body of the statement. This continues until the term reaches a normal form, which will be defined later in this section. This process is called *reduction*.

The partial evaluation procedure presented in this thesis is intended to be independent of a particular reduction strategy. That is, whether or not the rewriting language selects the outermost or innermost redex is irrelevant to the specialisation algorithm. On the other hand, a selection rule must be specified for the language *E*, so that later examples are clear to the reader. The reduction strategy for *E* will be normal order reduction, but the generality of the algorithm will be emphasised by indirectly referring to this selection rule via the notions of *selected term* and *evaluation context*. These are the only concepts that actually depend on the selection rule. The definitions of the selection function, $\mathcal{S}$, and selected terms are deferred until Section 4.3.1.

Definition 2.3.6 (Evaluation context, redex, free term)

Let e range over *contexts*, r range over *redexes*, and b over *free terms*.

$$\begin{array}{lcl}
e & ::= & [] \\
 & | & c(b_1, \ldots, b_{i-1}, e, t_{i+1}, \ldots, t_n) \\
 & | & p(b_1, \ldots, b_{i-1}, e, t_{i+1}, \ldots, t_n) \\
 & | & g(e, t_1, \ldots, t_n) \\
 & | & \texttt{LAMBDA}[x](e) \\
 & | & e\ t_2 \\
 & | & t_1\ e \\
 & | & \texttt{ITE}(e, t_2, t_3) \\
 & | & \{x \mid e\} \\
r & ::= & f(t_1, \ldots, t_n) \\
 & | & g(t_0, t_1, \ldots, t_n) \text{ if } \exists s \in TA(p, g)\ :\ s\theta = t_0 \\
b & ::= & x \mid c(b_1, \ldots, b_n)
\end{array}$$

Let $e[t]$ denote the replacement of [] in context e by term t.

Informally, a context e is an expression with a *hole*, denoted by $[]$. The hole acts as a placeholder for a subterm. Thus, the expression $e[t]$ is the term resulting from replacing the hole in the context e with the term t. Unlike substitution, free variables in t may become bound in $e[t]$. If the term $e[t]$ has no free variables, then e is a *closing context* for t.

Definition 2.3.7 (substitution, instance)
A substitution θ maps variables to terms. If $\theta = \{x_i := t_i\}$, $x_i\theta = t_i$. When a substitution is applied to a term, simultaneous substitution is performed, renaming bound variables as necessary to avoid variable capture. A term t is an instance of term s if there exists a substitution θ such that $t = s\theta$.

A *renaming* substitution for a term t is a variable pure substitution $\{x_i := y_i\}_{i=1}^{n}$ where each x_i is a member of the set of variables of t, V, and for all variables y_i,y_j, $y_i, y_j \notin V$ and $y_i \neq y_j$.

Rewriting of a term $t = e[r]$ is performed by matching the head of an instance of a statement with r. The statements in *E* programs are statement *schemas*, meta-expressions which represent a collection of the instances of that statement. In general, in a rewriting-based functional logic language, a term t_{j+1} is obtained from a term t_j by a *computation step* if the following are satisfied:

1. The set of redexes of t_j, L_j, is a non-empty set.

2. For each redex r_i in L_j such that $t_j = e[r_i]$,
    - r_i is identical to the head h_i of some instance $h_i \Rightarrow b_i$ of a statement schema (rule), and
    - $t_{j+1} = e[b_i]$, the result of replacing r_i by b_i in t_j.

In order to maintain independence from the reduction strategy, the above definition of a computation step allows terms with several redexes (i.e. in the case of a parallel reduction strategy). In the case of the *E* language, there is only one selected term r for the term $t_j = e[r]$, and the result of the computation step is $t_{j+1} = e[b]$.

A *computation*, $t \Rightarrow^* t_n$, is a sequence of terms $\{t_i\}_{i=1}^n$ such that t_{i+1} is obtained by a computation step from t_i and t_n contains no rewritable subterms. Otherwise, the term t reduces via infinitely many computation steps, abbreviated as $t \Rightarrow^* \bot$. The notation $t \not\Rightarrow$ indicates that the term t is not rewritable in the program.

A *partial computation* is the transitive closure of finitely many computation steps. The result of a partial computation may have rewritable subterms.

Definition 2.3.8 ($\Rightarrow^n$)
Let $t_1 \Rightarrow^n t_n$ represent the equation resulting from the partial computation $\{t_i\}_{i=1}^n$.

A term t has a *trivial* computation in P if it is not rewritable in P. On the other hand, if a term t reduces in at least one computation step in the program P, t has a *non-trivial* computation in P.

Chapter 3

Partial Evaluation of Functional Logic Programs

In this chapter, the theoretical foundations for the partial evaluation of rewriting-based functional logic programs are presented. The motivation behind the design of the partial evaluation procedure is discussed in Section 3.1. The partial evaluation procedure is defined in Section 3.2. The correctness of the algorithm is established in Sections 3.4 and 3.5. Finally, this procedure is compared with existing techniques for the partial evaluation of declarative programs in Section 3.6.

3.1 Motivation

In this section, the design of the partial evaluation procedure is discussed in general terms. The main inspiration for the procedure resulted from the ability of rewriting-based functional logic languages to simplify terms containing variable arguments. However, the computational mechanism alone is not sufficient to partially evaluate rewriting-based functional logic programs; the extension of the interpreter necessary to ensure correct residual programs is described later in this section.

As discussed in Section 2.1, the program transformation performed by partial evaluation involves specialising a program with respect to some fixed (static) data, resulting in a program which is operationally equivalent to the original program for this fixed data. Typically, this static input to the program is represented in an expression, such as a term [Sør96] or atom [LS91]. In the case of functional program specialisation, all variable arguments in the expression represent dynamic data; the run-time instantiation of the term must be ground. On the other hand, the distinction between dynamic data and variable arguments is lost during logic program specialisation.

Rewriting-based functional logic languages incorporate the computation mechanisms of functional and logic programming by ensuring deterministic computations while permitting the simplification of partially instantiated expressions. Like many other transformation procedures, partial evaluation of a rewriting-based functional logic program is defined with respect to a term. As above, the static input is encapsulated in this expression, and dynamic data are represented by variable arguments in the term. There exists a distinction between variable arguments and dynamic data in functional logic partial evaluation. The variable arguments which are required by the computation represent dynamic data which will be available at runtime. Otherwise, it is not possible to categorise the variable arguments into those representing dynamic data and those representing run-time variable arguments.

In other words, there is an implicit required *instantiation level* for a term to be computed to a pattern ($\S$ 2.3.4). Computation in a rewriting-based functional logic language will stop when there are no subterms of the term that are syntactically equal to the head of an instantiation of any schema statement in the program. For example, in *E*, given a term $G(x, t_1, \ldots, t_n)$ for a given g-function G, the term cannot be syntactically equal to the head of any statement in the definition of G, since the term argument is a variable. As noted earlier, this argument is *demanded* by the computation ($\S$ 2.3.4).

Therefore, the computation mechanism of rewriting-based functional logic languages allows the simplification of partially instantiated terms, but requires a certain instantiation level in order to fully evaluate the term. Since a program is partially evaluated with respect to a partially-instantiated term, as the dynamic data is represented by variable arguments, the partial evaluator exploits the natural power of the language to simplify these terms. This is similar to partial deduction, in which clauses are extracted from the non-failing derivations ($\S$ 2.1.4).

However, because of the required instantiation level, using the computation mechanism of the language on its own is not adequate for performing the partial evaluation of functional logic programs. It is necessary for the *residual* program, the program generated by the partial evaluator, to be able to correctly evaluate any run-time instantiation of the term used in the specialisation with respect to the semantics of the original program. In order to ensure the residual programs have this quality, the computational mechanism of the target language is extended with a *restart step*. This step intervenes when the unfolding is prematurely stopped by the lack of dynamic data in the expression. The computation is *restarted* by replacing the final term of the stopped computation with the minimal instantiations of the term that are reducible in the language. An example of the use of a restart step is shown in Figure 3.1.

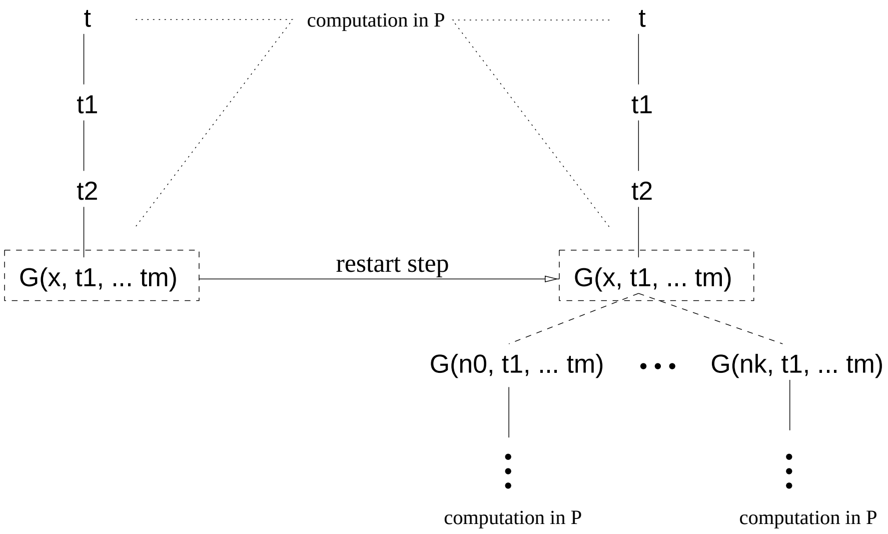


Figure 3.1: Extending the computational mechanism with the restart step to guarantee correct residual programs. In the left computation, the last term $G(x, t_1, \ldots, t_m)$ cannot be reduced. The variable x represents dynamic data that will be supplied at run-time. During the restart step, the rewritable instantiations of the term are computed, so unfolding can continue.

The computation mechanism extended with the restart step forms the basis of the partial evaluator for rewriting-based functional logic programs. The theory of the partial evaluation procedure and its correctness will be covered in the remainder of this chapter. An algorithm for the partial evaluation of functional logic programs is the subject of Chapter 4.

3.2 Partial Evaluation of Functional Logic Programs

In this section the theoretical foundations for the partial evaluation of rewriting-based functional logic programs are presented. The framework of the partial evaluation transformation is similar to that for logic program specialisation [Kom81, LS91]; in fact, this is quite natural, since both transformations use the computational mechanism of the language in order to perform unfolding. Therefore, similar to partial deduction, *resultants* are defined to be statements generated from partial computations.

Definition 3.2.1 (resultant)
Let P be a program, and t a term in E. Let $(t = t_0, \ldots, t_n)$, $n > 0$, be a partial computation of t in P. Then, the statement $t \Rightarrow t_n$ is a *resultant* of t in P.

Resultants reduce an n-step computation to a one-step computation. By definition, resultants do not exist for terms which have trivial computations, that is, computations for which $n = 0$. This prevents generating looping statements $t \Rightarrow t$, which will cause non-termination in the residual program.

For any term t and program P, the *restart terms* of t are the least instantiations of t that have non-trivial computations in P. In other words, the term t reduces by at least one computation step in the program P. This ensures a resultant exists for every restart term of t.

Definition 3.2.2 (restart terms)
Let P be a program, t a term. The set of *restart terms* for t wrt P is the set S of terms such that for all substitutions θ, $t\theta \in S$ if the following conditions hold.

- $t\theta$ has a non-trivial computation in P, and
- for all substitutions φ, if $t\varphi$ has a non-trivial computation in P, then $t\varphi$ either is or rewrites to an instance of some $t\theta$ in S.

According to the above definition, if the computation of t is non-trivial, then the set of restart terms for t wrt P is simply $\{t\}$. The set of restart terms is uniquely determined by the program P and the term t given a defined computation rule for the source language of the transformation.

Lemma 3.2.3 Given a program P and a term t, the set of restart terms for t wrt P is unique.

Given the computation mechanism of the *E* language, the following lemma specific to the *E* language is a direct consequence of Definition 3.2.2.

Lemma 3.2.4 Given an *E* program P and term t, let S be the set of restart terms for t wrt P.

1. For all terms $s \in S$, there exists a subterm s' in s such that s' is identical to $h_i\psi$ for some schema statement $h_i \Rightarrow b_i$ in P and some substitution ψ.
2. For all terms $t\theta \in S$, either t has a non-trivial computation in P and $\theta = \{\}$ or there is an assignment $x := t'$ in θ where x is the first demanded argument of t.

Proof Straightforward from the definition of the *E* language. □

Recall that an argument of a function H is *demanded* if there is a non-variable term in that argument position in the head of at least one of the statements in the definition of H in a program P (§ 2.3.4).

For example, in the *E* language, the term argument of g-functions is demanded. No arguments of an f-function are demanded during a computation.

In Definition 3.2.2, the instantiation is required to be minimal in order to ensure that the residual program is not more specific than is required. If the residual program is too specific, terms that have computations in the original program may not have computations in the residual program.

Example 3.2.5 The following program P contains a definition of the list concatenation function.

```
Concat([],y)    => y.
Concat([h|t],y) => [h|Concat(t,y)].
```

In order to generate the residual definition of $t =$ `Concat(Concat(x,y),z)` wrt P, first compute the set of restart terms for t wrt P. The term t has a trivial computation in P. The following instantiations of t:

$s_1 =$ `Concat(Concat([],y),z)` and
$s_2 =$ `Concat(Concat([h|t],y),z)`

are the restart terms for the term t. In other words, these terms are the only instantiations of t that satisfy the two conditions of Definition 3.2.2:

- Both terms s_1 and s_2 have non-trivial computations in P.
- In both cases, the terms s_1 and s_2 represent the minimal substitutions in order to ensure non-trivial computations in P.

Therefore, the terms s_1 and s_2 satisfy the conditions of Lemma 3.2.4.

- Both terms s_1 and s_2 have subterms that are identical to the head of a schema statement of P, namely the subterms `Concat([],y)` and `Concat([h|t],y)`, respectively.
- The substitutions $\{$`x := []`$\}$ and $\{$`x := [h|t]`$\}$ both contain a binding for the first demanded argument of t, `x`.

In summary, resultants are statements resulting from non-trivial computations and restart terms are instantiations of a term that have non-trivial computations in a program. Putting these two concepts together, a *residual definition* of a term is the set of resultants associated with the restart terms for the given term.

Definition 3.2.6 (residual definition)
Let P be a program, t be a term. Let $S = \{s_1, \ldots, s_n\}$ be the set of restart terms for t wrt P. Then, a *residual definition* of t wrt P is a set of resultants $\{r_1, \ldots r_n\}$, where r_i, $1 \leq i \leq n$, is a resultant of s_i in P.

If the set of restart terms for t wrt P is non-empty, the residual definitions of t wrt P have at least one resultant. Clearly, if the set of restart terms for t wrt P is empty, then the residual definition of t wrt P is empty.

Example 3.2.7 Given the program P of Example 3.2.5 above, a residual definition of $t =$ `Concat(Concat(x,y),z)` wrt P is a set of residual statements associated with the restart terms of t. Two resultants associated with the restart terms of t are

```
     Concat(Concat([],y),z) => Concat(y,z).
```

and

```
     Concat(Concat([h|t],y),z) => [h|Concat(Concat(t,y),z)].
```

Note that the second resultant could also be the statement

```
     Concat(Concat([h|t],y),z) => Concat([h|Concat(t,y)],z).
```

since the result of a partial computation is not required to be a term in normal form. On the other hand, constructing the resultants from computations (§ 2.3.5) often generates optimal residual programs.

The resultants are not necessarily *E* schema statements, since the head of the resultant is an arbitrary term. Renaming is necessary to convert the resultants into schema statements. The following definitions are extensions of those of [LSdW96]. The renaming function and translation function, as defined below, describe general properties of the renaming operation.

Definition 3.2.8 (renaming function)
Let P be a program. A *renaming function* for a set of terms A wrt P is a mapping σ which maps from terms in A to terms such that for any term $t \in A$:

- If $\mathcal{FV}(t) = (x_1, \ldots, x_n)$, then $\mathcal{FV}(\sigma(t)) = (x_1, \ldots, x_k)$ where $1 \leq k \leq n$, and
- For all t, t', $t \neq t'$, the outermost functions of $\sigma(t)$ and $\sigma(t')$ are different from each other, different from any built-in functions, and different from any other functions in P.

Recall that $\mathcal{FV}(t)$ is a sequence of the free variables of t; the variables occur in the sequence in the order that they occur in the term t. This ensures that the demanded argument of t remains the demanded argument of $\sigma(t)$. The definition above allows the sequence of free variables of the renamed function to be a subsequence of the original function; this permits the elimination of repeated or redundant variables. On the other hand, eliminating free variables during the renaming must be done carefully in order to ensure the program semantics are left unaltered. A study of redundant argument filtering in the context of logic program specialisation is available in [LS96]. In this work, it will be assumed that $k = n$.

Definition 3.2.9 (ordering function)
An *ordering* function ω is a mapping from sets of terms to sequences of terms such that for all sets A, $s \in A$ iff s occurs once in $\omega(A)$.

That is, the sequence $\omega(A)$ is simply an ordering of the terms of A. No other terms occur in the sequence and no elements of A are repeated in the sequence $\omega(A)$.

A *translation function* applies the renaming function σ for a set of terms A to arbitrary terms: if an instance of the term exists in A, it is renamed using σ. An ordering is imposed on the set of terms A to ensure that the translation function is deterministic. In the following definition, $\rho_\sigma(\theta)$ is the application of the translation function to all terms in the substitution θ, i.e. $\rho_\sigma(\{x := t\}) = \{\rho_\sigma(x) := \rho_\sigma(t)\}$.

Definition 3.2.10 (translation function)
A *translation function* based on a renaming function σ for a set of terms A and ordering ω is a mapping ρ_σ from terms to terms such that for any term t:

- if t is an instance of a term s_j in $\omega(A)$, say $s_j\theta = t$, and t is not an instance of any s_i, $i < j$, then $\rho_\sigma(t) = \sigma(s_j)\rho_\sigma(\theta)$.
- if $t = h(t_1, \ldots, t_n)$ is not an instance of a term in A, then $\rho_\sigma(t) = h(\rho_\sigma(t_1), \ldots, \rho_\sigma(t_n))$.
- otherwise, $\rho_\sigma(t) = t$.

Finally, the partial evaluation of a program with respect to a set of terms is defined.

Definition 3.2.11 (partial evaluation)
Let P be a rewriting-based functional logic program. Let $A = \{t_1, \ldots, t_m\}$ be a set of terms with associated residual definitions $R_1, \ldots R_m$. Let σ be a renaming of A, ω be an ordering of A, and ρ_σ be a translation function based on σ and ω. Then, the *partial evaluation* of P wrt A is the program $P_A = \{\rho_\sigma(h) \Rightarrow \rho_\sigma(b) \mid h \Rightarrow b \in \bigcup_{i=1}^{m} R_i\}$.

In other words, the schema statements of the residual program P_A, the program resulting from the partial evaluation of P wrt A, are the translated statements of the union of residual definitions for each term in A.

The definition of partial evaluation above describes a program which contains partial evaluations of the terms in A. This definition follows the structure of that for partial deduction as in [LS91]. This definition does not necessarily define a procedure that ensures the residual program P_A is operationally equivalent to the original program. The conditions to guarantee that partial evaluation generates correct residual programs will be presented in the next section, Section 3.3. These conditions relate the construction of the set A to the terms in the residual statements in P_A.

The finiteness of the partial computations, on which the resultants are based, and the finiteness of the set A guarantee the termination of partial evaluation as defined above.

Example 3.2.12 Consider again the program from Example 3.2.5 containing a definition for list concatenation. The partial evaluation of P with respect to the set of terms

$A = \{$`Concat(x,y), Concat(Concat(x,y),z)`$\}$.

is computed as follows.

For each term t in A, generate the set of restart terms for t. Applying the substitutions $\{x := [h|t]\}$ and $\{x := []\}$ to the term `Concat(x,y)` results in two restart terms satisfying the two conditions of Definition 3.2.2. Therefore, the set of restart terms for the term `Concat(x,y)` is

$S_1 = \{$`Concat([],y), Concat([h|t],z)`$\}$.

Similarly, the set of restart terms are constructed for the term `Concat(Concat(x,y),z)`. As shown earlier in Example 3.2.5, the set of restart terms for this term is

$S_2 = \{$`Concat(Concat([],y),z), Concat(Concat([h|t],y),z)`$\}$.

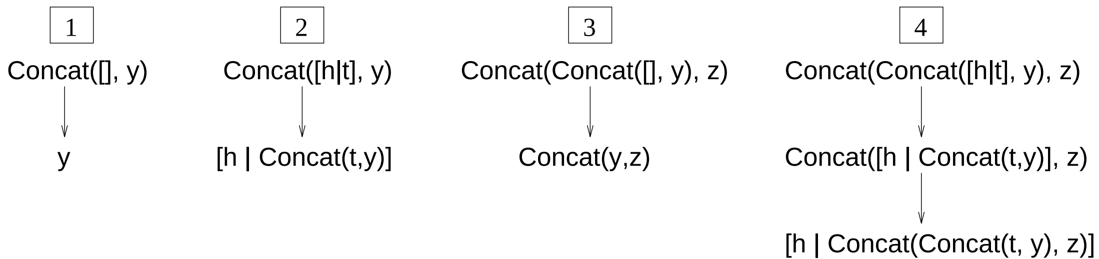
For each term s in S_1 and S_2, a partial computation of s in P is generated. Example partial computations and associated resultants for the terms in S_1 and S_2 are illustrated in Figure 3.2.

The fourth resultant in Figure 3.2 illustrates the advantage of applying partial evaluation in this case. An intermediate list is constructed and then destroyed during the computation of `Concat(Concat([h|t],y),z)`. The resultant extracted from this partial computation allows a computation of this term in the residual program P_A that avoids the construction of this intermediate list, thus saving time and memory.

Given a renaming σ for A, defined as

$\sigma = \{$`Concat(x,y)` $\mapsto$ `C(x,y)`, `Concat(Concat(x,y),z)` $\mapsto$ `CC(x,y,z)`$\}$,

an ordering ω such that



$\boxed{1}$ Concat([], y) => y

$\boxed{2}$ Concat([h|t], y) => [h | Concat(t,y)]

$\boxed{3}$ Concat(Concat([], y), z) => Concat(y,z)

$\boxed{4}$ Concat(Concat([h|t], y), z) => [h | Concat(Concat(t, y), z)]

Figure 3.2: Example partial computations and associated resultants for the double-append example.

$$\omega(A) = (\texttt{Concat(Concat(x,y),z)}, \texttt{Concat(x,y)})$$

and translation function ρ_σ based on σ and ω, the resulting residual program, P_A, is composed of the following set of schema statements.

```
C([],y)        => y.
C([h|t],y)     => [h|C(t,y)].
CC([],y,z)     => C(y,z).
CC([h|t],y,z) => [h|CC(t,y,z)].
```

What effect does the ordering function have on the resulting residual program? Suppose the ordering in this example was reversed, that is, the term `Concat(x,y)` was the first term of the sequence $\omega(A)$. In this example, the term `Concat(Concat(x,y),z)` is an instance of both terms in A. Based on this new ordering, the schema statement of P_A will be renamed to:

```
C(C([h|t],y),z) => [h|C(C(t,y),z)].
```

With respect to the *E* language, this renaming is correct, but it is not the optimal renaming in terms of efficiency. If the language restricts term arguments to those having outermost constructors, clearly this renaming is not correct.

3.3 The Correctness of the Transformation

In the following sections the correctness of the theoretical framework for the partial evaluation of rewriting-based functional logic programs is established for various program domains.

Proving the correctness of the program transformation ensures its *soundness* and *completeness*. That is, in terms of programs and computations, the correctness of the procedure guarantees:

- The computation of the term t in the original program P is finite and results in t' if and only if the computation of the translation of t in the residual program is finite and results in the translation of t', or
- The computation of a term t in the original program is infinite if and only if the computation of the translation of t in the residual program is infinite.

An intuitive approach to proving the correctness of the partial evaluation framework is to relate the computations in the two programs. Since the statements of the residual program are created directly from partial computations in the original program, it should be possible to "follow" the computation in the residual program by considering these partial computations implicitly stored in the residual definitions. If the computations are equivalent or there exists a mapping between terms of the two computations, then the correctness of the transformation is established. These underlying computations used to generate the statements of the residual program are called *base computations*.

Definition 3.3.1 (base computation)
Let P_A be a residual program resulting from the partial evaluation of the program P with respect to the set of terms A. Let $h \Rightarrow b$ be a statement in P_A such that h is renamed using term $c \in A$, i.e. $h = \rho_\sigma(c\theta)$ for some substitution θ. Then, the *base computation* of $h \Rightarrow b$ is the partial computation $c\theta \Rightarrow^n s$, where $b = \rho_\sigma(s)$.

Therefore, the base computation of a statement in the residual program is simply the partial computation in the original program used to generate its associated resultant. For example, in Example 3.2.12, the base computation of the statement:

```
    CC([h|t],y,z) => [h|CC(t,y,z)].
```

is the partial computation of `Concat(Concat([h|t],y),z)` in P used to generate the resultant. In this case, this is the partial computation of `Concat(Concat([h|t],y),z)` shown as the fourth computation in Figure 3.2.

So far, very few restrictions have been imposed on the rewriting-based functional logic language that is the target language of the transformation. For the E language, the term arguments occurring in the definition of a g-function are required to be pattern and instance non-overlapping. On the other hand, in order to prove the correctness for rewriting-based functional logic languages in general, this restriction is relaxed and only confluence of the program is required.

Therefore, the study of the correctness of the partial evaluation procedure is divided into two sections. In Section 3.4, the correctness of the transformation is proved for programs in which the pattern and instance non-overlapping condition is imposed. Then, in Section 3.5, the correctness of the transformation as applied to confluent systems is addressed. In each case, the aim is to show that the computations in the original program and the residual program can be related, and in some cases, are equivalent.

3.4 Proving Correctness for Non-Overlapping Programs

The correctness of the transformation for programs in which the statements are pairwise pattern and instance non-overlapping is established in this section. The subject of Section 3.4.1 is a restrictive version of the correctness theorem in which every term of the program is an instance of a term in A, but the elements of the instance substitution are limited to pattern terms (§ 2.3.4). Then, in Section 3.4.2, a stronger version of the correctness theorem is presented; this version allows the run-time term to contain nested terms of A.

3.4.1 Basic Correctness

As noted in the previous section, the approach to proving the correctness of the transformation is to relate the computation of the term t in the original program to the computation of the translated t in the residual program. An illustration of the approach using the simple double-append benchmark from Example 3.2.12 is presented in the following example.

Example 3.4.1 Consider the partial evaluation of the list concatenation program P with respect to the set of terms A of Example 3.2.12. Let the statements of P be indexed as follows (the indices are shown in parentheses on the right).

```
Concat([],y)     => y.                    (c1)
Concat([h|t],y)  => [h|Concat(t,y)].    (c2)
```

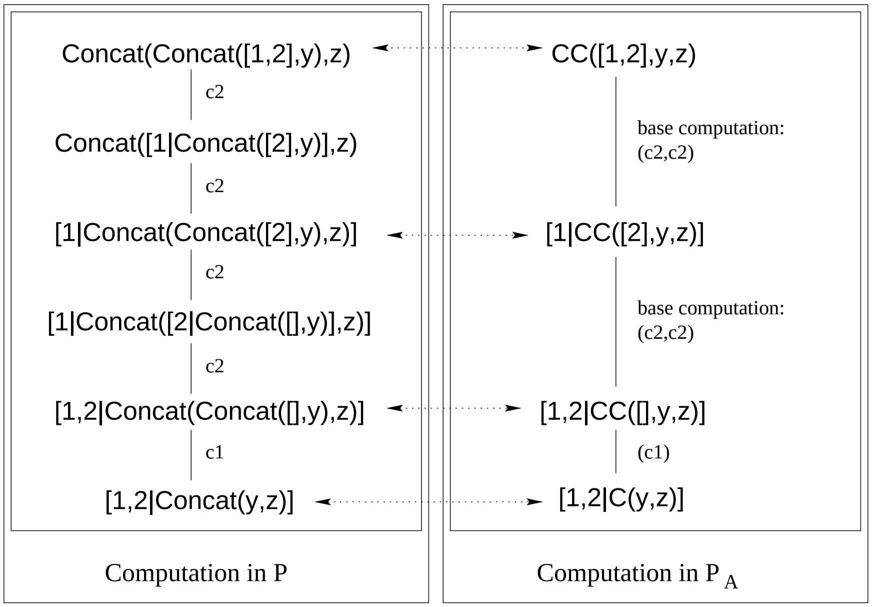


Figure 3.3: Comparing the expanded computation in the residual program and the computation in the original program. The sequence obtained by composing the base computations of the statements used in the computation in P_A is the same sequence of statement identifiers as occurs in the computation in P.

Now, consider the computation of the term `Concat(Concat([1,2],y),z)` in P and its renaming `CC([1,2],y,z)` in P_A. The computations of these terms in the respective programs are shown graphically in Figure 3.3. In this figure, the computation in the residual computation has been annotated with the base computations of the statements used in the computation. Clearly, the sequences of statements of the original program used to simplify the term is equivalent in the computation in P and the computation in P_A when the base computations are considered. In addition, the computations follow the same "path"; that is, in each case, the same redexes are chosen and it is possible to identify an injective mapping between terms in the computation in P and the renamed terms in the computation in P_A.

For this example, *any* instance of a term in A will have *exactly* the same computation in the residual program, considering the base computations, as in the original program. The order of evaluation in the term is unchanged by the transformation.

This section contains a proof of the correctness of the procedure for a limited class of terms and programs. The terms of the computation are restricted to instances of restart terms of the terms in A, in which the instances contain only pattern terms. Then, correctness is established by proving the relationship between the computation in the original program and the residual program.

One might argue that this is an overly restrictive class of terms and programs, of no practical use. On the other hand, in most partial evaluation cases, the run-time dynamic data are simply constants, which are pattern terms of the language. Typically, the underlying functionality of the specialised program is fixed at compile-time. *Passive* substitutions are substitutions which only contain pattern terms.

Definition 3.4.2 (passive substitution)
A substitution θ is a *passive substitution* if for all bindings $x := t$ in θ, t is a pattern, that is, t contains only constructors, constants, and variables. If a term t is an instance of a term s by a passive substitution, t is a *passive instance* of s.

The following definition of A-*passive* terms allows the instances of the terms in A to be embedded in constructor terms.

Definition 3.4.3 (A-passive)
Let $A = \{s_1, \ldots, s_n\}$ be a set of terms, and t be a term. Let the set A_R contain the restart terms for each s_i in A. Then t is A-passive if either t is a passive instance of a term in A_R, or $t = c(t_1, \ldots, t_n)$ for A-passive subterms $t_1, \ldots, t_n$.

The following lemma establishes the connection between computations of terms and the computations of their instances. If a term t is computed in a program P in which the heads of statements are pattern and instance non-overlapping, then if t reduces in finitely many steps to a term t' in P, any given instance of t, $t\theta$, will reduce in finitely many steps to $t'\theta$ in P.

In this section (§ 3.4), it is assumed that the non-overlapping property holds for statements in the definitions of the built-in functions as well as the user-defined definitions. Recall that the correctness of transforming confluent programs is the subject of the next section (§ 3.5).

Lemma 3.4.4 Let P be a program and t, t_n be terms such that $t \Rightarrow^n t'$ in P. Then, given any substitution θ, $t\theta \Rightarrow^n t'\theta$ in P.

Proof By induction on the length of computation n.
$n = 1$: Assume $t \Rightarrow t_1$. Without loss of generality, assume that t has one redex r, such that $t = e[r]$. By definition of a computation step, $r = h_i\varphi$ for some statement $h_i \Rightarrow b_i$ in P (where P is either simply the user-defined program or the user-defined program composed with the modules defining the primitive functions), and $t_1 = e[b_i\varphi]$.

Since the statements of P must be term non-overlapping, if r matches some statement head h_i in P, then for all substitutions θ, $r\theta$ must match h_i. Furthermore, the redex is unchanged in all instances of t. Therefore, $t\theta = (e[r])\theta = e\theta[r\theta] = e\theta[h_i\varphi\theta] \Rightarrow e\theta[b_i\varphi\theta] = (e[b_i\varphi])\theta = t_1\theta$.

Induction step: Straightforward from case $n = 1$. □

This lemma does not hold for arbitrary confluent programs, since the instance $r\theta$ may match more than one statement head in the program P. However, confluence of a program is also sufficient to prove the soundness of the transformation (§ 3.5).

Soundness

Proving the basic soundness of the procedure requires showing that the result of the computation of an A-passive term t in the residual program P_A is the same as the translated result of computing t in the original program P.

Generally, the proof proceeds as follows. First, show that if the term $\rho_\sigma(t)$ reduces to a term $\rho_\sigma(t')$ in P_A by a partial computation, then the result of the computation of t' in P is the same as the result of the computation of t in P. A graphical illustration of this approach is presented in Figure 3.4.

Then, show that a term is not rewritable in P if it is not rewritable in P_A, and vice versa. This establishes the soundness of the procedure.

Theorem 3.4.5 Let P be a program, A a set of terms, and t a term. Let P_A be the partial evaluation of P wrt A such that $P_A \cup \{t\}$ is A-passive. Then, if $\rho_\sigma(t)$ has a partial computation in P_A with result $\rho_\sigma(t')$ and $t' \Rightarrow^* t_n$ in P, then $t \Rightarrow^* t_n$ in P.

Proof
Let $\rho_\sigma(t) \Rightarrow^m \rho_\sigma(t')$. That is, $\rho_\sigma(t)$ has a finite computation of m-steps resulting in $\rho_\sigma(t')$ in P_A. Show if $t' \Rightarrow t_n$ in P, then $t \Rightarrow^* t_n$ in P by induction on the length of computation in P_A, m.

Case $m = 1$: For the base case, show that if $\rho_\sigma(t) \Rightarrow^1 \rho_\sigma(t')$ in P_A and $t' \Rightarrow^* t_n$ in P, then $t \Rightarrow^* t_n$ in P.

Assume $\rho_\sigma(t) \Rightarrow^1 \rho_\sigma(t')$, using some schema statement $h^A \Rightarrow b^A$ in P_A. Therefore, $\rho_\sigma(t) = e^\rho[r^\rho]$ for some context e^ρ and redex r^ρ, where $r^\rho = h^A\phi$ for some substitution ϕ. By definition of a computation step, $\rho_\sigma(t') = e^\rho[b^A\phi]$.

Furthermore, assume $t' \Rightarrow^* t_n$ in P. Then, since computations in P are deterministic, it remains to

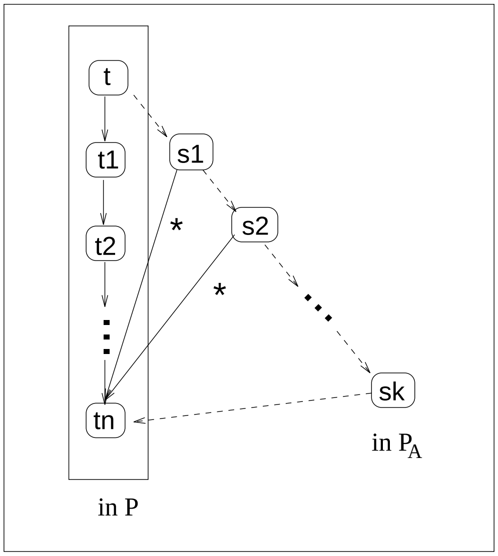


Figure 3.4: Proving soundness for the partial evaluation procedure. The solid lines represent computations in the original program P and the dashed lines represent computations in the specialised program P_A.

show that $t \Rightarrow^j t'$ in P.

By definition, the schema statement $h^A \Rightarrow b^A = \rho_\sigma(c\theta) \Rightarrow \rho_\sigma(c_k)$ for some $c \in A$ and some substitution θ such that $c\theta$ is a restart term for c in P and $c\theta \Rightarrow^k c_k$ is a partial computation in P. Therefore, $\rho_\sigma(t) = e^\rho[\rho_\sigma(c\theta)\phi] \Rightarrow e^\rho[\rho_\sigma(c_k)\phi] = \rho_\sigma(t')$.

Since t is A-passive, $\rho_\sigma^{-1}(\rho_\sigma(c\theta)\phi) = c\theta\phi$ (passive substitutions are unchanged by the translation function). Therefore, $t = e[c\theta\phi]$ and $t' = e[c_k\phi]$ for some context e. Furthermore, since the demanded argument is not changed by renaming, either $c\theta\phi$ is the redex of t or contains the redex of t. Thus, it remains to show that $e[c\theta\phi] \Rightarrow^k e[c_k\phi]$. This is a direct result of Lemma 3.4.4, namely that $c\theta \Rightarrow^k c_k$ in P implies $c\theta\phi \Rightarrow^k c_k\phi$ in P, and the fact that the only surrounding function calls in e are constructor functions.

Induction step: Assume $\rho_\sigma(t) \Rightarrow^j \rho_\sigma(t'_j)$ in P_A, $t'_j \Rightarrow^* t_n$ in P and $t \Rightarrow^* t_n$ in P. Then, it remains to prove if $\rho_\sigma(t) \Rightarrow^{j+1} \rho_\sigma(t'_{j+1})$ then $t'_{j+1} \Rightarrow^* t_n$.

By the argument above, $\rho_\sigma(t'_j) \Rightarrow^1 \rho_\sigma(t'_{j+1})$ in P_A and $t'_{j+1} \Rightarrow^* t_i$ in P implies $t'_j \Rightarrow^* t_i$ in P (since t'_j is A-passive). By the induction hypothesis, $t'_j \Rightarrow^* t_n$ in P. Since computations are deterministic, $t_n = t_i$. Therefore, $t'_{j+1} \Rightarrow^* t_n$ in P. □

The next theorem ensures that a term is in normal form with respect to the original program if its translation is in normal form with respect to the residual program.

Theorem 3.4.6 Let P be a program, A be set of terms, and t be a term. Let P_A be the partial evaluation of P wrt A such that $P_A \cup \{t\}$ is A-passive. Then, if $\rho_\sigma(t)$ cannot be rewritten in P_A, then t cannot be rewritten in P.

Proof
Proof by contradiction. Assume $\rho_\sigma(t) \not\Rightarrow$ in P_A and $t \Rightarrow t_1$ in P. Then, for some context e and redex r, $t = e[r]$ such that $r = h\psi$ for some schema statement $h \Rightarrow b$ in P and substitution ψ.

Since $\rho_\sigma(t)$ is not rewritable in P_A, there is no subterm of $\rho_\sigma(t)$ that is an instance of a schema statement head in P_A.

Since t is A-passive, there is a subterm $\rho_\sigma(s)$ of $\rho_\sigma(t)$ such that $r = s$ and $s = c\theta$ for some $c \in A$ and substitution θ. Assume without loss of generality that s is renamed in $\rho_\sigma(t)$ using $c \in A$.

Case 1: c has a non-trivial computation in P. Then there is a statement with head $\sigma(c)$ in P_A, and $\rho_\sigma(s) = \rho_\sigma(c)\rho_\sigma(\theta) = \sigma(c)\theta$. Contradiction.

Case 2: c has a trivial computation in P. Then there is a residual definition of c in P using restart terms $c\varphi_i$. Since s is an instance of c, it must either be equal to or be an instance of some restart term $c\varphi_i$, as the restart are the minimal instantiations of c with non-trivial computations and s has a non-trivial computation in P. Therefore, $\rho_\sigma(s)$ is an instance of the schema statement head $\rho_\sigma(c\varphi_i)$. Contradiction. □

Finally, the soundness theorem for A-passive terms is presented below.

Theorem 3.4.7 (Basic Soundness)
Let P be a program, A a set of terms, and t be a term. Let P_A be the partial evaluation of P wrt A such that $P_A \cup \{t\}$ is A-passive. Then, t has a finite computation in P with result t_n if $\rho_\sigma(t)$ has a finite computation in P with result $\rho_\sigma(t_n)$.

Proof
A direct consequence of Theorems 3.4.5 and 3.4.6. If $\rho_\sigma(t_n)$ cannot be rewritten in P_A, then t_n cannot be rewritten in P, and the result follows. □

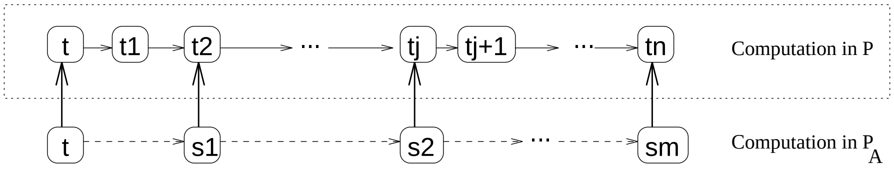


Figure 3.5: Proving the completeness of the partial evaluation procedure by constructing an injective mapping from terms of the computation in P_A to terms of the computation in P.

Completeness

In addition to contributing to the proof of soundness, Theorem 3.4.6 provides part of the proof of completeness as well. That is, if a computation terminates in the residual program, it is guaranteed to terminate in the original program. Therefore, in order to prove the completeness of the procedure, it remains to show that that a finite computation in the original program P guarantees a finite computation in the residual program P_A. In order to prove this, a one-to-one mapping from the terms in the computation of P_A to the terms in the computation of P is constructed, as illustrated in Figure 3.5. Since the mapping is injective, the finiteness of the computation in P_A is ensured.

Theorem 3.4.8 Let P be a program, A a set of terms, and t a term. Let P_A be the partial evaluation of P wrt A such that $P \cup \{t\}$ is A-passive. Then, if t has a finite computation in P, $\rho_\sigma(t)$ has a finite computation in P_A.

Proof

Let $t \Rightarrow^* t_n$ in P. Show that there exists an injective mapping from the terms of the computation of $\rho_\sigma(t)$ in P_A to the terms of the computation of t in P. That is, there exists a mapping ϱ such that for all natural numbers n, if $s_n \Rightarrow s_{n+1}$ in P_A, then $\varrho(s_n) = t_j$ and $\varrho(s_{n+1}) = t_{j+m}$ for some j and m such that $m > 0$ and $t_j \Rightarrow^m t_{j+m}$ in P.

Define the mapping $\varrho(s) = \rho_\sigma^{-1}(s)$. Show that this mapping satisfies the condition stated above by induction on the length of the computation in P_A.

Case $\rho_\sigma(t) \Rightarrow^1 s_1$: Assume $\rho_\sigma(t)$ rewrites in one computation step to s_1, using schema statement $h^A \Rightarrow b^A$ in P_A. Then, $\rho_\sigma(t) = e^\rho[h^A\phi] \Rightarrow e^\rho[b^A\phi] = s_1$, by definition of a computation step.

In order to prove the mapping ϱ satisfies the condition above, show that t rewrites to $t_{1+m} = \rho_\sigma^{-1}(s_1)$ via a non-trivial computation in P. The argument follows the approach in the proof of Theorem 3.4.5.

The schema statement $h^A \Rightarrow b^A = \rho_\sigma(c\theta) \Rightarrow \rho_\sigma(c_k)$ for some $c \in A$ and substitution θ such that $c\theta$ is a restart term for c and $c\theta$ has a non-trivial partial computation in P with result c_k. Therefore, $\rho_\sigma(t) = e^\rho[\rho_\sigma(c\theta)\phi] \Rightarrow e^\rho[\rho_\sigma(c_k)\phi] = s_1$.

Since t is A-passive, $\rho_\sigma^{-1}(\rho_\sigma(c\theta)\phi) = c\theta\phi$ (passive substitutions are unchanged by the translation function). Therefore, $t = e[c\theta\phi]$ and $\rho_\sigma^{-1}(s_1) = e[c_k\phi]$ for some context e. The demanded argument is not changed by renaming, thus $c\theta\phi$ is the redex of t, or contains the redex of t. It remains to show that $e[c\theta\phi] \Rightarrow^k e[c_k\phi]$. This follows from Lemma 3.4.4 and the fact that the only surrounding function calls in e are constructor functions. Therefore, the property is satisfied with $m = k - 1$.

Induction Step: Assume for all i, $1 \leq i \leq k$, if $s_{i-1} \Rightarrow s_i$ in P_A, then $\varrho(s_{i-1}) = t_j$ and $\varrho(s_i) = t_{j+m}$ for some j and m such that $m > 0$ and $t_j \Rightarrow^n t_{j+m}$ in P. Show this holds for all i, $1 \leq i \leq k+1$.

Straightforward from the base case above. □

Therefore, it has been shown that passive instances of terms of A have computations in the residual program in which every term can be related to a term in the computation in the original program. Relaxing the passive instance restriction results in the loss of this direct relationship between terms of the computations; the correctness proofs for this case are presented in the next section.

3.4.2 Strong Correctness

The correctness of the transformation was proved for a restricted class of terms and programs in the previous section. With this limitation, the computations in the residual programs could be related to the computations in the original programs.

In this section, the aim is to relax the A-passive restriction of the terms and programs. However, this requires the refinement of the proofs of soundness and completeness. The following example illustrates the loss of the direct relationship between the computations in P and P_A, on which several of the proofs of the previous chapter were based.

Example 3.4.9 Consider the list concatenation program P from Example 3.2.5. Let A' be the set of terms:

{`Concat(x,y), Concat(Concat([f,s|t],y),z), Concat(Concat(x,y),z)`}.

The residual program $P_{A'}$ resulting from the specialisation of P wrt A' is:

```
C([],y)        =>  y.
```

```
C([h|t],y)     =>  [h|C(t,y)].
CC([],y,z)     =>  C(y,z).
CC([h|t],y,z)  =>  [h|C(t,y,z)].
CC2(f,s,t,y,z) =>  [f,s|CC(t,y,z)].
```

where the renaming operator σ is defined as in 3.2.12 with the addition of the mapping:

`Concat(Concat([f,s|t],y),z)` $\mapsto$ `CC2(f,s,t,y,z)`.

Consider the computation of the term

`Concat(Concat(Concat(Concat([3,4],y),z),w),x)`

in the original program P and the equivalent term `CC(CC2([3,4],y,z),w,x)` in $P_{A'}$ (Figure 3.6). In this case, the order of evaluation is altered by the transformation. However, the change in evaluation order has not affected the result of the computation in $P_{A'}$. That is, although the sequence of the statement identifiers are the same in the computation in P and in the computation in P_A, considering the base computations, the redexes in the computations are chosen in a different order.

In this case, the `Concat` function has a property that ensures this change of evaluation order does not affect the result of the computations. This property, called *compositionality*, will be defined later in this section.

The previous example showed how the evaluation order may be altered by allowing run-time instances of terms in A to contain instances of terms in A. In this case, the resultant used in the generation of the statement in the definition of the `CC2` function produces *two* outermost constructor functions. This results in the nested term evaluating the innermost argument more than necessary for the immediate outermost function call, which only requires one outermost constructor function. Therefore, the innermost standard subterm is simplified more than is required by the computation at that time.

Unfortunately, the generality of the rewriting-based functional logic languages means that the construction of resultants using the unfolding mechanism of the language requires an additional condition to ensure the correctness of the specialised programs. The problem arises from allowing the term arguments of g-functions to be arbitrary non-variable terms.

In rewriting-based functional logic languages, a g-function is reduced as soon as the term matches one of the heads of the statements in the definition of the function. If the term arguments of g are patterns, there is no problem with computing resultants using arbitrary partial computations in P. When an outermost constructor is produced, it cannot be destroyed again during the partial

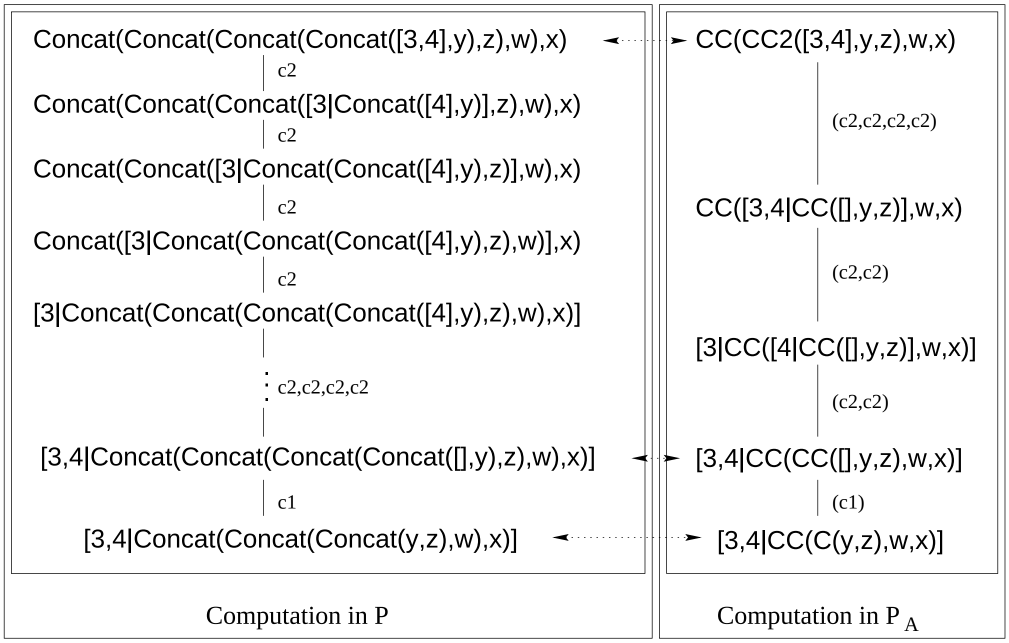

Concat(Concat(Concat(Concat([3,4],y),z),w),x)
c2
Concat(Concat(Concat([3|Concat([4],y)],z),w),x)
c2
Concat(Concat([3|Concat(Concat([4],y),z)],w),x)
c2
Concat([3|Concat(Concat(Concat([4],y),z),w)],x)
c2
[3|Concat(Concat(Concat(Concat([4],y),z),w),x)]
c2,c2,c2,c2
[3,4|Concat(Concat(Concat(Concat([],y),z),w),x)]
c1
[3,4|Concat(Concat(Concat(y,z),w),x)]
CC(CC2([3,4],y,z),w,x)
(c2,c2,c2,c2)
CC([3,4|CC([],y,z)],w,x)
(c2,c2)
[3|CC([4|CC([],y,z)],w,x)]
(c2,c2)
[3,4|CC(CC([],y,z),w,x)]
(c1)
[3,4|CC(C(y,z),w,x)]
Computation in P
Computation in P $_A$

Figure 3.6: Comparing the expanded computation in the residual program and the computation in the original program. The extended computation of the term in the residual program is not equivalent to the computation in the original program.

computation. However, if the term arguments are arbitrary non-variable terms, the correctness of the transformed program is not guaranteed. It is possible that the necessary syntactic term may be “lost” in one of the resultants. Consider the following example demonstrating the problem.

Example 3.4.10 Consider the following *E* program, P:

```
G([],y)      => y.
G([x|xs],y) => 1 + G(xs,y).
G(F(x),y)   => 10 + G(x,y).
H(x)         => F(x).
F(x)         => x.
```

This program is pattern and instance non-overlapping. Specialising this program with respect to the set of terms $A = \{\texttt{G(x,y), H(x)}\}$ may result in the residual program P_A shown below (assuming the identity renaming function in this case).

```
G([],y)      => y.
G([x|xs],y) => 1 + G(xs,y).
G(F(x),y)   => 10 + G(x,y).
H(x)         => x.
```

Computing the term `G(H([]),2)` in the original and residual programs results in different terms, in this case, `12` and `3`.

Note that this case is not covered by the basic correctness result of Section 3.4.1, since the term `G(H([]),2)` is not A-passive. On the other hand, specialising P with respect to the set $A' = \{\texttt{G(H(x),y)}\}$ results in the following residual program, P'_A:

```
G1(x,y)       => 10 + G2(x,y).
G2([],y)      => y.
G2([x|xs],y) => 1 + G2(xs,y).
G2(F(x),y)   => 10 + G2(x,y).
```

assuming the renaming function: $\{\texttt{G(H(x),y)} \mapsto \texttt{G1(x,y)}, \texttt{G(x,y)} \mapsto \texttt{G2(x,y)}\}$. The term `G(H([]),2)` is A' passive, and both the original program and this residual program simplify the term to `12`.

The above program is deterministic, and therefore, confluent. However, allowing the nesting of terms of A results in the "loss" of intermediate terms of the computation. For example, in the residual program P_A above (Example 3.4.10), the residual definition of `H` is optimal for an instance of the term `H(x)`. On the other hand, since the intermediate term `F(x)` is optimised away in the residual program, the program P_A is no longer equipped to correctly reduce a call of the form `G(H(x),y)`.

This type of problem arises when run-time calls containing nested terms of A are allowed *and* some functions in the definitions of functions occurring in A have standard terms (§ 2.3.4) for term arguments in their definitions. Then, it is possible to optimise away intermediate terms (as above) which are necessary to ensure a correct reduction in the residual program. This is a problem related to that of the partial evaluation of concurrent programs; intermediate states may be removed by the partial evaluation which are required for the correct operational behaviour of the residual program [MG97].

This problem is addressed by supplementing the *closedness* condition, defined below, with an extra condition of *compositionality*.

Definition 3.4.11 (compositional context, function)
Let P be a program, and $e[]$ be a context. Then $e[]$ is a compositional context if for any terms t_1, t_n such that $t_1 \Rightarrow^n t_n$ in P, $e[t_1] \Rightarrow^* t'$ in P iff $e[t_n] \Rightarrow^* t'$ in P. A compositional function is a g-function for which the context $g([], t_1, \ldots, t_n)$ is compositional.

A program P is *compositional* if all functions defined in P are compositional. By definition, f-functions are compositional, as they have no demanded argument. A g-function is automatically compositional if every term argument in its definition is a pattern. For example, the `Concat` function is compositional (Example 3.4.9), as is the context `Concat(Concat([],y),z)`. On the other hand, the program P of Example 3.4.10 is not compositional.

Without compositionality of functions, the only terms for which the residual program is guaranteed to be correct are A-passive terms, as shown in Section 3.4.1. By requiring the compositionality of functions of P, run-time terms are permitted to be A-*closed*.

Definition 3.4.12 (A-closed)
Let A be a finite set of terms, t be a term. Let the set A_R contain the restart terms for each s_i in A, $C = \mathcal{C}(t)$ be the set of closed terms for t. Then, t is A-closed if for every term $s \in C$, there exists a term s' in A_R such that $s = s'\theta$ for some substitution θ and the terms of θ are A-closed. For a set of terms B, B is A-closed if every term t in B is A-closed.

For a residual program $P_A = \{\rho_\sigma(h_1) \Rightarrow \rho_\sigma(b_1), \ldots, \rho_\sigma(h_n) \Rightarrow \rho_\sigma(b_n)\}$, P_A is *A*-closed if the set $\{b_1, \ldots, b_n\}$, is *A*-closed.

The definition of closedness above requires the property to be proved for every term in the set of the closed terms for the term t. This ensures that all the function calls occurring in the term are defined in the program P. The $\mathcal{C}$ function is dependent on the particular syntax of the target language. As an example, consider the function $\mathcal{C}(t)$ for the terms of the *E* language.

Definition 3.4.13 ($\mathcal{C}(t)$)
Given a term t, the set of closed terms for t, $\mathcal{C}(t)$, is defined as follows.

$$\begin{array}{lcll}
\mathcal{C}(t) & = & \{\} & \text{if } t = x \\
\mathcal{C}(t) & = & \mathcal{C}(t_1) \cup \ldots \cup \mathcal{C}(t_n) & \text{if } t = c(t_1, \ldots, t_n) \\
\mathcal{C}(t) & = & \mathcal{C}(t_1) \cup \ldots \cup \mathcal{C}(t_n) & \text{if } t = p(t_1, \ldots, t_n) \\
\mathcal{C}(t) & = & \{t\} & \text{if } t = f(t_1, \ldots, t_n) \\
\mathcal{C}(t) & = & \{t\} & \text{if } t = g(t_0, t_1, \ldots, t_n) \\
\mathcal{C}(t) & = & \mathcal{C}(t') & \text{if } t = \texttt{LAMBDA}[x](t') \\
\mathcal{C}(t) & = & \mathcal{C}(t_1) \cup \mathcal{C}(t_2) & \text{if } t = t_1\ t_2 \\
\mathcal{C}(t) & = & \mathcal{C}(t_1) \cup \mathcal{C}(t_2) \cup \mathcal{C}(t_3) & \text{if } t = \texttt{ITE}(t_1, t_2, t_3) \\
\mathcal{C}(t) & = & \mathcal{C}(t') & \text{if } t = \{x \mid t'\}
\end{array}$$

Example 3.4.14 Given the set A of Example 3.4.1, the following terms are A-closed:

```
Concat(Concat([1,2],[3,4]),[5,6])
[Concat(Concat([3,4],y),z) | w]
```

Example 3.4.15 Consider the set A of Example 3.4.9. The term

```
Concat(Concat(Concat(Concat([3,4],y),z),w),x)
```

is A-closed. Furthermore, the residual program P_A is A-closed.

The A-passive condition is a special case of A-closedness, in which the terms of the substitution are trivially A-closed because they are patterns.

Lemma 3.4.16 Let A be a set of terms, t be a term. If t is A-passive, then t is A-closed.

Proof Straightforward from the definition of A-passive and A-closed. □

Strong Soundness

The proof of soundness for this stronger version follows the approach of the proof in Section 3.4.1. On the other hand, the nested closed terms of t make the following proofs more complex than those of the previous section.

Theorem 3.4.17 Let P be a compositional program, A a set of terms, and t a term. Let P_A be the partial evaluation of P wrt A such that $P_A \cup \{t\}$. Then, if $\rho_\sigma(t)$ has a partial computation in P_A with result $\rho_\sigma(t')$ and $t' \Rightarrow^* t_n$ in P, then $t \Rightarrow^* t_n$ in P.

Proof
Let $\rho_\sigma(t) \Rightarrow^m \rho_\sigma(t')$. That is, $\rho_\sigma(t)$ has a finite computation of m-steps resulting in $\rho_\sigma(t')$in P_A. Show if $t' \Rightarrow t_n$ in P, then $t \Rightarrow^* t_n$ in P by induction on the length of computation in P_A, m.

Base case $m = 1$: For the base case, show that if $\rho_\sigma(t) \Rightarrow^1 \rho_\sigma(t')$ in P_A and $t' \Rightarrow^* t_n$ in P, then $t \Rightarrow^* t_n$ in P.

As in Theorem 3.4.5, the expression $\rho_\sigma(t) = e^\rho[\rho_\sigma(c\theta)\phi] \Rightarrow e^\rho[\rho_\sigma(c_k)\phi] = \rho_\sigma(t')$ is obtained using the schema statement $h^A \Rightarrow b^A$ in P_A where $h^A \Rightarrow b^A = \rho_\sigma(c\theta) \Rightarrow \rho_\sigma(c_k)$ for some c in A.

Since t is A-closed, $\rho_\sigma^{-1}(\rho_\sigma(c\theta)\phi) = c\theta\phi'$, where $\phi' = \rho_\sigma^{-1}(\phi)$. Therefore, $t = e[c\theta\phi']$ and $t' = e[c_k\phi']$ for some context e. Since the demanded argument is not changed by renaming, either $c\theta\phi'$ is the redex of t or contains the redex of t.

The compositionality of P implies the functions in e are compositional. By Lemma 3.4.4, $c\theta \Rightarrow^k c_k$ in P implies $c\theta\phi' \Rightarrow^k c_k\phi'$ in P. By definition of compositionality, $e[c\theta\phi'] \Rightarrow^* t_n$ iff $e[c_k\phi'] \Rightarrow^* t_n$.

Induction step: As in Theorem 3.4.5. □

The following theorem follows directly from Theorem 3.4.6.

Theorem 3.4.18 Let P be a program, A be set of terms, and t be a term. Let P_A be the partial evaluation of P wrt A such that $P_A \cup \{t\}$ is A-closed. Then, if $\rho_\sigma(t)$ cannot be rewritten in P_A, then t cannot be rewritten in P.

Proof As in Theorem 3.4.6, replacing A-passive by A-closed. □

Theorem 3.4.19 (Soundness)
Let P be a compositional program, A a set of terms, and t be a term. Let P_A be the partial evaluation

of P wrt A such that $P_A \cup \{t\}$ is A-closed. Then, t has a finite computation in P with result t_n if $\rho_\sigma(t)$ has a finite computation in P with result $\rho_\sigma(t_n)$.

Proof A direct consequence of Theorems 3.4.17 and 3.4.18. If $\rho_\sigma(t_n)$ cannot be rewritten in P_A, then t_n cannot be rewritten in P, and the result follows. □

Strong Completeness

Although the approach to the proof of soundness above was similar to that of Section 3.4.1, for the proof of completeness a simple injective mapping from elements of the computation in P_A to terms in the computation in P cannot be constructed because such a mapping is not guaranteed to exist. In fact, the following example presents a counter-example to the strong completeness of the procedure for arbitrary A-closed terms and programs.

Example 3.4.20 Consider the following *E* program:

```
G([],y)      => y.
G([x|xs],y) => 1 + G(xs,y).
G(F(x),y)   => 1 + G(x,y).
H(x)        => F(x).
F(x)        => H(x).
```

This program has been changed from the program in Example 3.4.10 to be compositional. Specialising this program with respect to the set $A = \{$ `G(x,y), H(x)`$\}$ may result in the following residual program P_A (again assuming the identity renaming function).

```
G([],y)      => y.
G([x|xs],y) => 1 + G(xs,y).
G(F(x),y)   => 1 + G(x,y).
H(x)        => H(x).
```

Rewriting the A-closed term `G(H([]),2)` in the original program results in the term `3`. The computation of this term in the residual program is non-terminating. Again, the intermediate term `F(x)` has been optimised away by the partial evaluation, causing the residual program to lose some of the functionality of the original program for A-closed terms.

Therefore, a further restriction on terms and programs is necessary for the strong completeness theorem. Programs are required to be *constructor-based*, that is, having only term arguments with outermost constructors. Constructors cannot be destroyed during the computation of resultants, unlike arbitrary terms. This prevents a situation as in Example 3.4.20.

Definition 3.4.21 (constructor-based program P)
A program P is a *constructor-based program* if for all functions h defined in P, the term arguments in the definition of h in P are pattern terms.

A constructor-based program is guaranteed to be compositional. The completeness of the procedure is proved by defining an injective mapping from terms in the computation in P_A to terms in the computation in P. Since the latter computation is finite, the computation in the residual program is guaranteed to be finite. However, as shown in Figure 3.6, the injective mapping ρ_σ^{-1} used in the proof of Theorem 3.4.8 cannot be used in the proof of completeness in this case. Instead, the mapping is based on *term identifiers*.

Definition 3.4.22 (term identifier)
Let P be a program, and t a term with redex r. Then, the *identifier* of t, $\xi(t)$, is the sequence constructed as follows: let $t = s_0, s_1, \ldots, s_{k-1}, s_k = r$ be the sequence of all outermost functions of subterms of t containing r, ordered by depth. Then the identifier of t is the sequence $H_0 \ldots, H_k$ where H_i is the outermost function of s_i.

Term identifiers simply record the outermost functions surrounding the redex in the term. For example, the term identifier of `Concat(Concat([],y),z)` is (`Concat`, `Concat`). In the case of ambiguity, the argument indices can be added to the outermost functions.

The *skeleton* of a redex r is an abstraction of the subterm including only the demanded arguments of r.

Definition 3.4.23 (skeleton)
Given a program P and redex r, *skel*(r), the skeleton of r, is defined as follows. Let $r = H(t_1, \ldots, t_{i-1}, t_i, \ldots, t_n)$, where the $1, \ldots, i-1$ arguments are required by the computation (i.e. the term arguments for H). Let $d_1, \ldots, d_{i-1}$ be the maximum depth of each term argument in the definition of H in P. Then, *skel*$(r) = H(t_1^{d_1}, \ldots, t_{i-1}^{d_{i-1}}, z_i, \ldots, z_n)$ where each $z_j \notin FV(r)$ and for each $t_k^{d_k}$ is the term t_k abstracted to depth d_k.

For example, the skeleton of the redex $r =$`Concat([1,2],Concat([3],y))` is `Concat([1|u],v)`. Now, the correctness of the transformation is established.

Theorem 3.4.24 Let P be a constructor-based program, A a set of terms, and t a term. Let P_A be the partial evaluation of P wrt A such that $P \cup \{t\}$ is A-closed. Then, if t has a finite computation in P, $\rho_\sigma(t)$ has a finite computation in P.

Proof

Let $t \Rightarrow^* t_n$ in P. Show that there exists an injective mapping from the terms of the computation of $\rho_\sigma(t)$ in P_A to the terms of the computation of t in P. That is, there exists a mapping ϱ such that for all natural numbers n, if $s_n \Rightarrow s_{n+1}$ in P_A, then $\varrho(s_n) = t_j$ and $\varrho(s_{n+1}) = t_{j+m}$ for some j and m such that $m > 0$ and $t_j \Rightarrow^m t_{j+m}$ in P.

Build a mapping as follows. Given a term s_i occurring in the computation of t in P_A, with $s_i = e^\rho[r^\rho]$ and redex r, then $\varrho(s_i) = t_i$ where $\xi(t_i) = \xi(\rho_\sigma^{-1}(s_i))$, $t_i = e[r]$ and $\mathit{skel}(\rho_\sigma^{-1}(r^\rho)) = \mathit{skel}(r)$.

Base case $\rho_\sigma(t) \Rightarrow^1 s_1$: Assume $\rho_\sigma(t)$ rewrites in one computation step to s_1, using schema statement $h^A \Rightarrow b^A$ in P_A. Then, $\rho_\sigma(t) = e^\rho[r^\rho] = e^\rho[h^A\phi] \Rightarrow e^\rho[b^A\phi] = s_1$, by definition of a computation step.

In order to prove the mapping ϱ satisfies the condition above, show that t rewrites to t_{1+m} via a non-trivial computation in P such that $\xi(t_{1+m}) = \xi(\rho_\sigma^{-1}(s_1))$, $t_{1+m} = e[r]$ and $\mathit{skel}(\rho_\sigma^{-1}(r^\rho)) = \mathit{skel}(r)$.

The schema statement $h^A \Rightarrow b^A = \rho_\sigma(c\theta) \Rightarrow \rho_\sigma(c_k)$ for some $c \in A$ and substitution θ such that $c\theta$ is a restart term for c and $c\theta$ has a non-trivial partial computation in P with result c_k. Therefore, $\rho_\sigma(t) = e^\rho[\rho_\sigma(c\theta)\phi] \Rightarrow e^\rho[\rho_\sigma(c_k)\phi] = s_1$.

Since t is A-closed, $\rho_\sigma^{-1}(\rho_\sigma(c\theta)\phi) = c\theta\phi'$, where $\phi' = \rho_\sigma^{-1}(\phi)$. Therefore, $t = e[c\theta\phi']$ and $\rho_\sigma^{-1}(s_1) = e[c_k\phi']$ for some context e. Since the demanded argument is not changed by renaming, either $c\theta\phi'$ is the redex of t or contains the redex of t.

Assume that $c_k\phi'$ has an outermost constructor function. Now, since the program is required to be constructor-based, a redex in the context e cannot be evaluated until the term $c\theta\phi'$ is rewritten to a term having an outermost constructor, call this term c'. Since constructors cannot be destroyed, the terms c' and $c_k\phi'$ must have the same outermost constructor function.

Let $t_{1+m} = e[c']$. Clearly, $\xi(t_i) = \xi(\rho_\sigma^{-1}(s_i))$, since the demanded argument of a term is not changed by the renaming. It remains to show that if r is the redex of $e[c']$, then $\mathit{skel}(\rho_\sigma^{-1}(r^\rho)) = \mathit{skel}(r)$. Since constructor-based programs have term arguments with maximum depth 1, this follows directly, since c' and $c_k\phi'$ have the same outermost constructor function.

Induction Step: Assume for all i, $1 \leq i \leq k$, if $s_{i-1} \Rightarrow s_i$ in P_A, then $\varrho(s_{i-1}) = t_j$ and $\varrho(s_i) = t_{j+m}$ for some j and m such that $m > 0$ and $t_j \Rightarrow^n t_{j+m}$ in P. Show this holds for all i, $1 \leq i \leq k+1$.

Straightforward from the base case above. □

In summary, overall in this section, the following correctness results related to programs containing pairwise pattern and instance non-overlapping statements were established:

- For A-passive programs and terms, the residual program P_A is sound and complete with respect to the original program P.
- For constructor-based, A-closed programs and A-closed terms, the residual program P_A is sound and complete with respect to the original program P.
- For compositional, A-closed programs and A-closed terms, the residual program P_A is sound with respect to the original program P.

3.5 Correctness Revisited

In this section, the correctness of the transformation is shown with respect to confluent programs. Programs are no longer restricted to contain statements having pairwise pattern and instance non-overlapping heads. This implies a term may be reduced using one of several rewrite rules in the program. Confluence ensures two different computations will be "brought together" after finitely many computation steps [Pla93]. A confluent program P is defined as follows.

Definition 3.5.1 (confluence)
Let P be a program. For any terms t, t_j, t_k, if $t \Rightarrow^j t_j$ and $t \Rightarrow^k t_k$ in P, then there exists a term t_n such that $t_j \Rightarrow^i t_n$ and $t_k \Rightarrow^{i'} t_n$.

The motivation to prove the lemma for confluent programs was inspired from the specialisation of Boolean expressions. For example, in the Escher functional logic language [Llo95], the schema statements to rewrite terms with an outermost & operator include the following:

```
x & (y \/ z) => (x & y) \/ (x & z)

False & x => False
```

As noted in Section 3.4.1, Lemma 3.4.4 does not hold for confluent program. Given a term t that reduces to a term t' in P, it is not possible to prove, for any substitution θ, that $t\theta$ will reduce to $t'\theta$. The term $t\theta$ may match a different rewrite rule in the program P, and therefore, bypass the term $t'\theta$ altogether. For example, the term `x & (y \/ z)` may be evaluated in Escher using the first schema statement above, but if the substitution $\{x := \texttt{False}\}$ is applied to the term, either rewrite rule could be used to simplify the term.

The following lemma is a weaker version of Lemma 3.4.4 that holds for confluent programs.

Lemma 3.5.2 Let P be a confluent program and t, t' be terms such that $t \Rightarrow^m t'$ in P. Then, given a substitution θ, there exists a term t_n such that $t\theta \Rightarrow^n t_n$ and $t'\theta \Rightarrow^{n'} t_n$ in P.

Proof
Assume $t \Rightarrow^m t'$ in P. In order to prove the above lemma, show (1) that $t\theta \Rightarrow^m t'\theta$ is a possible computation in P (it may not be the computation used by the interpreter to reduce $t\theta$). Then, by Definition 3.5.1, there exists a term t_n such that $t\theta \Rightarrow^n t_n$ and $t'\theta \Rightarrow^{n'} t_n$ in P.

Proof of (1) by induction on the length of computation n.
$n = 1$: Assume $t \Rightarrow t_1$. Without loss of generality, assume that t has one redex r, such that $t = e[r]$. By definition of a computation step, $r = h_i\varphi$ for some statement $h_i \Rightarrow b_i$ in P (where P is either simply the user-defined program or the user-defined program composed with the modules defining the primitive functions), and $t_1 = e[b_i\varphi]$.

Assume there is no possible computation of $t\theta$ to $t'\theta$ in P. By definition $t\theta = e[r]\theta$. By definition of matching, if r matches h_i, then $r\theta$ must also match h_i. Furthermore, the redex is unchanged in all instances of t. Therefore, $t\theta = (e[r])\theta = e\theta[r\theta] = e\theta[h_i\varphi\theta] \Rightarrow e\theta[b_i\varphi\theta] = (e[b_i\varphi])\theta = t_1\theta$.

Induction step: Straightforward from case $n = 1$. □

Soundness

The approach to the proof of soundness is the same as for the proof for pattern and instance non-overlapping programs (§ 3.4.1). The proof will proceed in the following way. It is assumed that the translated term t reduces to the renamed term t_k in k computation steps in P_A and the term t_k rewrites to the normal form t_n in P. By showing that there exists a term t' such that t and t_k reduce to t' in P,

Theorem 3.5.3 Let P be a confluent, compositional program, A a set of terms, and t a term. Let P_A be the partial evaluation of P wrt A such that $P_A \cup \{t\}$ is A-closed. Then, if $\rho_\sigma(t)$ has a partial computation in P_A with result $\rho_\sigma(t')$ and $t' \Rightarrow^* t_n$ in P, then $t \Rightarrow^* t_n$ in P.

Proof
Let $\rho_\sigma(t) \Rightarrow^m \rho_\sigma(t')$. That is, $\rho_\sigma(t)$ has a finite computation of m-steps resulting in $\rho_\sigma(t')$ in P_A. Show if $t' \Rightarrow t_n$ in P, then $t \Rightarrow^* t_n$ in P by induction on the length of computation in P_A, m.

Case $m = 1$: For the base case, show that if $\rho_\sigma(t) \Rightarrow^1 \rho_\sigma(t')$ in P_A and $t' \Rightarrow^* t_n$ in P, then $t \Rightarrow^* t_n$ in P.

As in Theorem 3.4.5, the expression $\rho_\sigma(t) = e^\rho[\rho_\sigma(c\theta)\phi] \Rightarrow e^\rho[\rho_\sigma(c_k)\phi] = \rho_\sigma(t')$ is obtained using the schema statement $h^A \Rightarrow b^A$ in P_A where $h^A \Rightarrow b^A = \rho_\sigma(c\theta) \Rightarrow \rho_\sigma(c_k)$ for some c in A.

Since t is A-closed, $\rho_\sigma^{-1}(\rho_\sigma(c\theta)\phi) = c\theta\phi'$, where $\phi' = \rho_\sigma^{-1}(\phi)$. Therefore, $t = e[c\theta\phi]$ and $t' = e[c_k\phi]$ for some context e. Since the demanded argument is not changed by renaming, either $c\theta\phi$ is the redex of t or contains the redex of t.

By Lemma 3.5.2, $c\theta \Rightarrow^k c_k$ in P implies there exists a term c' such that $c\theta\phi' \Rightarrow^{k'} c'$ and $c_k\phi' \Rightarrow^{k''} c'$. The compositionality of P implies the functions in e are compositional. By definition of compositionality, $e[c\theta\phi'] \Rightarrow^* t_n$ iff $e[c'] \Rightarrow^* t_n$ iff $e[c_k\phi'] \Rightarrow^* t_n$.

Induction step: As in Theorem 3.4.5. □

The proof of Theorem 3.4.18 does not require the pattern and instance non-overlapping property of programs. Therefore, the soundness of the procedure can be established immediately.

Theorem 3.5.4 (Soundness) Let P be a confluent, compositional program, A a set of terms, and t be a term. Let P_A be the partial evaluation of P wrt A such that $P_A \cup \{t\}$ is A-closed. Then, t has a finite computation in P with result t_n if $\rho_\sigma(t)$ has a finite computation in P with result $\rho_\sigma(t_n)$.

Proof A direct consequence of Theorems 3.5.3 and 3.4.18. If $\rho_\sigma(t_n)$ cannot be rewritten in P_A, then t_n cannot be rewritten in P, and the result follows. □

Completeness

As in Theorem 3.4.24, the completeness of the procedure is limited to constructor-based programs (Def. 3.4.21). The confluence of the program does not prevent intermediate terms being “lost” in the resultants as in Example 3.4.20. The proof of the completeness is almost identical to that of Theorem 3.4.24; therefore, the theorem is presented without proof below.

Theorem 3.5.5 Let P be a constructor-based, confluent program, A a set of terms, and t a term. Let P_A be the partial evaluation of P wrt A such that $P \cup \{t\}$ is A-closed. Then, if t has a finite computation in P, $\rho_\sigma(t)$ has a finite computation in P.

3.6 Discussion

Since functional logic languages incorporate features of both of the major declarative programming paradigms, it seems natural that the framework for specialising these programs shares its foundation with existing transformation procedures for functional and logic programs. In this section, the partial evaluation procedure for functional logic programs presented in this work is compared to those techniques for other declarative programs.

The framework of the partial evaluation procedure for functional logic programs is based on the partial deduction foundations established by Komorowski [Kom81] and Lloyd and Shepherdson [LS91]. The definition of a resultant is slightly different in this formalisation, since resultants are not defined in terms of a non-failing derivation, but instead, just with respect to a term t. This means this definition of resultant is more general than that in partial deduction. A resultant for a term t is not unique, since it depends on the length of the partial computation of t in P.

In both partial deduction [LS91] and the specialisation of narrowing-based functional logic programs [AFJV97], the generation of restart terms is unnecessary, due to the unification-based computation mechanism of the interpreter. Restart terms are implicitly computed in positive supercompilation [SGJ96] by propagating the patterns to the terms by unification during the specialisation of an expression with an outermost g-function. Wadler's deforestation requires the arguments in the left-hand sides of equations to be variables, that is, every function in the program is an f-function [Wad90]. Restart terms do not have to be computed for f-functions, since they will always reduce by at least one computation step.

The renaming operators are based on those defined for conjunctive partial deduction [LSdW96, Leu97a]; similar renaming techniques are required during the partial evaluation of narrowing-based functional logic programs [AFJV97]. In each of these techniques, it is noted that without an ordering operator as defined in this chapter, the renaming is non-deterministic. Related non-determinacy exists in partial deduction: recall that the correctness of the procedure requires the independence of the set A (§ 2.1.4). An atom in A may have a common instance with more than one element of A. Without an ordering on the set of the terms of A, the selection of the element of A to generalise in order to maintain independence is non-deterministic.

Several conditions were identified in the theorems of soundness and completeness to restrict the domains of programs and terms (Sections 3.4 and 3.5). In Section 3.4.1, the correctness of the procedure was established for A-*passive* programs and terms. That is, the terms of the program P and the term t must be instances of restart terms of the elements of A, where the instances contain only pattern terms (§ 2.3.4). With this restriction, it was possible to show the equivalence between computations in the original and residual programs.

Section 3.4.2 considered the correctness of the transformation in terms of a *compositional*, A-*closed* program P and A-*closed* term t. The instances of the restart terms of elements of A are permitted to be nested in the run-time call to the residual program P_A. However, since term arguments can be arbitrary non-variable terms of *E*, it is possible that the residual statements would optimise away terms that are required for the correct computation in the residual program. Therefore, the compositionality of the program P was required to ensure the correctness of the transformation. Furthermore, Example 3.4.10 showed that the completeness of the transformation could not be ensured for compositional programs if the term and program are A-closed. Instead, the program is required to contain only pattern term arguments.

Finally, the proof of correctness of confluent rewriting-based programs was established in Section 3.5. These theorems mirror those of the strong correctness of non-overlapping programs, as defined in Section 3.4.2.

The theorems of Section 3.4.1 were proved in order to demonstrate the approach to the proof of correctness; A-passive terms are A-closed by definition. Partial deduction [LS91], conjunctive partial deduction [LSdW96], and narrowing-based partial evaluation [AFV96b] all have a related closedness condition. Furthermore, the closedness condition of narrowing-based partial evaluation recursively inspects all the subterms of the term [AFV96a, AFV96b, AFJV97]. On the other hand, the narrowing-based language for which the closedness is defined is not as rich as the language considered in this thesis.

Evaluation order cannot be changed by partial deduction or conjunctive partial deduction. However, the introduction of nested functions, as in narrowing-based functional logic programming, can cause such a change. In [AFJV97], it is not entirely clear how the correctness of the narrowing-based partial evaluation procedure is established, since no proof is provided by the authors. In later work on the partial evaluation of lazy narrowing functional-logic programs [AAF$^+$98], Albert et al. restricted the unfolding to stop when weak head normal form is reached; imposing this constraint on the unfolding in the algorithm guarantees the correctness of the procedure for a lazy language having only pattern term arguments. However, even with a similar restriction on the construction of resultants, the arbitrary term arguments of *E* require the compositionality of the original program for the proof of correctness.

3.7 Summary

In this chapter, the partial evaluation procedure for rewriting-based functional logic programs was defined. The powerful computation mechanism of functional logic languages motivated the basic design of the procedure, while the functional aspects of the language required the addition of a ’restart step’ to ensure all possible computation paths for a term are represented in the specialised program.

The procedure is sound and complete for given classes of terms and programs. The set of terms used during the program transformation must be carefully constructed, as the specialisation has the capability of changing the evaluation order of nested specialised terms.

Chapter 4

An Algorithm for Constraint-based Partial Evaluation

In this chapter, the algorithm for constraint-based partial evaluation of rewriting-based functional logic programs is presented [LG98a, LG97]. The main algorithm for the partial evaluation procedure is defined in Section 4.1. Each of the features of this algorithm is discussed in detail, beginning in Section 4.2 with an introduction to m-trees, the structures used to record the specialised terms during the transformation. In Section 4.3, the functions which generate the resultants and, in doing so, extend and expand the m-tree are presented. In addition, an introduction to the constraint-based information propagation is included in this section. The engine of the partial evaluation procedure, the *restart* step, is defined in Section 4.3.1.

The latter part of this chapter is concerned with the local and global control of the algorithm (Sections 4.4 and 4.5) and the construction of the residual program from the terms of the m-tree (Section 4.6). The correctness of this algorithm is outlined in Section 4.7. In Section 4.8, several extensions of the procedure to further improve the quality of the residual programs are discussed. Finally, the features of this algorithm are compared with those of established specialisation techniques in Section 4.9.

4.1 The Algorithm

In the previous chapter, the theoretical foundations of the partial evaluation procedure for rewriting-based functional logic programs were presented. This chapter concentrates on the formalisation of the algorithm for constraint-based partial evaluation and its features. Detailed presentations of the restart step, the control of the algorithm, and the generation of the residual program will be presented in the later sections of this chapter.

The basic idea of the program specialisation is for a term t and program P, all possible computation paths of t in P are generated and represented in the residual program P'. This technique ensures that any instance of t has a correct computation in the residual program.

In order to generate all possible computation paths of t in P, the computation mechanism of the rewriting-based language is combined with a “restart step”. This step is used when it is unclear which rewrite rule of the program P will be applied to the term during the run-time computation. The possible computations of t in P are stored in a tree structure. Points at which the restart step is applied during partial evaluation form branching nodes in the tree (see Figure 3.1).

In general terms, the algorithm transforms a program as follows. For the partial evaluation of a program P with respect to a term t, the term t is stored in the root node of the tree. This tree of possible computations is repeatedly *extended* by adding partial computations to the leaves of the tree and *expanded* by branching the computation paths using the restart step. A graphical description of this procedure is shown in Figure 4.1. The terms in the leaves of the tree (1) are given as input to the interpreter (2). The interpreter generates deterministic partial computations for each term in the program P. These partial computations are added to the respective leaves of the tree (3).

Now, a partial computation can either end with a term in its normal form or not (§ 2.3.5).

- If the term is not in normal form, it is passed as input to the interpreter again on the next iteration of the procedure (5).
- If the final term of this partial computation is in normal form and does not contain any calls to functions defined in the original program P, the branch of the tree is *closed*.
- If the term is in normal form and contains calls to functions defined in the original program P, the restart step is used to obtain the set of all possible computations of the term at run-time. Since the final term has no redexes but contains calls to user-defined functions, there must be data that is required to evaluate the term further. Since this data may be available at run-time, a definition for the user-defined function must be included in the residual program to ensure its correctness (§ 3.1). The definitions in the original program are used to generate the set of restart terms. Since it is unclear which rewrite rule will be used at run-time to simplify the term, the restart step must generate all possible *minimal* instantiations of this term which may be encountered during a run-time computation. In this way, all possible computation paths

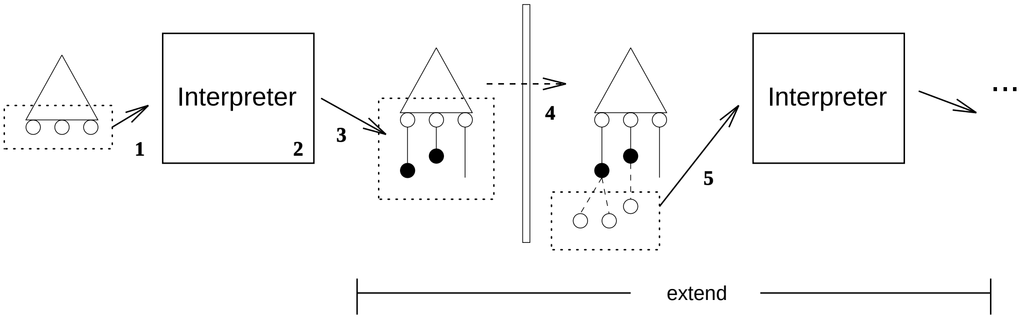


Figure 4.1: The cyclic application of the interpreter and the restart step during the partial evaluation procedure. The interpreter generates partial computations for each of the terms stored in leaves of the tree (1,2). The partial computations are added to their respective branches (3). Depending on the form of the final term, either a restart step is performed or a branch is closed (4). The cycle continues with the terms in the new leaves until all of the branches of the tree are closed (5).

> are simulated by the partial evaluator. These restart terms are added in leaf nodes of the tree (4). The restart terms are sent to the interpreter on the next iteration of the algorithm (5).

This process ends when all the branches of the tree are closed.

In relation to the theoretical framework of Chapter 3, the tree structure above not only contains the terms of the set A, but also the residual definitions of the terms. As noted earlier, restarting the terms stored in leaf nodes of the tree requires generating the restart terms for the terms of the set A. Resultants are stored implicitly in the tree since the partial computations are recorded on the branches. On each iteration of the procedure, the terms in the new leaves of the tree are bodies of resultants in the residual definition of the terms in A. Furthermore, some of the terms stored in the new leaf nodes are also members of A. This implies residual program extracted from the tree is guaranteed to be A-closed, as required by the correctness conditions of Section 3.5. This will be discussed in more detail in Section 4.7. However, as the next example demonstrates, the set of terms A is not guaranteed to be finite by this basic procedure; the expansion and extension of the tree described above has to be closely controlled to generate a *finite* computation tree.

The double-append benchmark from Example 3.2.5 provides a simple, yet illustrative example of the algorithm's basic operation.

Example 4.1.1 Given the following program P containing the list concatenation function definition:

```
FUNCTION  Concat : List(a) * List(a) -> List(a).
```

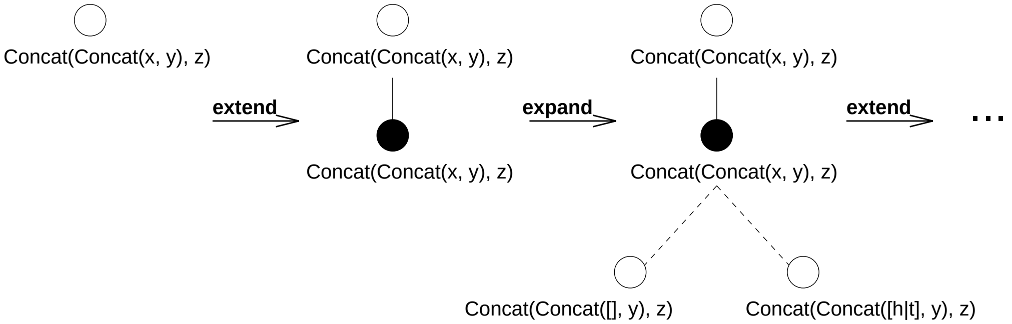


Figure 4.2: Extending and expanding the computation tree in order to record all possible computation paths. The term in the root node has no computation in the original program. Therefore, the restart step generates possible run-time instantiations of the term that are reducible in the original program. These are added to the tree in new leaf nodes on the branch.

```
Concat([],y)    => y.
Concat([h|t],y) => [h|Concat(t,y)].
```

consider the specialisation of P wrt $t =$ `Concat(Concat(x,y),z)` (as in the earlier examples from Chapter 3).

The term `Concat(Concat(x,y),z)` is stored in the root of the tree. According to the above description, the terms stored in the leaves of the tree are passed to the interpreter; in this case, the root is the only leaf. This term cannot be rewritten by either statement of P because of the occurrence of the variable `x` in place of the term argument of the possible redex. Therefore, the empty computation is returned by the interpreter. This indicates that a restart step must be performed — the residual program must contain at least a definition of `Concat` to be correct!

The two restart terms for t represent the possible run-time instantiations of the term which may be rewritten using the definition in P. Since the `Concat` function is a g-function, the schema statement applied at run-time depends on the value of the term argument of the redex. According to the definition of `Concat`, the term argument must have an outermost list constructor in order to be simplified using a schema statement in P. Therefore, for the term `Concat(Concat(x,y),z)` to be simplified at run-time, `x` must be instantiated with a term which reduces to either `[]` or `[h|t]`. The two restart terms for t are added to the computation tree. The cycle begins again; these terms are passed to the interpreter for evaluation, since they are now the leaves of the computation tree. Partial computations of both restart terms are added to the tree as shown in Figure 4.3.

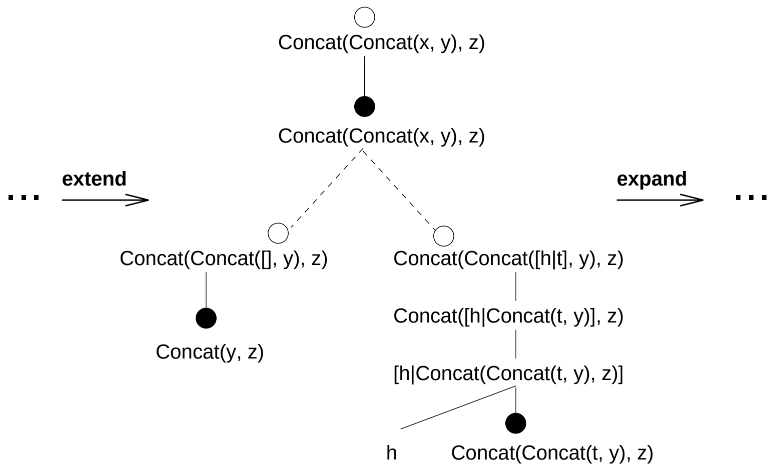


Figure 4.3: Extending the tree by adding partial computations of the restart terms to their respective branches. The unnecessary constructor function is removed in the right-hand branch of the tree.

Of course, this procedure does not limit the residual program to *only* evaluating instances of the terms `Concat(Concat([],y),z)` or `Concat(Concat([h|t],y),z)`. The procedure simply exploits the property that if the nested term `Concat(x,y)` will be simplified, then the run-time instantiation of `x` must reduce to one of the patterns before the term can be rewritten.

Three new terms occur in leaf nodes of the tree. As `h` does not contain any standard subterms, this branch of the tree is closed. The two other new terms in the leaf nodes of the tree need to be restarted, since they are in normal form (i.e. they have no redexes). The term `Concat(y,z)` is restarted using the definition of `Concat` in P, by instantiating `y`. The result of passing the restart terms `Concat([],z)` and `Concat([h|t],z)` to the interpreter is illustrated in Figure 4.4. The term `Concat(Concat(t,y),z)` is a renaming of the term that was restarted during the last iteration. It is also restarted again, using substitutions for `t`. Clearly, continuing this process will generate an infinite computation tree.

Unfortunately, as Example 4.1.1 shows, this iterative extend and expand procedure typically generates infinite trees, either with infinite deterministic computations or, as in the above example, with infinitely many restart steps on one or more branches. It is unsatisfactory for the partial evaluator not to terminate and return a residual program. In order to ensure termination of the specialiser, both of these possible sources of non-termination must be controlled. In this procedure, the object-level interpreter is extended with an ordering on terms to guarantee finite computations. This forms the local control of the unfolding in the algorithm; it is described in Section 4.4. To ensure that a branch only contains finitely many restarted terms, all branches of the tree are checked for repeated

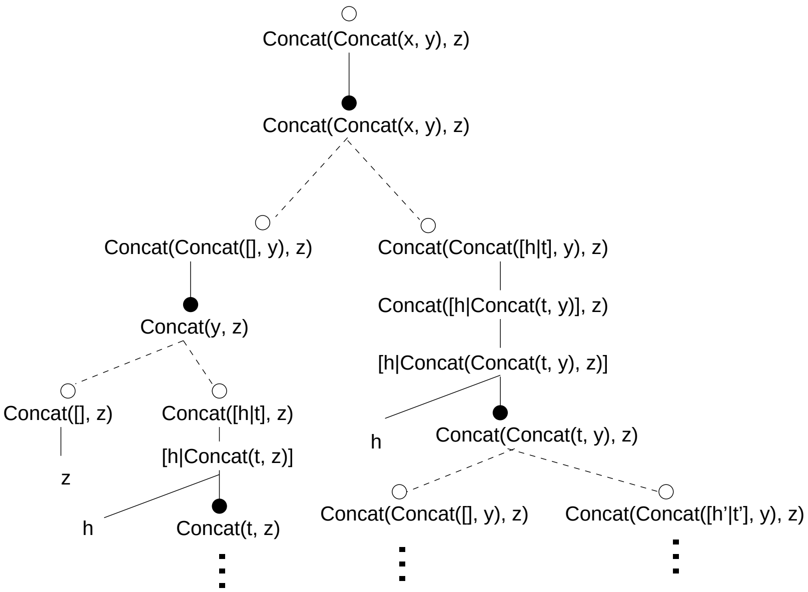


Figure 4.4: The infinite tree of possible computation paths for the double-append example. Infinitely many restart steps are performed on two branches of the tree. Therefore, the branches cannot be closed, and the algorithm does not terminate.

or growing terms. The global control of the algorithm, α, is described in Section 4.5.

The algorithm for constraint-based partial evaluation is presented below.

Definition 4.1.2 Algorithm

Input: a program *P*, a term *t*, and set of constraints C.
Output: a program P' which is the partial evaluation of *P* wrt *t*, and an m-tree of terms μ.
Initialisation:

```
  $P_0 := P \cup \{Ans(x_1, \ldots, x_n) \Rightarrow t\}$, where $FV(t) = (x_1, \ldots, x_n)$;
  $\mu_0 :=$ the tree containing one node labelled $(Ans(x_1, \ldots, x_n), C, \epsilon)$;
  $i := 0$;

repeat
  $\mu_{i+1} = \alpha(extend(\mu_i, P_0))$;
  $i = i + 1$;
until $\mu_i = \mu_{i-1}$
return $\mu_i$
$P' := \mathcal{R}_\sigma(\mu_i)$.
```

Given a program P and term t, the procedure begins by adding an extra statement to the original program P defining a function *Ans*. It is assumed that the function identifier *Ans* does not occur in P. The arguments of *Ans* are the free variables of t. The tree of all possible computation paths is initialised by storing the initial term in the root node. The function *extend* both extends, that is, adds partial computations to the appropriate branches, and expands the tree of computation paths by performing restart steps in a manner similar to that illustrated earlier in this section. The growth of this tree is controlled globally by the α function (§ 4.5). Finally, when the tree has stabilised, the residual program P' is extracted by means of the function $\mathcal{R}_\sigma$, which includes the renaming operations necessary to ensure the resultants are statements in the language (§ 3.2). This extraction process will be discussed in Section 4.6.

A major feature of this partial evaluation procedure is the incorporation of constraint solving to allow improved specialisation. This is achieved by propagating additional information about terms through the tree structure; this information cannot be directly represented in the terms, but can be represented by constraints associated with the terms of the computation. The constraint solving integration is not immediately obvious from the above presentation of the overall algorithm. Most of the handling of constraints is taken care of during the *extend* procedure and the control of the algorithm. However, it may be noted that the labelled tree structures used in the algorithm to store the possible run-time computations of t also store two other associated elements: the set of constraints for that term, and the subterm used to restart the computation path, called the *selected term*. Of course, the selected term may not exist for all labelled nodes. In this case, a dummy term ϵ is used in the place of the selected term in nodes that are not restarted (for example, in the root node of the tree). The labelled tree structures used to store the terms and their associated resultants, m-trees, are introduced in the next section.

4.2 M-trees

Section 4.1 introduced the constraint-based algorithm for the partial evaluation of rewriting-based functional logic programs. The core of the technique is an iterative extend and expand operation on the tree of all possible computation paths for a term t in a program P. Furthermore, in the previous section, it was shown that the terms of an m-tree are directly related to the set of terms A from the theoretical basis of the procedure (§ 3.2). As noted in the last chapter, partial deduction can be formalised similarly; a logic program is specialised with respect to a set of atoms and the run-time goal is a conjunction of instances of these atoms. Gallagher and Martens [MG95] showed that storing these atoms in a tree structure can provide additional dependence information to the partial evaluator. This allows better precision for the control of polyvariance. Similarly, in this technique,

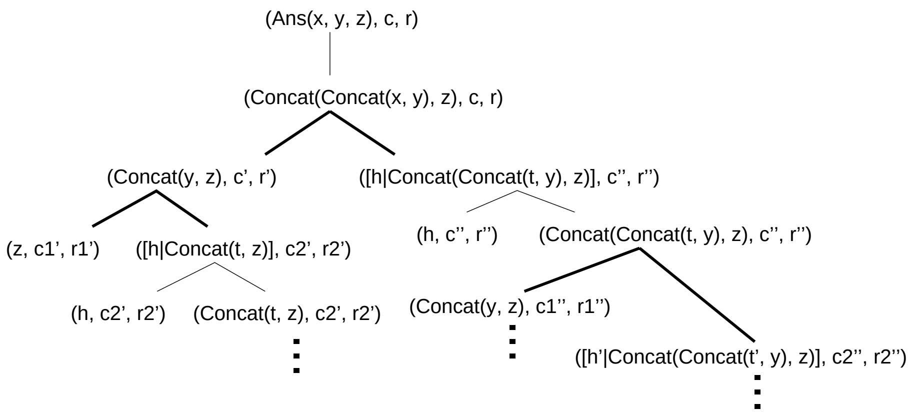

(Ans(x, y, z), c, r)
(Concat(Concat(x, y), z), c, r)
(Concat(y, z), c’, r’)
([h|Concat(Concat(t, y), z)], c’’, r’’)
(z, c1’, r1’)
([h|Concat(t, z)], c2’, r2’)
(h, c’’, r’’)
(Concat(Concat(t, y), z), c’’, r’’)
(h, c2’, r2’)
(Concat(t, z), c2’, r2’)
(Concat(y, z), c1’’, r1’’)
([h’|Concat(Concat(t’, y), z)], c2’’, r2’’)

Figure 4.5: The m-tree for double-append.

a tree structure is used to store the terms of the set A and the residual definitions of these terms; the name *m-trees* for these structures is adopted from [MG95].

Definition 4.2.1 (M-trees)
An *m-tree* μ is a labelled tree in which nodes can either be marked or unmarked. A node N is labelled with a tuple of terms (t_N, C_N, s_N), where t_N is the term of the computation, C_N is the set of constraints associated with term t_N, and s_N is the term used to restart the computation. For a branch β of the tree, $\mathcal{N}_\beta$ is the set of labels of β and for a leaf L of a tree, β_L is the unique branch containing L.

It is possible to consider m-trees as simplified computation trees, in which the terms of the partial computations are not recorded. The label of each node of an m-tree stores three elements: a term t, the set of constraints associated with t, and the subterm of t used in the restart step, the *selected term*. The latter elements will be discussed in detail later in the chapter. For now, let us refer to these elements simply using variables.

As an example, Figure 4.5 contains the m-tree representation for the double-append benchmark (Example 4.1.1). In this example, each arc in bold print represents a restart step and a call to the interpreter. The terms linked by a bold arc in the m-tree will be renamed to form the terms of a resultant in the residual program.

In addition, it will be useful to define the concept of an m-graph. M-trees are acyclic m-graphs. That is, in m-graphs, arcs from a child to an ancestor node in a branch are permitted. M-graphs are used in the procedure when a term is encountered twice in the branch. When this happens, an arc is added from the child node to the ancestor node containing the first occurrence of the term in the branch. In

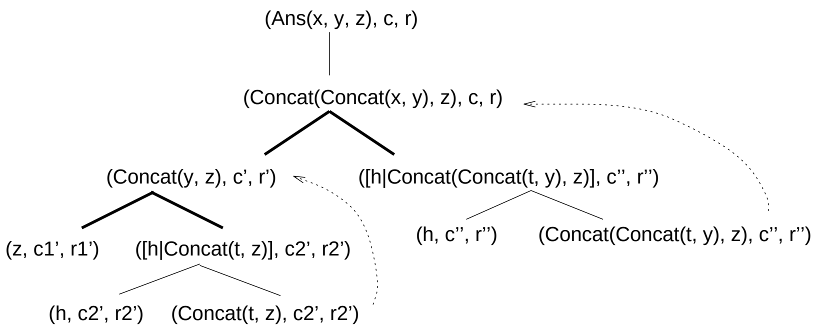

(Ans(x, y, z), c, r)
(Concat(Concat(x, y), z), c, r)
(Concat(y, z), c’, r’)
([h|Concat(Concat(t, y), z)], c’’, r’’)
(z, c1’, r1’)
([h|Concat(t, z)], c2’, r2’)
(h, c’’, r’’)
(Concat(Concat(t, y), z), c’’, r’’)
(h, c2’, r2’)
(Concat(t, z), c2’, r2’)

Figure 4.6: The m-graph for double-append.

this way, a finite m-graph can represent an infinite m-tree. As an example, the m-graph representing the m-tree of Figure 4.5 is shown in Figure 4.6. The term `Concat(Concat(t,y),z)` is a renaming of a term stored in an ancestor node. Therefore, an arc from the leaf node to the ancestor node is added to the m-graph.

4.3 Growing M-trees

In this section, the operations to generate the m-tree structure during the partial evaluation of a functional logic program are formalised. The operation on the tree was introduced broadly in Section 4.1 in terms of the extension and the expansion of the tree. The *extend* function incorporates both the expansion and the extension operations, adding children to a leaf node of an m-tree for branches that are not closed. Closed branches of the m-tree are indicated by marked leaf nodes.

Definition 4.3.1 $extend(\mu, P)$:
Given m-tree μ and program P, the m-tree $extend(\mu, P)$ is computed as follows.

```
$extend(\mu, P)$ = for all unmarked leaf nodes $L \in \mu$ with label $(t_L, C_L, s_L)$:
    mark $L$ in $\mu$;
    $B = specialise(t_L, C_L, P)$;
    if $B \neq \emptyset$,
        for all $(r, C, s) \in B$, add leaf $L'$ to branch $\beta_L$ with label $(solve(rhs(r), C), C, s)$;
    otherwise, if $B = \emptyset$,
        $t_L$ = Split($t_L$);
        $B'$ = covered($t_L, C_L, P$);
        for all $b \in B'$, add leaf $L'$ to branch $\beta_L$ with label $b$.
```

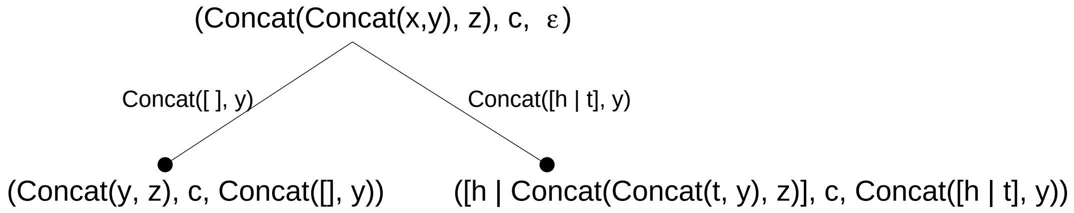

(Concat(Concat(x,y), z), c, ε)
Concat([ ], y)
Concat([h | t], y)
(Concat(y, z), c, Concat([], y))
([h | Concat(Concat(t, y), z)], c, Concat([h | t], y))

Figure 4.7: The initial section of the m-tree generated during the partial evaluation of the double-append example. The arcs of the m-tree have been annotated with the instances of the selected terms used to obtain the terms in the leaf nodes respectively.

The functions *specialise* (Def. 4.3.3), *solve* (Def. 4.3.7), and *covered* (Def. 4.3.8) are defined below.

Basically, for all branches of the m-tree that are not closed, the terms stored in the leaf nodes are passed to the *specialise* function. This function checks if the term is in normal form or not and performs a restart step, if required. In the event of a restart step, the substitutions used to generate the restart terms are checked against the current set of constraints to ensure their validity. This function returns a set of tuples containing a resultant in the residual definition of the term (r), an associated set of constraints (C), and the subterm of t used to restart the computation (s), if it exists. A new leaf node containing the body of the statement r is added to the branch. The following example illustrates the manipulation of an m-tree using *extend*.

Example 4.3.2 Consider again the double-append benchmark program and term (Example 4.1.1). The m-tree with the single leaf node labelled with the term $t =$ `Concat(Concat(x,y),z)` and set of constraints C is passed as input to the *extend* function. Since t is in normal form, the *specialise* function computes the restart terms for t using the definition of `Concat` in the program P. As shown in the earlier examples, the restart terms for t are the terms `Concat(Concat([],y),z)` and `Concat(Concat([h|t],y),z)`. The interpreter then generates a partial computation for each restart term in the program P to obtain the associated resultants.

The set B returned by the call to *specialise* is:

{(`Concat(Concat([],y),z) => Concat(y,z)`, C, `Concat([],y)`),
(`Concat(Concat([h|t],y),z) => [h|Concat(Concat(t,y),z)]`, C,
`Concat([h|t],y)`)}.

That is, the result is a set of tuples, with each tuple containing a schema statement, a set of constraints, and the term used to restart the computation.

The bodies of these statements are extracted from the set B and stored in labels in new unmarked leaves on the branch, as illustrated in Figure 4.7.

4.3.1 The Restart Step: Obtaining the Residual Definition

In this section, the *specialise* function is defined. As noted earlier, given a term t, this function performs a restart step in order to generate the set of restart terms for t (§ 3.2). Then, associated resultants are formed for each restart term. The computed residual definition of t with respect to P is returned by *specialise*.

Definition 4.3.3 *specialise*(t, C, P):
Let t be a term, C a set of constraints, and P a program. Let $\mathcal{S}(t, P) = s$ such that s is a subterm of t. Let H be the outermost function of s. Then, *specialise*(t, C, P) is the set of tuples:

$$\begin{aligned} \mathit{specialise}(t, C, P) = \{(t\theta_i \Rightarrow t_j, C\theta_i, s\theta_i) \mid h_i \Rightarrow b_i \in \mathrm{Def}_P^F \ \&\ \theta_i = \lceil s, h_i \rceil \neq \mathit{fail} \ \ \& \\ C\theta_i \text{ is satisfiable } \&\ t = e[s], \ \&\ e[b_i]\theta_i \Rightarrow^* t_j\} \end{aligned}$$

Otherwise, if $\mathcal{S}(t, P) = \epsilon$, $\mathit{specialise}(t, C, P) = \emptyset$.

The expression $\lceil s, t \rceil$ denotes the idempotent most general syntactic unifier (mgu) of terms s and t if it exists; otherwise, it equals *fail*.

The concept of restart terms for a term t was introduced in Section 3.2. Restart terms are the minimal instantiations of t that have non-trivial computations in the program. In the definition of *specialise* above, a *selection function* $\mathcal{S}(t, P)$ indicates the subterm of t which should be used to compute the restart terms for t. This selection function finds a standard subterm of t that would be selected as the next redex by the computation mechanism, if it was sufficiently instantiated to match the heads of any statements in the program. The restart terms for t are obtained by performing the syntactic unification of this term with the heads of the statements in P.

It may be the case that t does not contain any standard subterms. In this case, the selection function returns the dummy term ϵ and the *specialise* function returns the empty set. Otherwise, as noted earlier, the *specialise* function returns the residual definition of t with respect to the program P (§ 3.2). The *specialise* function checks satisfaction of the set of constraints with the applied mgu substitution in order to remove any resultants which are unreachable during the computation in the residual program.

Of course, the selection of the subterm of the term t which is used to generate the restart terms depends on the operational semantics of the target language. On the other hand, it is possible to define a general property of the selection function which should be satisfied by its formalisation.

Definition 4.3.4 (Select Property)
Given a program P, term t, let $S \subseteq \mathcal{U}_P(t)$ be the set of standard subterms of t which unify with at least one head of a statement in the program. Then, $s = \mathcal{S}(t, P) \in S$, the subterm of t selected for restarting the computation, contains the first demanded argument of t (§ 2.3.4).

For example, the following definition formalises a selection function for the *E* language.

Definition 4.3.5 $\mathcal{S}(t, P)$:
Given term t and program P, the selection function $\mathcal{S}(t, P)$ returns a selected term according to the following rules.

$$\begin{array}{lcl}
\mathcal{S}(x, P) & = & \epsilon \\
\mathcal{S}(c(b_1, \ldots, b_n), P) & = & \epsilon \\
\mathcal{S}(c(b_1, \ldots, b_{i-1}, t_i, t_{i+1}, \ldots, t_n), P) & = & \mathcal{S}(t_i, P) \\
\mathcal{S}(p(t_1, \ldots, t_n), P) & = & \epsilon \\
\mathcal{S}(f(t_1, \ldots, t_n), P) & = & f(t_1, \ldots, t_n) \\
\mathcal{S}(g(t_0, t_1, \ldots, t_n), P) & = & \begin{cases} g(t_0, t_1, \ldots, t_n) \text{ if } match(t_0, g, P) \\ g(t_0, t_1, \ldots, t_n) \text{ if } \mathcal{S}(t_0, P) = \epsilon \\ \mathcal{S}(t_0, P), \ \mathrm{otherwise} \end{cases} \\
\mathcal{S}(\texttt{LAMBDA}[x](t), P) & = & \epsilon \\
\mathcal{S}(\texttt{ITE}(t_1, t_2, t_3), P) & = & \mathcal{S}(t_1, P) \\
\mathcal{S}(t_1 \ t_2, P) & = & \mathcal{S}(t_1, P) \\
\mathcal{S}(\{x \mid t\}, P) & = & \mathcal{S}(t, P)
\end{array}$$

where *match*(t_0, g, P) holds if for some $s \in TA(g, P)$, and some substitutions θ, ρ , $t_0\theta$ is syntactically equivalent to $s\rho$.

The function *match* in the definition of $\mathcal{S}$ is necessary since the term arguments of g-functions can be any terms, not only patterns as in traditional functional programming languages. Therefore, when a term t with outermost g-function is encountered, the term argument must be tested. If it matches a term argument in the definition of g in P, the term t is the selected term.

Example 4.3.6 Consider the following program P defining an inverter function `Inv`.

```
FUNCTION   Inv : Boolean -> Boolean.

Inv(True)  => False.
Inv(False) => True.
```

In order to generate the residual definition of the term $t = \texttt{Inv(Inv(x))}$, the restart terms for t are computed. By the definition of the selection function, $\mathcal{S}(t, P) = \texttt{Inv(x)}$. Therefore, the restart terms of t are the terms $\texttt{Inv(Inv(True))}$ and $\texttt{Inv(Inv(False))}$. Based on these restart terms, a residual definition of t is:
$\{\texttt{Inv(Inv(True)) => True}, \texttt{Inv(Inv(False)) => False}\}$

The *solve* function allows the computation in *specialise* to advance by finitely more steps by using constraint solving to eliminate conditional expressions.

Definition 4.3.7 *solve*(t, C)
Let t be a term, C be a set of constraints. Then, *solve*(t, C) is defined as follows.
If $t = \texttt{ITE}(c, t_1, t_2)$ where c can be represented as a constraint in one of the domains of the partial evaluator, then either
solve$(t, C) =$ *solve*$(t_1, C \cup \{c\})$, if $C \cup \{\neg c\}$ is unsatisfiable, or
solve$(t, C) =$ *solve*$(t_2, C \cup \{\neg c\})$, if $C \cup \{c\}$ is unsatisfiable;
else, *solve*$(t, C) = t$.

Therefore, if the body of the schema statement returned by *specialise* is a conditional expression which has a condition representable by the constraint domains of the partial evaluator *and* one of the branches can be removed based on the current set of constraints for that expression, *solve* replaces the conditional body with the appropriate subterm of the expression. In this way, constraint solving is applied as early and often as possible during partial evaluation in order to use the extra information to the best degree. Only finitely many computation steps can be added to the partial computation, since the outermost conditional is removed with each call to *solve*.

The *covered* function (Def. 4.3.8) includes the functionality of the *solve* function; that is, constraint solving is used to determine whether a branch of the conditional can be removed by the partial evaluator. The duplication is necessary to ensure that branches of conditional statements are removed as early as possible from the computation. As a result of the global control of the algorithm 4.5, conditional statements may occur in leaf nodes of the m-tree that do not result from a call to *specialise*. Therefore, the *covered* function must integrate the functionality of *solve*, in order to ensure unreachable branches are removed, with an operation to ensure that the resulting program is A-closed, a task not addressed by the basic *solve* function.

4.3.2 Constraint Generation and Splitting Terms

The latter part of the *extend* function handles terms stored in unmarked leaf nodes which contain standard subterms but cannot be restarted. Usually this is caused by demanded arguments having outermost primitive/built-in functions. Restart terms cannot be generated for such terms, for two reasons. The *specialise* function does not have access to the definitions of primitive functions. Secondly, the definitions of these functions should not be extended needlessly by extra schema statements in the residual program. Therefore, terms with primitive outermost functions cannot be restarted. In this case, the *specialise* function will return the empty set.

On the other hand, it may be the case that the term in the leaf node cannot be restarted, yet it contains standard subterms which may have to be simplified at run-time. Therefore, the *covered* function is invoked.

Definition 4.3.8 *covered*(t, C, P):
Let t be a term, C a set of constraints, and P a program.
If $t = \texttt{ITE}(c, t_1, t_2)$ where c can be represented as a constraint in one of the domains of the partial evaluator, then
covered$(t, C, P) = \{(t_1, \mathit{proj}\,(t_1, C \cup \{c\}), \epsilon)\}$, if $C \cup \{\neg c\}$ is unsatisfiable, or
covered$(t, C, P) = \{(t_2, \mathit{proj}\,(t_2, C \cup \{\neg c\}), \epsilon)\}$, if $C \cup \{c\}$ is unsatisfiable, or
covered$(t, C, P) = \{(t_1, \mathit{proj}\,(t_1, C \cup \{c\}), \mathit{Thn}(c)), (t_2, \mathit{proj}\,(t_2, C \cup \{\neg c\}), \mathit{Els}\,(c))\}$, otherwise;
else, *covered*(t, C, P) = *split*(t, C, P).

The *Thn* and *Els* functions in the labels returned by *covered* simply record the branch of the conditional expression from which the term was extracted.

The *covered* function performs two operations. If t is a conditional expression, then if the condition is representable by a constraint, the test is extracted from the conditional. This constraint is added to the current set of constraints C, and the satisfiability of the new set is checked. If this set is unsatisfiable, the then-branch of the conditional can be removed. Likewise, the negation of the constraint is added to the set of constraints, and its satisfiability is also checked. If both sets are satisfiable, the constraint and its negation are each added to C, and the two sets of constraints are associated with the terms in the then and else-branches of the conditional, respectively. Then, these term and constraint pairs are stored in leaves on the branch of the m-tree. This process is illustrated in Figure 4.8. This step forms the constraint-based information propagation of the algorithm. Otherwise, the *covered* function calls the *split* function (Def. 4.3.10) to remove the subterm that is preventing the generation of restart terms.

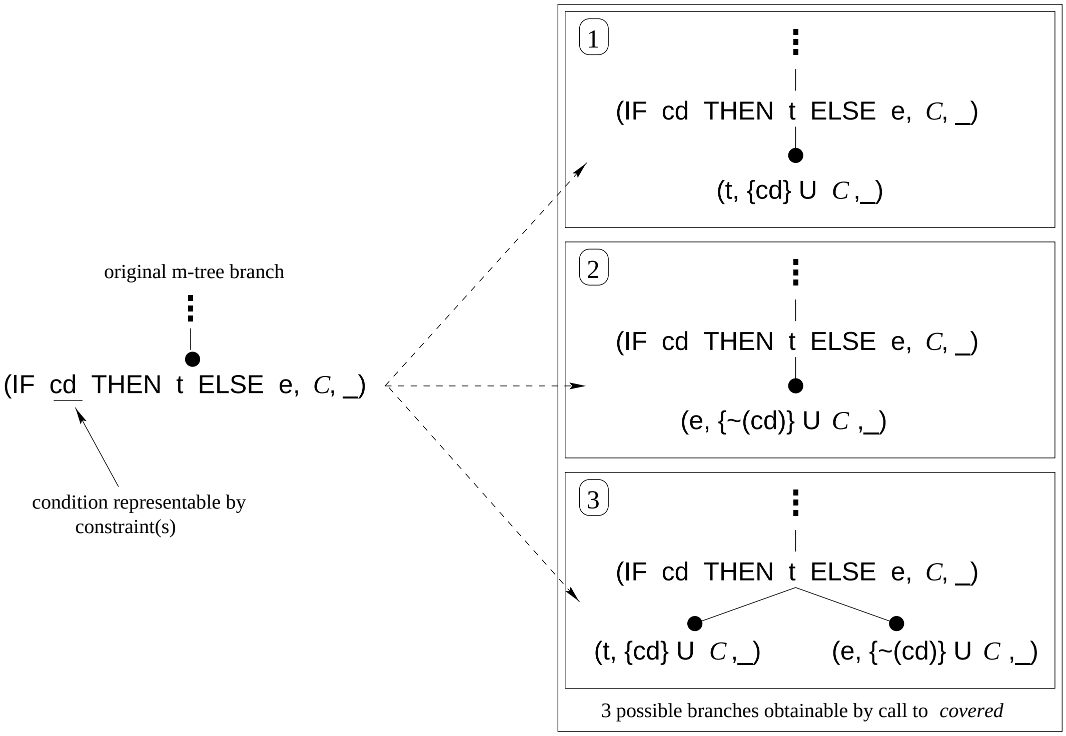

original m-tree branch
(IF cd THEN t ELSE e, C, _)
condition representable by constraint(s)
1
(IF cd THEN t ELSE e, C, _)
(t, {cd} U C ,_)
2
(IF cd THEN t ELSE e, C, _)
(e, {~(cd)} U C ,_)
3
(IF cd THEN t ELSE e, C, _)
(t, {cd} U C ,_)
(e, {~(cd)} U C ,_)
3 possible branches obtainable by call to covered

Figure 4.8: Propagating the positive and negative information of a conditional statement, when the information can be represented by a constraint of the partial evaluator. The conditional expression stored in the leaf node of the branch of the m-tree, shown on the left side of the diagram, can be transformed in one of three ways. In the first case (1), the constraint set $\{\sim (cd)\} \cup C$, that is the set of constraints resulting from adding the negation of the condition to the set of constraints in the leaf node, is unsatisfiable. This means the term in the else-branch of the conditional is unreachable, and therefore, can be eliminated from the tree. Likewise, in the second case (2), the constraint set $\{cd\} \cup C$ is unsatisfiable. This indicates that the term in the then-branch of the conditional will not be encountered by any computation in the residual program, and can be removed. Finally, in the third case (3), both constraint sets $\{cd\} \cup C$ and $\{\sim (cd)\} \cup C$ are satisfiable. Therefore, both terms must be added to the branch of the m-tree, as they are both reachable. In each case, the new set of constraints is associated with the terms of the then-branch and else-branch.

Using constraints to store both the positive and negative information of a conditional statement is a powerful addition to the partial evaluation procedure. In particular, conditional statements and case statements are a key element of functional logic programming style, since the heads of program statements often have to be pattern and instance non-overlapping. Testing the satisfiability of the constraints allows the possible pruning of unreachable computations, thus increasing the efficiency and decreasing the size of the residual program.

Example 4.3.9 Consider the specialisation of the program P containing a definition of McCarthy's 91-function.

```
FUNCTION  F : Integer -> Integer.

F(x) =>
    IF    x > 100
    THEN  x - 10
    ELSE  F(F(x + 11)).
```

Consider the specialisation of the above program with respect to the term `F(x)`. Although static data are not available, the specialisation is intended to improve the efficiency of the program by altering its recursive structure.

During the specialisation of P with respect to the term `F(x)`, the call to *specialise* returns:
$B = \{($`F(x) => IF x > 100 THEN x - 10 ELSE F(F(x + 11))`$, C, \epsilon)\}$

On the next iteration of the algorithm, the *specialise* function will be called again with the body of the statement in B. This term cannot be restarted by *specialise* since all the functions occurring in the test of the conditional are primitive functions. However, there exists a standard subterm in the else-branch of the conditional. Therefore, the *covered* function is invoked. Assuming the linear arithmetic constraint `100 < x` is representable in one of the constraint domains of the partial evaluator, this constraint is added to the set C, and the satisfiability of $C \cup \{$`100 < x`$\}$ is tested. Likewise, the negation of the constraint, `x =< 100`, is added to the set of constraints C, and its satisfiability is tested. Assuming C is the empty set, both constraint sets are satisfiable. Two new leaf nodes storing the terms in the branches of the conditional expression are added to the m-tree branch, as shown in Figure 4.9.

Finally, the *split* function is defined and its operation is demonstrated with an example.

Definition 4.3.10 *split*(t, C, P):
Let t be a term, and C a set of constraints. Let $z_1, z_2, \ldots$ be variables not occurring in t or C.

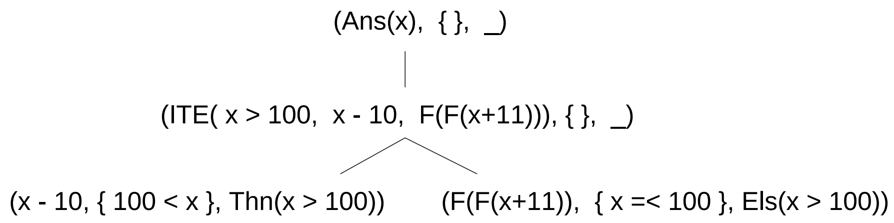

(Ans(x), { }, _)
(ITE(x > 100, x - 10, F(F(x+11))), { }, _)
(x - 10, { 100 < x }, Thn(x > 100))
(F(F(x+11)), { x =< 100 }, Els(x > 100))

Figure 4.9: M-tree resulting from the partial evaluation of McCarthy's 91-function, after one application of the *extend* function.

If $\mathcal{S}(t,P) = h(t_0,\ldots,t_i,t_{i+1},\ldots,t_n)$ where $t_0,\ldots t_i$ are demanded by the computation, then
split$(t,C,P) = \{(h(z_0,\ldots z_i,t_{i+1},\ldots t_n), \mathit{proj}\,((h(z_0,\ldots z_i,t_{i+1},\ldots t_n),C),\epsilon),$
$(t_0,\mathit{proj}\,(t_0,C),\epsilon),\ldots (t_i,\mathit{proj}\,(t_i,C),\epsilon)\}$,
else, if $t = \mathtt{ITE}(t_1,t_2,t_3)$, then
split$(t,C,P) = \{(t_1,\mathit{proj}\,(t_1,C),\epsilon), (t_2,\mathit{proj}\,(t_2,C),\epsilon), (t_3,\mathit{proj}\,(t_3,C),\epsilon)\}$,
otherwise, *split*$(t,C,P) = \{\}$.

The function *proj(t,C)* is the projection of the constraints in the set C onto the free variables of the term t (§ 2.2.2).

Basically, the select function $\mathcal{S}(t,P)$ will return a subterm of t that either would be chosen as the redex, if its demanded argument was instantiated, or contains a subterm that would be chosen as the redex, but the subterm cannot be restarted by the procedure. In the first case, the term is restarted by the *specialise* function. In the second case, the subterm that cannot be restarted must be removed from the term. This is the vital operation to ensure the closedness of the residual program (§ 3.5). This splitting operation is demonstrated in the following simple example.

Example 4.3.11 Consider the partial evaluation of the following program P with respect to the term `Inv(x & y)`.

```
FUNCTION  Inv : Boolean -> Boolean.

Inv(True)  => False.
Inv(False) => True.
```

The partial evaluator cannot restart the subterm with outermost built-in logical operator &. The inverter function `Inv` is a g-function with one demanded argument. The *specialise* function cannot determine the restart terms of `Inv(x & y)` that have non-trivial computations in P, since the

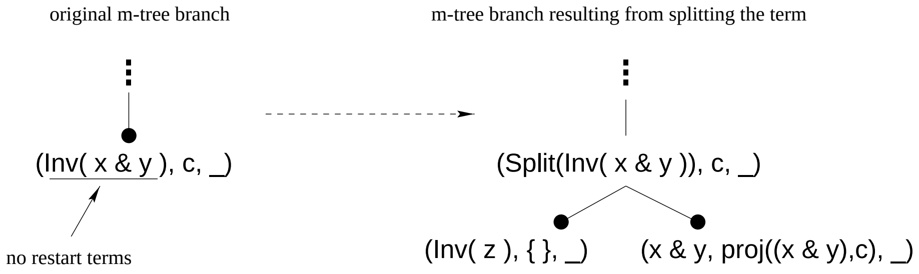

original m-tree branch
m-tree branch resulting from splitting the term
(Inv(x & y), c, _)
no restart terms
(Split(Inv(x & y)), c, _)
(Inv(z), { }, _)
(x & y, proj((x & y),c), _)

Figure 4.10: M-tree resulting from splitting the leaf node of the branch, using the *split* function. The built-in logical operator `&` in the term argument of `Inv` cannot be restarted by the algorithm. This subterm must be removed in order to generate a residual definition for `Inv`.

partial evaluator does not have access to the definition of the built-in logical operator `&`. In this case, the set returned by the call to *specialise* is the empty set; that is, the *specialise* function has failed to generate a residual definition in this case. However, a definition for `Inv` must be included in the residual program for the transformation to be correct.

The solution lies in extracting the subterm `x & y` from the term. According to the *split* function definition, the first argument is replaced with a new variable, z. The set B' returned to the *extend* function is:

$\{ (\texttt{Inv(z)}, \{\}, \epsilon), (\texttt{x \& y}, C, \epsilon)\}$

Since the variable `z` does not occur in t or C, the projection of the constraints in C onto the free variables in `Inv(z)` is the empty set.

Each node label in B' is stored in new unmarked leaf nodes on the appropriate branch of the m-tree. This process is illustrated in Figure 4.10. In addition, the term t_L in the label of the parent node is changed to *Split*(`Inv(x & y)`).

Performing this splitting operation ensures that there is a residual definition for the term `Inv(x & y)` in the specialised program. On the next iteration of the algorithm, the residual definition for the term `Inv(z)` is generated by the *specialise* function and added to the left-hand branch of the m-tree. Of course, no specialisation is gained by generating the residual definition of `Inv(z)`, but the semantics of the program has been preserved by performing this operation. The residual program is shown below.

```
FUNCTION   Ans : Boolean * Boolean -> Boolean.
Ans(x,y) => I(x & y).

FUNCTION   I : Boolean -> Boolean.
```

```
I(True)  => False.
I(False) => True.
```

Of course, the *split* function is also language dependent; the terms which must be removed are directly dependent on the operational semantics of the target language. Therefore, *split* uses the selection function to identify the term argument which should be removed, if there exists a selected subterm. The *split* function will divide a conditional expression if its condition cannot be represented by a constraint in the partial evaluator. This operation is not performed in the *covered* function. The *covered* function only transforms conditional expressions which have a condition that can be represented as a constraint.

Example 4.3.11 illustrates the use of a split node, one of the special nodes of an m-tree. These nodes only aid the extraction of the residual program from the m-tree; otherwise, they do not affect the transformation procedure.

This completes the formalisation of the procedures to form the residual definitions for the terms of the constraint-based partial evaluation. The interaction between the *specialise*, *covered*, and *split* functions ensure that all user-defined functions occurring in terms of the computations are processed during specialisation. In the following sections, functions to control the construction of the m-trees during partial evaluation will be presented.

4.4 Local Control

Up to now, the procedures that construct the m-tree and therefore generate the residual definitions for terms occurring in the tree have been defined. However, as Example 4.1.1 illustrated, the operations on the m-tree are not guaranteed to terminate. In fact, it is very rare for the procedure described above to terminate. The unfolding and the restart step must be carefully controlled to ensure that the object-level computations are finite and the number of restart steps performed is finite (that is, the program is specialised with respect to a finite set of terms). This describes the local and global control of the algorithm respectively. In this section, the local control of the algorithm will be defined. The global control of the specialisation algorithm is formalised in Section 4.5.

Computations in rewriting-based functional logic languages are not guaranteed to terminate (i.e. $\Rightarrow$ may not be strongly normalising). Infinite computations can result from a redex being evaluated infinitely many times or from a redex which is growing. Consider the following example of a non-terminating computation in the *E* rewriting-based functional logic language.

Example 4.4.1 Consider the following program, containing a definition of Append:

```
FUNCTION   Append : List(a) * List(a) * List(a) -> Boolean.

Append(u,v,w) =>
    (u = [] & v = w) \/
    SOME [r,x,y] (u = [r|x] & w = [r|y] & Append(x,v,y)).
```

The call Append(u,v,w) will result in a non-terminating computation. For example, the first three terms of the computation are the following:

```
Append(u,v,w)

((u = []) & (v = w)) \/ SOME [r_1,x_1,y_1] ((u = [r_1|x_1]) &
(w = [r_1|y_1]) & Append(x_1,v,y_1))

((u = []) & (v = w)) \/ SOME [r_1,x_1,y_1] ((u = [r_1|x_1]) &
(w = [r_1|y_1]) & (((x_1 = []) & (v = y_1)) \/
SOME [r_2,x_2,y_2] ((x_1 = [r_2|x_2]) & (y_1 = [r_2|y_2]) &
Append(x_2,v,y_2))))

...
```

In each term, the redex Append(x1,x2,x3) is simplified. The only instantiations of this term having finite computations are those in which either the first argument or the third argument is a closed list.

On the other hand, by supplementing the interpreter with an ordering on the terms, the computations of arbitrary terms are ensured to be finite. Imposing a well-founded ordering on the terms of a sequence guarantees its finiteness. This application of orderings on terms is based on the *tree theorem* of Kruskal [Kru60] and later work on the termination of rewriting systems by Dershowitz [Der87]. This is not the only method for ensuring strong normalisation, but one of the most powerful [Klo92].

The basic idea behind the local control is that computations are arrested when the ordering of the terms is violated in the sequence. In this algorithm, the redexes used in a computation step are checked against the redexes of previous computation steps for an ordering violation. If such a violation occurs, the computation step is not performed, and the partial computation is returned by the interpreter. Otherwise, the computation is permitted to proceed.

4.4.1 The Homeomorphic Embedding Relation

The formalisation of the local control of the algorithm begins with the definition of an ordering on terms in the language. This ordering will be used to identify “growing terms”, which are a source of possible non-termination of computations. The following definitions are from [Leu97b].

Definition 4.4.2 (wfo)
A strict partial order $>_S$ on a set S is an anti-reflexive, anti-symmetric, and transitive binary relation on $S \times S$. A sequence of elements $s_1, s_2, \ldots$ is called admissible wrt $>_S$ iff $s_i >_S s_{i+1}$ for all $i \geq 1$. The binary relation $>_S$ is a well-founded order (wfo) iff there is no infinite admissible sequence wrt $>_S$.

A quasi order $\geq_S$ on a set S is a reflexive and transitive binary relation on $S \times S$.

Definition 4.4.3 (wqr, wqo)
Let $\leq_S$ be a binary relation on $S \times S$. A sequence of elements $s_1, s_2, \ldots$ is called admissible wrt $\leq_S$ iff there are no $i < j$ such that $s_i \leq_S s_j$. The binary relation $\leq_S$ is a well-quasi relation (wqr) on S iff there are no infinite admissible sequences wrt $\leq_S$. If $\leq_S$ is a quasi order on S, then $\leq_S$ is a well-quasi order (wqo) on S.

The strict homeomorphic embedding relation $\trianglelefteq$ [LM96] is a well-quasi relation [Leu97b]. If a newly computed term violates the embedding relation, the term may have been seen before or may be growing. For two terms s_1 and s_2, $s_1 \prec s_2$ indicates that s_2 is a strict instance of s_1.

Definition 4.4.4 (strict homeomorphic embedding)

- For variables x, y, $x \trianglelefteq y$.
- For terms $s, H(t_1, \ldots, t_n)$, $s \trianglelefteq F(t_1, \ldots, t_n)$ if $s \trianglelefteq t_i$ for some i.
- For terms $H(s_1, \ldots, s_n), H(t_1, \ldots, t_n)$, $H(s_1, \ldots, s_n) \trianglelefteq H(t_1, \ldots, t_n)$ if $s_i \trianglelefteq t_i$ for all i and $H(s_1, \ldots, s_n) \not\prec H(t_1, \ldots, t_n)$.

The strict homeomorphic embedding extends the homeomorphic embedding operator by requiring that the instance relation does not hold. So, for example, $H(x, x) \not\trianglelefteq H(x, y)$ but $H(x, y) \trianglelefteq H(x, x)$.

4.4.2 Safe Computations

Imposing an ordering on the redexes of the computation ensures that the sequence of redexes is not infinite. Safe redexes are those that do not embed any of the previous redexes of the computation, according to the strict homeomorphic ordering relation on terms.

Definition 4.4.5 (safe redexes)
Given a partial computation $\{t_i\}_{i=1}^m$, the set of *safe redexes*, L_m^S, is defined inductively as follows.

- For $m = 1$, $L_1^S = L_1$, the set of redexes of t_1.
- For $m > 1$, if $\exists r \in L_m$ and $\exists r' \in L_j^S$ such that $r' \trianglelefteq r$ for some $1 \leq j < m$, then $L_m^S = \emptyset$; otherwise, $L_m^S = L_m$.

A safe computation step is a computation step that only uses safe redexes.

Definition 4.4.6 (safe computation step)
A term t_{j+1} is obtained from a term t_j by a *safe computation step* if the following are satisfied:

1. The set of safe redexes of t_j, L_j^S, is a non-empty set.
2. For each redex r_i in L_j^S such that $t_j = e[r_i]$,
   - r_i is identical to the head h_i of some instance $h_i \Rightarrow b_i$ of a statement schema (rule), and
   - $t_{j+1} = e[b_i]$, the result of replacing r_i by b_i in t_j.

Definition 4.4.5 restricts the redexes that can be used in the computation mechanism of the object language. If a redex in the set of redexes renames or embeds a redex of a previous stage of the computation, the computation stops. Otherwise, a redex could be evaluated infinitely many times (as in Example 4.4.1), and a partial computation may not be returned by the interpreter.

Of course, there are terms for which it is preferable to evaluate an “unsafe” redex. The implication of this technique is that the new interpreter will terminate more often than the original interpreter for the target language. That is, in some cases, this technique identifies terminating computations as potentially non-terminating, thus ending the computation prematurely, as in the double-append example below.

Example 4.4.7 Consider again the double-append benchmark of Example 4.1.1.

Example 4.3.2 illustrated the operation of the *specialise* function to generate the restart terms for the term `Concat(Concat(x,y),z)`. The term `Concat(Concat([h|t],y),z)` is a restart term for this example. This term is passed to the interpreter in order to generate its associated resultant.

Unfortunately, simply extending the interpreter with an ordering on the redexes as defined in Definition 4.4.6 does not allow the generation of the residual statement which eliminates the redundant data structure (see Example 3.2.5). The first redex is the subterm `Concat([h|t],y)`. Using the schema statements in the definition of `Concat`, this term reduces to

`Concat([h|Concat(t,y)],z)`.

This entire term forms the next redex of the computation. However, `Concat([h|t],y)` $\trianglelefteq$ `Concat([h|Concat(t,y)],z)`. The redex is deemed to be unsafe, and the computation ends.

Clearly, this technique can be improved. The two redexes have the same outermost function name and share the same structure, but actually, they are not the same redex and in this case there is no danger of non-termination. There are two ways of avoiding this early termination of the partial computations.

1. Extend the condition of Definition 4.4.5 to check both the embedding of both the redexes and the terms. That is, even if the embedding relation holds for some redexes in the sets L_j^S and L_m^S, allow the computation step to be performed. After the computation step, if $t_j \trianglelefteq t_m$, stop the computation.
2. Index the f and g-functions in the term using natural numbers. When a redex is rewritten, all the new f and g-functions are annotated with the index of the outermost function of the redex. That is, for term $e[r]$, where the outermost function of r has index i and r is rewritten using the schema statement $h \Rightarrow b$, the new functions of b are indexed with i.

The homeomorphic embedding relation is expensive to compute with respect to term size. Therefore, indexing the functions is an inexpensive technique to rename apart the redexes in order to allow more precision in the local control of the algorithm.

Example 4.4.8 By indexing the g-functions in the term `Concat`$_1$`(Concat`$_2$`(x,y),z)`, it is possible for the interpreter to differentiate between the two redexes. Using indexing, the safe computation is permitted to proceed until a term in normal form is reached.

Definition 4.4.9 (safe indexed redexes)

Let η be a mapping from terms to indexed terms, such that $\eta(t)$ assigns a unique index i to each functor in t. Given a partial computation $\{t_i\}_{i=1}^{m}$, the set of *safe indexed redexes*, L_m^I, is defined inductively as follows.

- For $m = 1$, $L_1^I = L_1$, the set of redexes of $\eta(t_1)$.
- For $m > 1$, if for some k, $\exists r_k \in L_m$ and $\exists r'_k \in L_j^I$ such that $r'_k \trianglelefteq r_k$ for some $1 \leq j < m$, then $L_m^I = \emptyset$; otherwise, $L_m^I = L_m$.

A safe computation step is redefined in terms of safe *indexed* redexes. A safe computation $\{t\}_{i=1}^{n}$, $t \Rightarrow^{\phi} t_n$, is the sequence of terms obtained using the safe indexed computation steps where $L_n^I = \emptyset$.

Definition 4.4.10 (safe computation, $\Rightarrow^{\phi}$):
Given a safe [indexed] computation $\{t\}_{i=1}^{n}$, $t_1 \Rightarrow^{\phi} t_n$.

The proof of termination of the unfolding step during partial evaluation now follows from the following theorem, known as Higman's lemma, which is a special case of Kruskal's tree theorem [Hig52]. The proofs of both this lemma and Kruskal's theorem can be found in [NW63].

Theorem 4.4.11 Let $t_1, t_2, \ldots$ be an infinite sequence of terms over a finite set of function names, constructor names, and variable names. Then there exist $i < j$ such that $t_i \trianglelefteq t_j$.

Therefore, it follows that the sequence of redexes used in a computation cannot be an infinite sequence. If the sequence of redexes is finite, then by definition, the computation must be finite.

Theorem 4.4.12 For every term t, the safe computation of t is a finite sequence.

Proof

Assume the safe computation $t = t_1, t_2, \ldots$, the safe computation of t in program P, is an infinite sequence. Then, by definition of computation step, there exists an infinite sequence of associated sets of redexes: $L_1^S, L_2^S, \ldots$. Assume without loss of generality that there only exists one redex in each set $L_k^S = \{r_k\}$. Then, by Theorem 4.4.11, there exist $i < j$ such that $r_i \trianglelefteq r_j$. Contradiction, by definition of a safe computation step (Def. 4.4.5). □

Thus, safe computations are guaranteed to be finite. This ensures the termination of the unfolding step in the constraint-based partial evaluation algorithm.

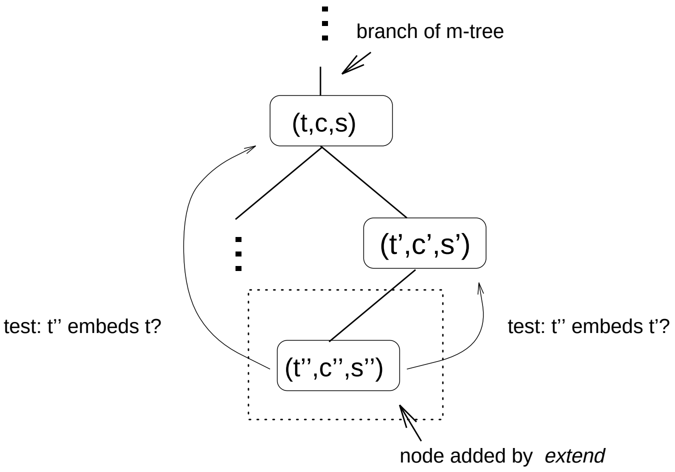


Figure 4.11: Testing the embedding relation for terms stored in new leaves of an m-tree. The result of the call to *extend* is stored in the leaf node of the branch β. Each of the terms stored in ancestor nodes on the branch are tested to ensure the ordering of the terms is not violated by the addition of the new term.

4.5 Global Control

In the last section, the local control of the partial evaluation algorithm was formalised. This ensures the finite unfolding of terms in the target language. However, as Example 4.1.1 illustrated, there is also a potential for non-termination resulting from the restart step of the procedure. In particular, the global control of the algorithm must guarantee that the branches of the m-tree have finite length. This implies that only finitely many restart steps are permitted on a branch of the m-tree. On the other hand, the algorithm specialise enough terms in order to be able to achieve the optimal specialisation of the program. That is, the global precision should be maximised, while guaranteeing the finiteness of the m-tree.

In order to control the growth of the m-tree, an ordering is imposed on the terms of the tree as in the local control (§ 4.4). After a term is stored in a new leaf node on a branch of the tree, a check is performed to ensure that the new term does not violate the ordering of the terms on that branch.

The global control procedure, using the homeomorphic embedding relation as the well-quasi ordering of the terms (Def. 4.4.4), is illustrated in Figure 4.11. Consider the branch β of Figure 4.11. A new leaf node L'' is added to β as a result of calling *extend* (Section 4.3). This leaf node stores the new term t''. If the ordering on the branch is violated by the addition of the new term, i.e. for some (t, C, s) in $\mathcal{N}_\beta$, $t \trianglelefteq t''$, then it is not safe to restart the term t''.

If the relation $t \trianglelefteq t''$ holds for some (t, C, s) and (t'', C'', s'') in $\mathcal{N}_\beta$, there are two possible scenarios:

> *Instance* case: The term t'' is an instance or a renaming of t. For example, the infinite branches of the m-trees created during the specialisation of the double-append benchmark are a result of adding terms which are instances of earlier terms stored in the branch (Figure 4.5).
>
> *Embedding* case: The term t is embedded in t''. This indicates that the terms are growing on the branch of the m-tree.

As an example of potential non-termination resulting from a growing term, consider the following partial evaluation.

Example 4.5.1 Consider the program below, describing a function `Rev` which returns its argument in reverse order:

```
FUNCTION   Rev : List(a) -> List(a).
Rev(x)     => R(x,[]).

FUNCTION   R : List(a) * List(a) -> List(a).
R([], y)   => y.
R([h|t],y) => R(t,[h|y]).
```

The specialisation of this program with respect to the term `Rev(x)` is known as the *accumulating parameter* benchmark. With each restart step, the term grows, but each new term is not an instance of any term already specialised in the tree. The infinite m-tree constructed during the partial evaluation of this term is shown in Figure 4.12.

In this section, two functions are defined that prevent the non-termination of the program transformer for each of these cases.

4.5.1 Folding Expressions

When an *Instance* scenario is encountered, the m-tree must be *folded*. The folding function fold is defined as follows.

Definition 4.5.2 $\mathit{fold}(\beta)$:
For a m-tree branch β containing an unmarked leaf L labelled (t_L, C_L, s_L), $\mathit{fold}(\beta)$ is the m-graph

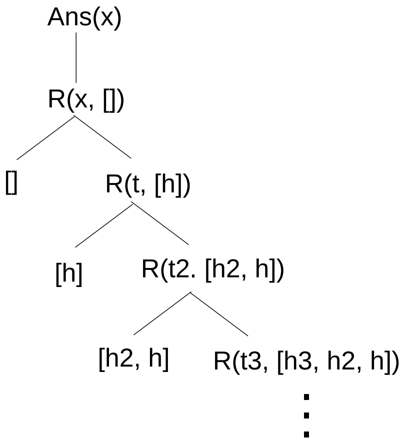


Figure 4.12: The infinite m-tree resulting from the specialisation of the reverse program. For simplicity, only the terms of the nodes are shown.

branch defined as follows.
If there exists $(t_N, C_N, s_N) \in \mathcal{N}_\beta$ on node $N(\neq L)$ such that the following conditions hold:

- $t_N\theta = t_L$,
- $\theta = \{x_1 = t_1, \ldots, x_m = t_m\}$, and
- C_L entails C_N,

then *fold* the branch by performing the following operations:

- replace t_L with $Fold(x_1, \ldots, x_m)$,
- draw a dashed line to N, and
- add leaves to β with labels $(t_1, proj(t_1, C_L), \epsilon), \ldots, (t_m, proj(t_m, C_L), \epsilon)$;

else, $fold(\beta) = \beta$.

Recall that *proj*(t, C) is the projection of the set of constraints C onto the free variables of the term t (§ 2.2.2).

As an example of folding an m-tree, consider the double-append example. The leaf L storing term t_L = `Concat(Concat(t,y),z)` is added to β, indicated in Figure 4.13 by the dotted rectangular box. There is an ancestor node in branch β, call it N, storing the term t_N =

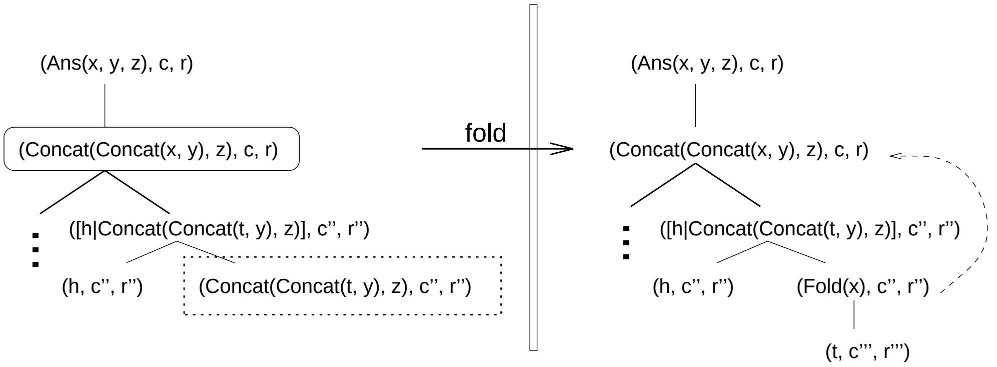

(Ans(x, y, z), c, r)
(Concat(Concat(x, y), z), c, r)
([h|Concat(Concat(t, y), z)], c'', r'')
(h, c'', r'')
(Concat(Concat(t, y), z), c'', r'')
fold
(Ans(x, y, z), c, r)
(Concat(Concat(x, y), z), c, r)
([h|Concat(Concat(t, y), z)], c'', r'')
(h, c'', r'')
(Fold(x), c'', r'')
(t, c''', r''')

Figure 4.13: Folding the m-tree during the double-append specialisation.

`Concat(Concat(x,y),z)`. The term t_L is a renaming instance of t_N. Assume for this example that C_L entails C_N. Then, the β branch of the m-tree is folded by replacing t_L with a new term *Fold*`(x)`, a dashed arc is drawn in the m-graph to the node N and the leaves corresponding to the terms of the substitution θ are added to the tree as new leaves of β.

It is assumed that *Fold* is a restricted function not occurring in any program P. Like the split-node introduced in Section 4.3, the *Fold* indicates a fold-node in the m-graph. A reference to an m-tree in the following denotes the acyclic m-graph resulting from omitting the dashed folding arcs.

4.5.2 Generalising Expressions

If an *Embedding* scenario arises as a result of adding a leaf to the branch β, there is a danger that the terms on β are growing. In this case, β must be generalised to avoid an infinitely growing restart term on the branch. The following definition is from [GS96].

Definition 4.5.3 (generalisation, msg, mssg)
A *generalisation* of t_1 and t_2 is a triple (t, θ_1, θ_2) where θ_1, θ_2 are substitutions, $t\theta_1 = t_1$, and $t\theta_2 = t_2$. A *most specific generalisation (msg)* of t_1 and t_2 is a generalisation (t, θ_1, θ_2) of t_1 and t_2 such that for every generalisation $(t', \theta'_1, \theta'_2)$ of t_1 and t_2, t is an instance of t'. A *most specific safe generalisation (mssg)* of t_1 and t_2 is a most specific generalisation (t, θ_1, θ_2) where θ_i does not introduce bound variables into t_i.

The most specific *safe* generalisation of two terms t and s, *mssg*(t,s) is computed using the following rewrite rules, beginning with $t_g = x$, $\theta_1 = \{x = t\}$ and $\theta_2 = \{x = s\}$.

If $h \neq$ `LAMBDA` :

$$\left(\begin{array}{l} t_g \\ \{x = h(t_1,\ldots,t_n)\} \cup \theta_1 \\ \{x = h(s_1,\ldots,s_n)\} \cup \theta_2 \end{array}\right) \rightarrow \left(\begin{array}{l} t_g\{x = h(y_1,\ldots,y_n)\} \\ \{y_1 = t_1,\ldots,y_n = t_n\} \cup \theta_1 \\ \{y_1 = s_1,\ldots,y_n = s_n\} \cup \theta_2 \end{array}\right)$$

$$\left(\begin{array}{l} t_g \\ \{x = t, y = t\} \cup \theta_1 \\ \{x = s, y = s\} \cup \theta_2 \end{array}\right) \rightarrow \left(\begin{array}{l} t_g\{x = y\} \\ \{y = t\} \cup \theta_1 \\ \{y = s\} \cup \theta_2 \end{array}\right)$$

Since this procedure is defined in terms of syntactic unification, a most specific safe generalisation is guaranteed to exist. These rules compute a conservative generalisation of the terms, since the rules prevent the inspection of subterms in lambda expressions. This restriction can be relaxed by examining the structure of these subterms and ensuring that terms containing bound variables are not included in the resulting substitutions.

A most specific safe generalisation is either the same as, or more general than the most specific generalisation of the terms. The difference is that the mssg does not permit the inspection of lambda expressions. This is necessary to avoid variable capture in the resulting residual program.

For example, the msg of the terms `F(LAMBDA[x](G(x)))` and `F(LAMBDA[x](x))` is `F(LAMBDA[y](z))`. This term will be specialised and a residual definition will occur in the program. However, it is necessary to be able to substitute the `x` variable for `y` or `z` at run-time. Clearly, this is not possible. The mssg operator is discussed further in Section 4.8.1.

The generalisation of a branch of an m-tree can now be defined.

Definition 4.5.4 $gen(\beta)$:
For a branch β containing an unmarked leaf L labelled (t_L, C_L, s_L), $gen(\beta)$ is the m-graph branch defined as follows.
If there exists $(t_N, C_N, s_N) \in \mathcal{N}_\beta$ on node $N(\neq L)$ such that the following conditions hold:

- $s_N \trianglelefteq s_L$ and
- $t_N \trianglelefteq t_L$,

then *generalise* in the following manner:

- let $mssg(t_L, t_N) = (t, \theta_1, \theta_2)$ where $\theta_2 = \{x_1 = t_1, \ldots, x_m = t_m\}$;
- if t is not a variable, then:
  1. delete the branch β from after N to L;
  2. replace t_N with $Gen(t_N, x_1, \ldots, x_m)$;
  3. generate new set of constraints $C = \mathcal{W}((t_N, C_N), (t_L, C_L))$;
  4. add $m + 1$ leaves to branch β_N with labels $(t, proj(t, C), \epsilon)$, $(t_1, proj(t_1, C), \epsilon), \ldots$ $(t_m, proj(t_m, C), \epsilon)$;
- otherwise, if t is a variable:
  1. replace t_L with $Gen(t_L, y)$;
  2. let $t_L = e[H(s_1, \ldots, s_k)]$ where $t_N \trianglelefteq H(s_1, \ldots, s_k)$ and H is the outermost function of t_N;
  3. add two leaves to branch β with labels $(e[z], C_L, \epsilon)$, $(H(s_1, \ldots, s_k), proj(H(s_1, \ldots, s_k), C_L), \epsilon)$;

else, $gen(\beta) = \beta$.

First, the selected subterm which was used to restart the computation must embed a selected subterm of an ancestor. Using this check reduces the occurrences of generalising when there is no danger of a growing term. Then, if the term of the ancestor node is embedded in the term of the leaf node, a generalisation must be performed.

In Definition 4.5.4, the operator $\mathcal{W}$ is a widening operator. Given two tuples $(t_1, C_1), (t_2, C_2)$ where t is a term and C is a set of constraints, the operator $\mathcal{W}$ computes a widening of C_1 and C_2 [CC77]. The widening operator $\bigtriangledown$ is a mapping from $D \times D$ to D defined as follows [CC77, CH78]:

$$\forall C, C' \in D : C \Rightarrow^E C \bigtriangledown C'$$
$$\forall C, C' \in D : C' \Rightarrow^E C \bigtriangledown C'$$

where $C_1 \Rightarrow^E C_2$ if the set of constraints C_1 entails the set C_2.

As an example of the functionality of the *gen* operator, consider again the specialisation from Example 4.5.1.

Example 4.5.5 The specialisation of the reverse program in Example 4.5.1 showed the possible non-termination of partial evaluation resulting from a growing term on the branch of the m-tree.

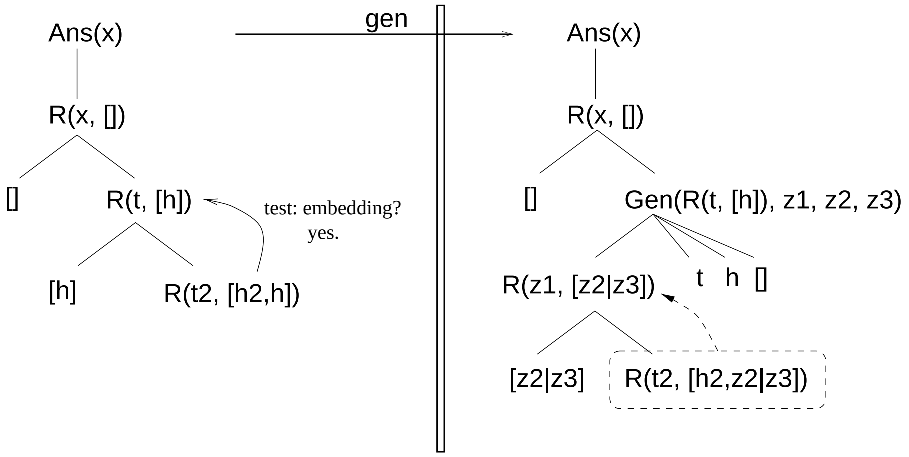


Figure 4.14: Generalising a branch of the m-tree to prevent an infinite branch. For the sake of simplicity, only the terms in the node labels are shown. The term `R(t2,[h2,h])` embeds the term `R(t,[h])`. Therefore, the branch is generalised by deleting it up to the node containing `R(t,[h])`, adding the msg to the branch, and continuing the partial evaluation. After generalisation, it is possible to fold the branch as indicated in the m-tree.

In this example, each restart causes the production of an internal list constructor in the new term. Termination does not occur, since each new term is not an instance of any previous terms.

In this case, the terms of the branch must be generalised, in order to isolate the growing argument. The basic idea is by removing this argument, the subterm which is obstructing the folding of this branch is eliminated.

The application of the *extend* function results in the addition of the terms `[h]` and `R(t2,[h2,h])` to branch β of the m-tree (Figure 4.14). Let L be the leaf node of β such that $t_L =$ `R(t2,[h2,h])` and N be the node storing the term `R(t,[h])`. The ordering relations $s_N \trianglelefteq s_L$ and $t_N \trianglelefteq t_L$ are satisfied for these terms. Therefore, the msg of the terms `R(t2,[h2,h])` and `R(t,[h])` is computed. The msg is the term `R(z1,[z2|z3])`. Since the msg is a non-variable term, the branch β is deleted up to the node N, and the msg and the terms of the substitution, `t`, `h`, and `[]`, are added in labels of new leaf nodes of β.

By generalising the term, it is possible on the next iteration to fold the branch, using the *fold* function, as shown in Figure 4.14.

4.5.3 Selecting the Ancestor

Both the $fold$ and gen functions above depend on the existence of a term t_N stored in an ancestor node on the branch β. Consider again the simple m-tree shown in Figure 4.11. Suppose *both* terms t and t' were embedded in the term t'', but $t \not\trianglelefteq t'$. Which term should be selected for generalisation or folding?

Traditionally, the term stored in the ancestor node closest to the leaf on the branch is selected. This will be called the *bottom-up* ancestor selection strategy. Alternatively, one could select the term stored in an ancestor node closest to the root of the m-tree. This is the *top-down* strategy. A third approach to the selection problem is to search the branch for the “best” term for which the embedding relation holds. This is the *selection-metric* approach.

The bottom-up strategy is guaranteed to result in the least amount of destruction of the branch during the generalisation step. This is a clear disadvantage of the top-down strategy: there is a potential that nearly the entire branch could be lost if a term near the root node is generalised. On the other hand, the term near the top of the branch will typically be the best choice for folding: for example, consider Figure 4.11 with $t = F(G(x))$, $t' = F(y)$ and $t'' = F(G(G(z)))$.

Finding the optimal term on a branch for folding or generalising requires the definition of a measure of “best fit”. This measure could be defined in many ways. A naive approach may be to record the number of couplings during the homeomorphic embedding test.

Definition 4.5.6 (selection metric)
Given two terms t, t' such that $t \trianglelefteq t'$, define the selection metric $\chi(t, t')$ as follows:

- If $t = H(s_1, \ldots s_n)$ and $t' = H(s'_1, \ldots s'_n)$, then $\chi(t, t') = \Sigma_{i=1}^{n} \chi(s_i, s'_i)$;
- Otherwise, $\chi(t, t') = 0$.

Alternatively, the term could be further inspected, and a value could be subtracted for each depth-level of the term that is searched. This problem is related to that of the translation operation in Chapter 3. By imposing one of the strategies, the terms on the branch are ordered. This eliminates the non-determinism in the choice of an ancestor. Several examples demonstrating the need for a selection strategy are presented in Chapter 5.

4.5.4 Removing Outermost Functions

The final component of the global control of the algorithm is a function which ensures the terms stored in the unmarked leaf nodes of the tree have an outermost user-defined function. In particular, outermost primitive functions occurring in the terms stored in unmarked leaves are removed from the term. This is an iterative procedure; if a term in an unmarked leaf node has an outermost function that is not user-defined, the outermost function is stripped from the term, and the function's arguments are stored in new unmarked leaf nodes on the branch. Then, the $divide$ function checks if the new terms have outermost user-defined functions.

Definition 4.5.7 $divide(\beta)$:
Given a branch β, let $divide(\beta)$ be defined by the following procedure:

```
**initialise:**
    Let $\beta_0 = \beta$.

**repeat**
    For all unmarked leaf nodes $L$ of $\beta_i$:
        $B = divide(t_L, C_L)$;
        for all $b \in B$, add a leaf $L'$ to $\beta_{i+1}$ with label $b$;
    $i = i + 1$;
**until** $\beta_{i+1} = \beta_i$.
**return** $\beta_{i+1}$.
```

This process terminates either if a user-defined function is found to be the outermost function of the term or if an indivisible term is encountered. For example, during the specialisation of the double-append example (Example 4.1.1), the term `[h|Concat(Concat(t,y),z)]` is divided by the removal of the outermost list constructor.

Definition 4.5.8 $divide(t, C)$:
Let t be a term and C a set of constraints.

$$\begin{array}{lcll}
divide(t,C) & = & \{(t_1, proj(t_1, C), \epsilon), \ldots (t_n, proj(t_n, C), \epsilon)\} & \text{if } t = c(t_1, \ldots, t_n) \\
divide(t,C) & = & \{(t_1, proj(t_1, C), \epsilon), \ldots (t_n, proj(t_n, C), \epsilon)\} & \text{if } t = p(t_1, \ldots, t_n) \\
divide(t,C) & = & \{(t, proj(t, C), \epsilon)\} & \text{if } t = \texttt{LAMBDA}[x](t) \\
divide(t,C) & = & \{(t_1, proj(t_1, C), \epsilon), (t_2, proj(t_2, C), \epsilon)\} & \text{if } t = t_1\ t_2 \\
divide(t,C) & = & \emptyset & \mathrm{otherwise}
\end{array}$$

The outermost functions are removed if they are not user-defined because the partial evaluator does not have access to their definitions. Therefore, the term cannot be restarted by *specialise*. On the other hand, removing outermost primitive functions has a negative effect on the specialisation of Boolean expressions. The $divide$ function will be redefined in Section 4.8.1 in order to improve the handling of Boolean expressions in the algorithm.

4.5.5 The Alpha Operator

Finally, the complete global control of the algorithm α can be defined. The α operator incorporates the folding and generalisation operations with the *divide* function as defined in Definition 4.5.8 above.

Definition 4.5.9 $\alpha(\mu)$:
Given a m-tree μ, $\alpha(\mu)$ is computed by the following procedure. For all branches β in μ with unmarked leaves:

```
**initialise:**
    Let $\beta_0 = \beta$.

**repeat**
    $\beta' = divide(\beta_i)$;
    $\beta_{i+1} = fold(\beta')$;
    if $\beta_{i+1} = \beta'$ then $\beta_{i+1} = gen(\beta')$;
    $i = i + 1$;
**until** $\beta_{i+1} = \beta_i$.
**return** $\beta_{i+1}$.
```

The global control of the algorithm guarantees the construction of a finite m-tree.

Theorem 4.5.10 For all programs P and terms t, the m-tree generated during the partial evaluation of P wrt t is finite.

Proof
By definition of the global control operator α and Higman's lemma (Theorem 4.4.11). The global control of the algorithm only replaces terms stored in leaf nodes with strictly smaller terms. □

Theorems 4.4.12 and 4.5.10 guarantee the termination of the constraint-based partial evaluator.

4.6 Extracting the Residual Program

The procedure for constructing a finite m-tree during the partial evaluation of functional logic programs has been defined in the previous sections. This section concerns the definition of the function $\mathcal{R}_\sigma$ which generates a residual program from a finite m-tree. This involves the *extraction* of the residual definitions from the m-tree and the *renaming* of the resultants in these definitions in order to obtain the statements of the residual program.

The extraction and translation function $\mathcal{R}_\sigma$ takes a finite m-tree as input. An intermediate function T_σ extracts the residual definition of a non-leaf node of the m-tree. The concept of a translation function was defined previously in Section 3.2. The $\mathcal{R}_\sigma$ function incorporates a translation function based on a renaming operator σ.

Definition 4.6.1 $\mathcal{R}_\sigma(\mu)$:
For a given m-tree μ and renaming function σ, $P' = \mathcal{R}_\sigma(\mu)$ is the set of schema statements extracted from μ.

$$P' = \bigcup\{T_\sigma(N,\mu) \mid N \text{ is a non-leaf node in } \mu\}$$

That is, given a non-leaf node, the residual definition for the term stored in that node is extracted by means of the function T_σ. A function B_σ, defined below, generates the terms in the bodies of the statements.

Definition 4.6.2 $T_\sigma(N,\mu)$:
For a given m-tree μ, non-leaf node N in μ, and renaming operation σ, $T_\sigma(N,\mu)$ is a set of schema statements extracted from node N using μ in the following manner.
Let (t_N, C_N, s_N) be the label of node N and $N_0, \ldots N_m$ its immediate child nodes.

$$\begin{array}{lcll}
T_\sigma(N,\mu) & = & \{\sigma(t_N) \Rightarrow B_\sigma(N_0,\mu)\{z_i = B_\sigma(N_i,\mu)\}_{i=0}^m\}, & \text{if } t_N = \mathit{Split}(t). \\
T_\sigma(N,\mu) & = & \{\sigma(t_N) \Rightarrow B_\sigma(N_0,\mu)\{x_i = B_\sigma(N_i,\mu)\}_{i=0}^m\}, & \text{if } t_N = \mathit{Gen}(t, x_1, \ldots, x_n). \\
T_\sigma(N,\mu) & = & \{\}, & \text{if } t_N = \mathit{Fold}(x_0, \ldots, x_n). \\
T_\sigma(N,\mu) & = & \{\sigma(t_N)\ \theta_i \Rightarrow B_\sigma(N_i,\mu)\}_{i=0}^m & \text{otherwise.}
\end{array}$$

where the substitutions $\theta_i, 0 \leq i \leq m$ are extracted from the selected terms stored in the nodes $N_0, \ldots N_m$.

Definition 4.6.3 $B_\sigma(N,\mu)$:

For node N in m-tree μ with term label t_N and immediate children nodes $N_0, \ldots, N_m$:

$$\begin{array}{lcll}
B_\sigma(t_N) & = & \sigma(t) & \text{if } t_N = \mathit{Split}(t). \\
B_\sigma(t_N) & = & \sigma(t) & \text{if } t_N = \mathit{Gen}(t, y_1, \ldots, y_n). \\
B_\sigma(t_N) & = & \sigma(t')\{x_i = B_\sigma(N_i, \mu)\}_{i=0}^{m}, & \text{if } t_N = \mathit{Fold}(x_0, \ldots, x_n). \\
B_\sigma(t_N) & = & t_N, & \text{if } N \text{ leaf node.} \\
B_\sigma(t_N) & = & \sigma(t_N), & \text{otherwise.}
\end{array}$$

where t' in the *Fold* case is the term stored in the node indicated in the m-graph by the folding arc from node N.

This extraction and renaming operation is described graphically in Figure 4.15. The following example completes the specialisation of the double-append benchmark by demonstrating the construction of the residual program from the m-tree.

Example 4.6.4 Given the program P and term t = `Concat(Concat(x,y),z)` from Example 4.1.1, the m-graph shown in Figure 4.16 is constructed as a result of the partial evaluation of P wrt t, assuming the set of constraints $C = \{\}$. In the m-graph of Figure 4.16, the constraints have been omitted, since they are unchanged by the transformation, and leaves with variable terms have been removed for simplicity. The selected terms are indicated on the arcs of the m-graph in Figure 4.16.

Assume the σ renaming function is the mapping which constructs the function name by appending the index of the node to the string "`FN_SP`."[1] The arguments of the function is the sequence of free variables of the term stored in the node. Based on this informal definition, σ performs the following assignments for the double-concat example:

$\sigma($`Concat(Concat(x,y),z)`$) =$ `FN_SP1(x,y,z)`, and

$\sigma($`Concat(x,y)`$) =$ `FN_SP2(x,y)`.

The program extracted from the m-graph by the $\mathcal{R}_\sigma$ function is:

```
FN_SP1([],y,z)     => FN_SP2(y,z).
FN_SP1([w|ws],y,z) => FN_SP3(w,ws,y,z).

FN_SP2([], z)      => z.
FN_SP2([v|vs],z)   => FN_SP4(v,vs,z).
```

[1] This is the renaming approach of the implementation, described in Chapter 5.

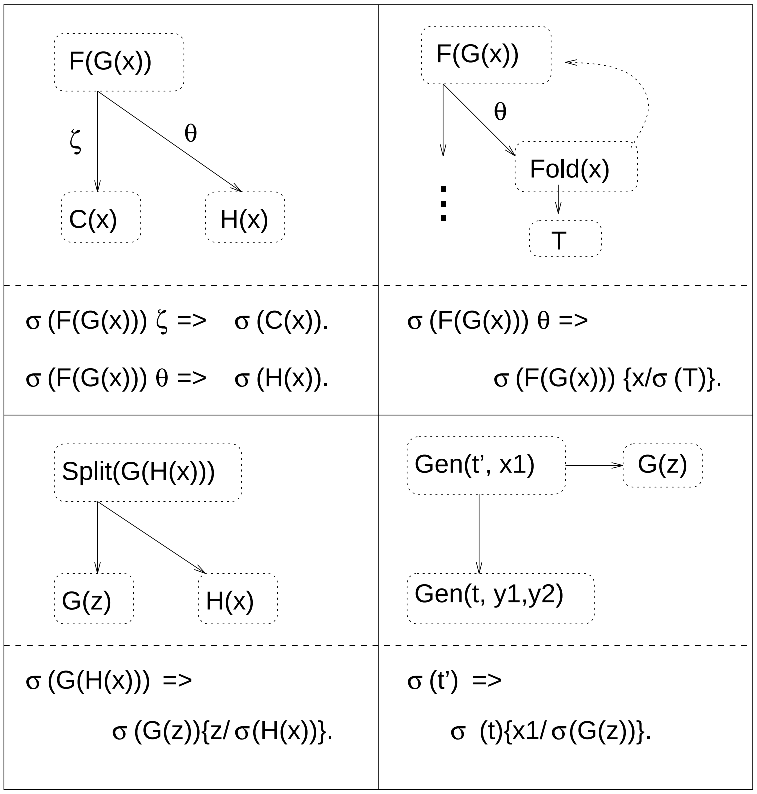


Figure 4.15: Example extractions of schema statements from m-trees. The top left corner presents an example in which a restart step is performed. The terms are renamed using σ, then the substitutions are applied. The top right shows and example of a fold-node in the tree. The bottom left example demonstrates the reconstruction of a term that has been split. Finally, in the bottom right corner, extraction of a generalised term is performed.

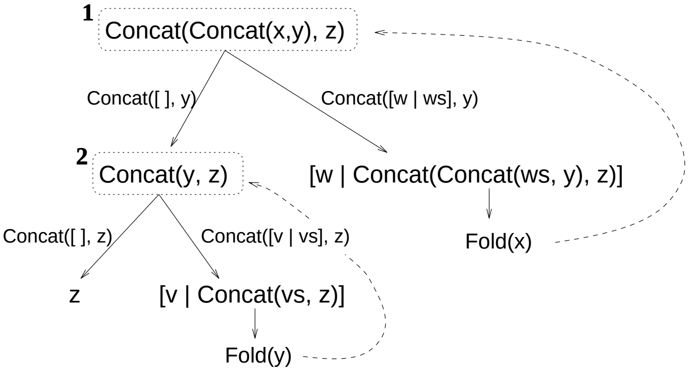


Figure 4.16: M-graph from the partial evaluation of double-append.

```
FN_SP3(w,ws,y,z)   => [w | FN_SP1(ws,y,z)].

FN_SP4(v,vs,z)     => [v | FN_SP2(vs,z)].
```

Then, after post-unfolding, the residual program returned by the partial evaluator is the following program.

```
FN_SP1([], y, z) => FN_SP2(y, z).
FN_SP1([x_3 | y_4], y, z) =>
    [x_3 | FN_SP1(y_4, y, z)].

FN_SP2([], x_5) => x_5.
FN_SP2([x_5 | y_6], z_7) =>
    [x_5 | FN_SP2(y_6, z_7)].
```

This section is concluded with the definition of the post-unfolding program transformation.

Definition 4.6.5 (post-unfolding)
Let P' be a residual program. For all n-ary f-functions F defined in P' such that $\mathrm{Def}^F_{P'} = \{h \Rightarrow b\}$, replace all terms $e[F(t_1, \ldots, t_n)]$ with $e[b\{x_i = t_i\}^n_{i=1}]$. Then, remove the definition of F from P'.

This simple definition of post-unfolding does not guarantee its termination. Ideally, post-unfolding should remove all residual statements in the definitions of f-functions that are only encountered once during a computation. This simple post-processing definition can be supplemented with one of the existing methods for ensuring the termination of post-unfolding, such as checking if there are two or more calls to the function in the residual program [JGS93].

4.7 Relating the Algorithm to the Theory

The procedure for the partial evaluation of rewriting-based functional logic programs is correct with respect to the theoretical framework described in Chapter 3. In this section, the operations of the algorithm are related to components of the theoretical definition in the previous chapter. A formal proof of the correctness of the algorithm is beyond the scope of this thesis.

Generating Residual Definitions

In Chapter 3, the partial evaluation of a program P is defined with respect to a set of terms A (§ 3.2, Def. 3.2.11). In order to specialise the program, a residual definition is formed for each element of A (Def. 3.2.6). A residual definition for a term t with respect to a program P is a set of resultants, one for each restart term of t. According to Definition 3.2.2, if t has a non-trivial computation in P, the only restart term for the term t is itself. Otherwise, the restart terms of t are the minimal instantiations of t having non-trivial computations in P.

In the algorithm, the terms stored in the non-leaf nodes of the m-tree form the elements of the set A (§ 4.1). Residual definitions are generated by the *specialise* function (§ 4.3.1). The *specialise* function calls the selection function in order to obtain the subterm of t that is demanded by the computation, call it s. Assuming that the outermost function of s is H, the heads of the schema statements in the definition of H are unified with the term s. For each statement head, if the unification does not fail, the most general unifier is applied to the term t to form a restart term of t.

The resultants are implicitly stored in the m-tree by storing the result of the partial computation in the leaf nodes of the tree on every iteration (§ 4.2).

Translating the Resultants

The translation operation defined in Section 3.2 is necessary to ensure that the resultants are transformed into statements of the language (Def. 3.2.10). In the algorithm, such a translation operation is performed during the extraction of the program (§ 4.6). This operation is performed by the functions T_σ and B_σ. Since every non-leaf node of the m-tree is an element of the set A, the translation function simply renames the non-leaf nodes and leaves the terms in the leaf nodes unchanged.

The functions T_σ and B_σ are parameterised by a renaming operation σ. This operation ensures that the new function names are unique, and the arguments of the function are the free variables of the term. A particular renaming function was not defined; a simple renaming function indexes the nodes of the m-tree and uses these indices in the function names to ensure their uniqueness.

The ordering function ω is explicitly defined in the m-tree by the arcs from folding nodes to ancestor nodes on the branch. This clearly indicates which terms of the set A should be used to translate the term in the fold node; this operation is performed by the B_σ function (§ 4.6).

Ensuring Finiteness

The partial evaluation of a program P is defined in terms of a finite set of terms A (Def. 3.2.11). It has already been noted that the set of terms stored in the non-leaf nodes of the m-tree corresponds to the set of terms A. Therefore, ensuring the finiteness of A requires ensuring the finiteness of the m-tree. This is performed by the global control of the algorithm, α ($\S$ 4.5).

Resultants are formed using non-trivial partial computations of a term t in P. Therefore, in the algorithm, it must be guaranteed that the computation of each restart term during the call to *specialise* terminates. This is the task of the local control of the algorithm ($\S$ 4.4). An ordering is imposed on the redexes of the computation in order to catch redexes that have already been evaluated during the computation. Since the sequence of redexes is guaranteed to be finite, all computations are also guaranteed to be finite.

Ensuring Closedness

Finally, in order to be correct in terms of the theorems established in Chapter 3, the algorithm must generate an A-closed residual program. Again, recall that the set of terms stored in the non-leaf nodes of the m-tree corresponds to the set A in the theoretical definition of partial evaluation (Def. 3.2.11). The functions *covered* and *split* are responsible for ensuring that all standard subterms in a term of the m-tree have a residual definition in the specialised program ($\S$ 4.3.2).

For example, if the condition of a conditional expression cannot be restarted by the *specialise* function, then either the *covered* function or the *split* function must ensure that the terms in the then- and else-branches are extracted from the expression. Otherwise, these terms will not have definitions in the residual program, and the residual program will not be correct with respect to the operational semantics of the original program.

4.8 Extensions of the Procedure

This section introduces several extensions of the constraint-based algorithm for the partial evaluation of rewriting-based functional logic programs which improve the efficiency of the residual programs.

4.8.1 Specialising Boolean Expressions

As noted in Section 4.5.4, removing outermost primitive functions from the terms in the unmarked leaf nodes of the m-tree has an adverse effect on the specialisation of Boolean expressions. By performing this operation, the Boolean function specialisation is similar to that of logic programming: residual definitions are defined for individual Boolean expressions, rather than their conjunction, disjunction, etc. Consider the following specialisation of the double-append benchmark according to the algorithm defined in this chapter.

Example 4.8.1 Consider the program P containing a definition of the list concatenation function (`Append`) as a Boolean function of three arguments:

```
FUNCTION  Append : List(a) * List(a) * List(a) -> Boolean.

Append([],y,z)    => z = y.
Append([h|t],y,z) => SOME[w](z = [h|w] & Append(t,y,w)).
```

Of course, this is not the most efficient way to implement list concatenation in a rewriting-based functional logic language, but this simple example should demonstrate some of the complexity of Boolean expression transformation. Consider the partial evaluation of P with respect to the term $t =$ `Append(x,y,z) & Append(z,u,v)`. The m-tree generated during this partial evaluation is shown in Figure 4.17.

The residual program resulting from this partial evaluation is exactly the same as the original program. Since the Boolean expression is divided by the global control, the interdependency between the terms is lost. Therefore, there is no specialisation that is possible by a constraint-based partial evaluator.

Therefore, greater specialisation is obtainable if the Boolean expressions are not divided during the global control operation. On the other hand, the problem with extending the partial evaluation procedure to undivided Boolean expressions is the commutativity of the operators. That is, the residual program may be less efficient than the original program, as the following example demonstrates.

Example 4.8.2 The following schema statements are included in the definition of the logical conjunction operator &:

```
x & False => False.
False & x => False.
x & True  => x.
True & x  => x.
```

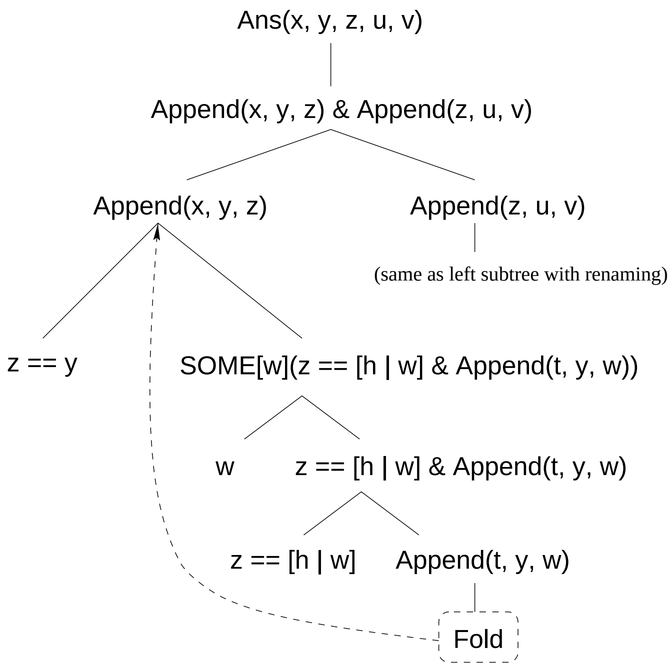

Ans(x, y, z, u, v)
Append(x, y, z) & Append(z, u, v)
Append(x, y, z)
Append(z, u, v)
(same as left subtree with renaming)
z == y
SOME[w](z == [h | w] & Append(t, y, w))
w
z == [h | w] & Append(t, y, w)
z == [h | w]
Append(t, y, w)
Fold

Figure 4.17: M-graph from the partial evaluation of the Boolean double-append example.

Consider the specialisation of the `Append` program from Example 4.4.1 with respect to the term `Append(x,y,z) & u`. Suppose the term was specialised without being divided by the *divide* function in the global control. Then, the residual program may be the program containing the following schema statements:

```
Ans(x, y, z, u) => FN_SP1(x, y, z, u).

FN_SP1([], y, z, u) => (z = y) & u.
FN_SP1([h_3 | t_4], y, z, u) =>
    SOME [w_7] (FN_SP1(t_4, y, w_7, ((z = [h_3 | w_7]) & u))).
```

Evaluation of the term `Append(x,y,z) & False` reduces in one computation step to `False` in the original program, but in the residual program, the translation `Ans(x,y,z,False)` reduces only to `FN_SP1(x,y,z,False)`. Therefore, the transformation has not preserved the operational semantics of the program. The correctness of the residual program can only be regained by instantiating `x` with a ground term. In this case, the residual program will require more reduction steps to reach the final term `False`, but the operational semantics are preserved.

Therefore, the specialisation of Boolean expressions has to be treated very carefully in order to preserve the correctness of the transformation or ensure residual programs that are more efficient than the original program.

Modifications to the Algorithm

In this section, the necessary modifications to the algorithm that allow the safe specialisation of Boolean expressions are presented. Basically, two operations need to be changed in order to allow advanced Boolean expression specialisation in the algorithm: the selection function $\mathcal{S}$ (§ 4.3.1) and the $divide$ function (§ 4.5.4). These functions are redefined below. First, the selection function $\mathcal{S}$ is modified to return g-functions occurring in the left-hand side of a Boolean expression.

Definition 4.8.3 $\mathcal{S}_B(t,P)$:
Given term t and program P, the selection function $\mathcal{S}_B(t,P)$ returns a selected term according to the following rules. Let o be a Boolean-typed primitive function.

$$\begin{array}{lcl}
\mathcal{S}_B(x,P) & = & \epsilon \\
\mathcal{S}_B(c(b_1,\ldots,b_n),P) & = & \epsilon \\
\mathcal{S}_B(c(b_1,\ldots,b_{i-1},t_i,t_{i+1},\ldots,t_n),P) & = & \mathcal{S}_B(t_i,P) \\
\mathcal{S}_B(o(n_1,n_2),P) & = & \mathcal{S}(n_1,P) \\
\mathcal{S}_B(p(t_1,\ldots,t_n),P) & = & \epsilon \\
\mathcal{S}_B(f(t_1,\ldots,t_n),P) & = & f(t_1,\ldots,t_n) \\
\mathcal{S}_B(g(t_0,t_1,\ldots,t_n),P) & = & \left\{\begin{array}{l} g(t_0,t_1,\ldots,t_n) \text{ if } match(t_0,g,P) \\ g(t_0,t_1,\ldots,t_n) \text{ if } \mathcal{S}(t_0,P)=\epsilon \\ \mathcal{S}_B(t_0,P), \text{ otherwise} \end{array}\right. \\
\mathcal{S}_B(\mathtt{LAMBDA}[x](t),P) & = & \epsilon \\
\mathcal{S}_B(\mathtt{ITE}(t_1,t_2,t_3),P) & = & \mathcal{S}(t_1,P) \\
\mathcal{S}_B(t_1\ t_2,P) & = & \mathcal{S}(t_1,P) \\
\mathcal{S}_B(\{x \mid t\},P) & = & \mathcal{S}(t,P)
\end{array}$$

where *match*(t_0,g,P) holds if for some $s \in TA(g,P)$, and some substitutions θ,ρ, $t_0\theta$ is syntactically equivalent to $s\rho$.

Then, the $divide$ function is redefined. The logical operators should not be stripped from the outside of terms *unless* one of the arguments is a passive term.

Definition 4.8.4 $div_B(t, C)$:
Let t be a term and C a set of constraints. Let o represent a Boolean-typed primitive function.

$$\begin{array}{lcll}
div_B(t,C) & = & \{(t_1, proj(t_1,C), \epsilon), \ldots (t_n, proj(t_n,C), \epsilon)\} & \text{if } t = c(t_1, \ldots, t_n) \\
div_B(t,C) & = & \{(t', proj(t',C), \epsilon)\} & \text{if } t = o(x,t') \mid o(t',x) \\
div_B(t,C) & = & \{(t_1, proj(t_1,C), \epsilon), \ldots (t_n, proj(t_n,C), \epsilon)\} & \text{if } t = p(t_1, \ldots, t_n) \\
div_B(t,C) & = & \{(t, proj(t,C), \epsilon)\} & \text{if } t = \mathtt{LAMBDA}[x](t) \\
div_B(t,C) & = & \{(t_1, proj(t_1,C), \epsilon), (t_2, proj(t_2,C), \epsilon) & \text{if } t = t_1\ t_2 \\
div_B(t,C) & = & \emptyset & \mathrm{otherwise}
\end{array}$$

These modifications are sufficient for the improved specialisation of Boolean expressions.

Generalising Boolean Expressions

The most specific safe generalisation (mssg) is particularly important when the specialisation of Boolean expressions is considered. Simply using the msg to generalise quantified expressions can result in a loss of the operational semantics of the original program.

The following rule defines bound variable elimination.

```
SOME [x1,...,xn] (x & (xi = u) & y)  =>
    SOME [x1,...,xi-1,xi+1,...,xn] (x{xi/u} & y{xi/u})
```

where `xi` is not free in `u`, and `u` is free for `xi` in `x` and `y`. Recall that the notation
`SOME [x_1,...x_n] e` is syntactic sugar for the term:

`Sigma(LAMBDA [x_1] (... LAMBDA[x_n](e) ...))`.

This schema statement in the definition of `Sigma` can cause the loss of terms in the program, as shown in the following example.

Example 4.8.5 Consider the following definition of the function `Split`.

```
FUNCTION   Split : List(a) * List(a) * List(a) -> Boolean.

Split([], x, y) => x = [] & y = [].
Split([h | t], x, y) =>
    (x = [] & y = [h | t]) \/
    (SOME[z](x = [h | z] & Split(t, z, y))).
```

Suppose the above program is partially evaluated with respect to the term `Split([1 | t], x, y)`. The following terms occur on a branch of the m-tree:

t_1: `SOME[z](x = [1 | z] & Split(t, z, y))`, and

t_2: `SOME[z1](x = [1, h1 | z1] & Split(t1, z1, y))`

Since $t_1 \trianglelefteq t_2$, it is necessary to generalise the term t_1. The msg of the terms t_1 and t_2 is the term t_3:

t_g: `SOME[u1](u2 = [u3 | u4] & Split(u5, u1, u7))`

The variable `u1` remains bound in the msg. However, the bound variables `z` and `z1` of the growing lists in the original terms are now generalised with a free variable `u4`.

If a value for `u1` is computed during a later Escher computation, all occurrences of variable `u1` will be eliminated, and the value will be lost. This will cause the computation of an incorrect residual program, since there is no way to propagate this computed value of `u1` to `u4`.

On the other hand, the most specific safe generalisation of the terms t_1 and t_2 is simply the term `Sigma(v)`, where v does not occur in the either term t_1 or t_2. Then, the lambda expression is divided during the abstraction step of the algorithm, and all variables of the innermost Boolean expression become free; thus, there is no possible loss of information by bound variable elimination.

Therefore, advanced Boolean specialisation is possible in the context of this algorithm, with the restriction that all arguments of the logical operators are non-variable terms.

4.8.2 Transforming Conditional Expressions

According to the definition in Section 4.3, the *extend* function, responsible for generating residual definitions of terms in the m-tree, handles a conditional expression in one of three ways.

1. If the conditional has a standard subterm that is selected according to the *selection function* (§ 4.3.1), then this term is restarted using the *specialise* function. If no such term exists in the condition of the conditional expression, or if a term is not selectable, then the conditional expression is passed to the *covered* function for evaluation (§ 4.3.2).

2. If the condition of the conditional expression can be represented by a constraint in one of the domains of the partial evaluator, the *covered* function extracts the constraint from the condition, and generates the two possible terms, corresponding to the `True` and `False` Boolean cases. If the condition is not representable by a constraint, the conditional expression is passed to the *split* function (§ 4.3.2).

3. Given a conditional expression `ITE(t1,t2,t3)`, the *split* function simply creates three leaf nodes on the current branch to store the immediate subterms of the conditional expression, `t1, t2` and `t3`.

As it is defined, *covered* can only extract the condition from an outermost conditional expression (Def. 4.3.8). This is a very basic approach to the specialisation of conditional expressions. Conditional expressions occur often in functional logic programming specialisation, particularly when partially evaluating functions defined over the integers. For example, consider the partial evaluation of the following example adapted from [Wad90][2].

Example 4.8.6 The following program performs the summation of a series of numbers from 1 to n.

```
FUNCTION  SumAll : Integer -> Integer.
SumAll(n) => Sum(UpTo(n)).

FUNCTION  Sum : List(Integer) -> Integer.
Sum([]) => 0.
Sum([x|xs]) => x + Sum(xs).

FUNCTION  UpTo : Integer -> List(Integer).
UpTo(n) =>
  IF    n < 1
  THEN  []
  ELSE  [n | UpTo(n-1)].
```

The aim of the program transformation in this case is to remove any unnecessary functions, such as the list construction created by the `UpTo` function, and removed by the `Sum` function (§ 5.3). Therefore, the program is partially evaluated with respect to the term `SumAll(n)`. On the first iteration of the algorithm, the m-tree shown in Figure 4.18 is generated.

Clearly, in the term `Sum(ITE(n < 1, [], [n|UpTo(n-1)]))`, the condition `n < 1` will have to be reduced to either `True` or `False` at run-time. Since it is evident that the condition

[2] This example is covered in more detail in Chapter 5.

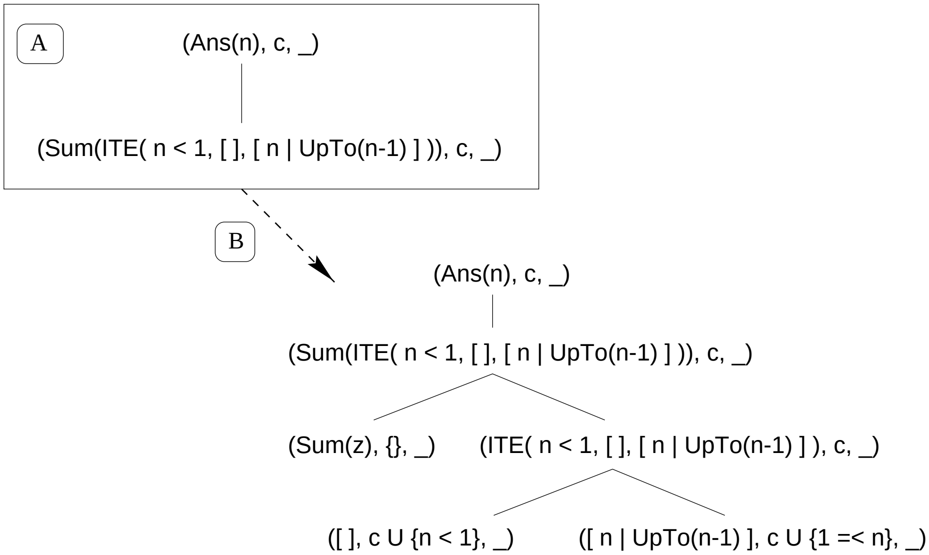

A
(Ans(n), c, _)
(Sum(ITE(n < 1, [ ], [n | UpTo(n-1)])), c, _)
B
(Ans(n), c, _)
(Sum(ITE(n < 1, [ ], [n | UpTo(n-1)])), c, _)
(Sum(z), {}, _)
(ITE(n < 1, [ ], [n | UpTo(n-1)]), c, _)
([ ], c U {n < 1}, _)
([n | UpTo(n-1)], c U {1 =< n}, _)

Figure 4.18: M-trees constructed during the partial evaluation of the summation program (Example 4.8.6). The m-tree marked with an "A" is the result of one iteration of the algorithm. Although the simple condition `n < 1` may be representable in the constraint domains of the partial evaluator, the *covered* cannot extract the condition, as the conditional expression is not outermost. Therefore, during the next iteration of the algorithm, marked "B", the term will be split, resulting in the loss of all possible specialisation.

is demanded by the computation, it is safe to extract the condition from the nested conditional expression. This requires the following revised definition of the *extend* function.

Definition 4.8.7 (*cond(t)*)
Let t be a term with demanded argument s. Then, *cond(t)* is defined as follows:

$$\begin{array}{lcll} cond(t) & = & \mathtt{ITE}(s, e[t_1], e[t_2]) & \text{if } t = e[\mathtt{ITE}(s, t_1, t_2)]; \\ cond(t) & = & t & \text{otherwise.} \end{array}$$

Then, the *extend* function can use the transformed term *cond(t)* in the call to the *covered* function, instead of t. The new function, called *extend*$_C$ is defined below.

Definition 4.8.8 $extend_C(\mu, P)$:
Given m-tree μ and program P, the m-tree $extend_C(\mu, P)$ is computed as follows.

```
$extend_C(\mu, P)$ = for all unmarked leaf nodes $L \in \mu$ :
    mark $L$ in $\mu$;
    $B = specialise((t_L, C_L), P)$;
    if $B \neq \emptyset$,
        for all $(r, C, s) \in B$, add leaf $L'$ to branch $\beta_L$ with label $(solve(rhs(r), C), C, s)$;
    otherwise, if $B = \emptyset$,
        $t_L$ = Split($t_L$);
        $B'$ = covered($cond(t_L), C_L, P$);
        for all $b \in B'$, add leaf $L'$ to branch $\beta_L$ with label $b$.
```

Example 4.8.9 Consider again the partial evaluation described in Example 4.8.6.

The m-tree shown in Figure 4.19 is constructed after two iterations of the partial evaluation algorithm with the $extend_C$ function. The nested conditional expression is allowed to be processed by *covered*, in order to obtain the terms in the leaves of the tree.

These terms will be generalised by the partial evaluator, but they will still be kept together, resulting in the elimination of the intermediate list constructor. A related specialisation example is presented in Section 5.3.

4.8.3 Generalising Expressions with Arithmetic Constraints

In the previous section, an extension of the algorithm was presented which extracts the demanded condition from a term (§ 4.8.2). There is a further extension possible for the specialisation of

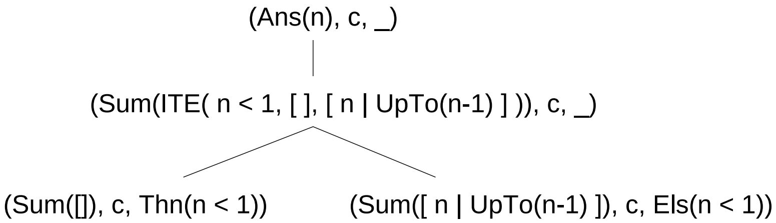

(Ans(n), c, _)
(Sum(ITE(n < 1, [ ], [n | UpTo(n-1)])), c, _)
(Sum([]), c, Thn(n < 1))
(Sum([n | UpTo(n-1)]), c, Els(n < 1))

Figure 4.19: The m-tree constructed after two iterations of the partial evaluator, using the extension for handling nested conditional expressions.

conditionals testing values of integer or natural numbers. Such a modification to the global control of the algorithm (§ 4.5) is formalised in this section.

As described in Section 4.5, the entailment of the constraints is tested before folding can occur. That is, if a term t is added to the branch β in a leaf node and if t is an instance of an ancestor stored on the branch, s, then the set of constraints must be tested to ensure that C_t entails C_s before folding can occur.

On the other hand, while C_t entails C_s, the constraints C_t may be describing more specific information about the term t. In some cases, it may be preferential to specialise the term t, rather than folding the term to the more general s. In particular, this occurs frequently during the processing of integers or natural numbers, as illustrated in the next example.

Example 4.8.10 Consider the following program to compute the summation of a series of positive numbers from i to 10.

```
FUNCTION  SumSeries : Integer -> Integer.
SumSeries(i) =>
   IF   i > 0
   THEN Sum(UpToTen(i))
   ELSE Sum(UpToTen(0)).

FUNCTION  Sum : List(Integer) -> Integer.
Sum([]) => 0.
Sum([x|xs]) => x + Sum(xs).

FUNCTION  UpToTen : Integer -> List(Integer).
UpToTen(i) =>
  IF    i =< 10
  THEN  [i | UpToTen(i+1)]
  ELSE  [i].
```

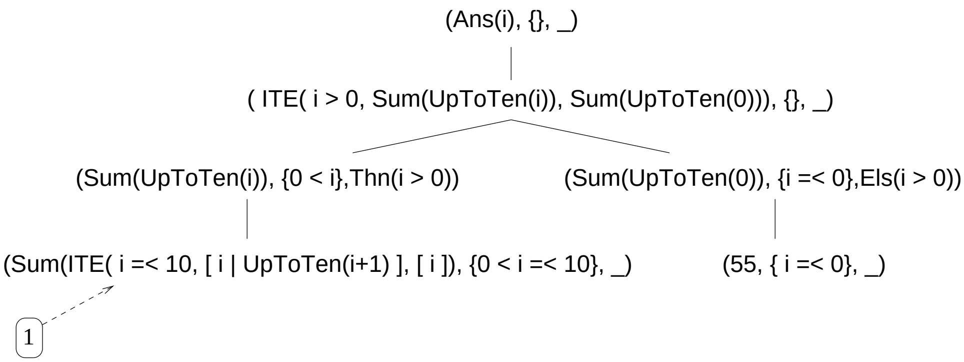

(Ans(i), {}, _)
(ITE(i > 0, Sum(UpToTen(i)), Sum(UpToTen(0))), {}, _)
(Sum(UpToTen(i)), {0 < i},Thn(i > 0))
(Sum(UpToTen(0)), {i =< 0},Els(i > 0))
(Sum(ITE(i =< 10, [i | UpToTen(i+1)], [i]), {0 < i =< 10}, _)
(55, { i =< 0}, _)
1

Figure 4.20: An m-tree constructed during the partial evaluation of the summation program with respect to the term `SumSeries(i)`. Note that the term marked with the “1” will be generalised by the global control of the algorithm, as it looks like it is a growing term on the branch.

Partial evaluation of this program with respect to the term `SumSeries(i)` results in the m-tree shown in Figure 4.20.

The partial evaluator will identify the term marked by the “1” in Figure 4.20 as a growing term and will try to generalise the term. On the other hand, it is preferential to unfold this term further to obtain the residual program shown below:

```
FUNCTION   Ans : Integer -> Integer.
Ans(i) =>
  IF  i > 0  THEN  FN1(i)  ELSE  55.

FUNCTION  FN1 : Integer -> Integer.
FN1(i) =>
  IF    i =< 10
  THEN  i + FN1(i+1)
  ELSE  i.
```

The key to achieving quality specialisation in these cases is to identify that the addition of the condition to the constraint set will generate a finite interval for the variable `i`. Therefore, even though the term in question contains a recursive call to `UpToTen`, its argument has been bound finitely by the addition of the constraint `i =< 10` to the set {`0 < i`}.

On the other hand, this extra unfolding step is only permitted when the transition from unbound to a bound interval for all variables in the expression. Since the interval does not uniquely identify a branch of the conditional that can be eliminated by the transformation, both branches must remain in the program. Therefore, if this process was allowed to continue as the interval is strictly decreasing, the program would grow in size quite dramatically. This may have a negative effect on the quality of the residual program.

This extension of the algorithm is related to bounded static variation [GJ96] and abstract interpretation based on interval analysis [JBE94].

4.8.4 Annotating Function Variables

The algorithm for the constraint-based partial evaluation of rewriting-based functional programs specialises arbitrary higher-order programs. However, there is very little specialisation that is possible for a function variable if it is not instantiated. In this section, an extension to the algorithm is proposed for the advanced processing of function variables. This involves annotating the function variables with their run-time types and restricting their run-time instances to function names.

Example 4.8.11 Consider the following rewriting-based functional logic program.

```
FUNCTION  Map : List(a) * (a -> b) -> List(b).
Map([], f) => [].
Map([h|t], f) => [f(h) | Map(t,f)].

FUNCTION  Abs : Integer -> Integer.
Abs(int) => IF int > 0 THEN int ELSE (-1 * int).

FUNCTION  Times10 : Integer -> Integer.
Times10(int) => int * 10.

FUNCTION  Sum : List(Integer) -> Integer.
Sum([]) => 0.
Sum([x|xs]) => x + Sum(xs).
```

Suppose the program is partially evaluated with respect to `Sum(Map([1,3,5,7,9],f))`. The best specialisation that is available in this case using the basic partial evaluation procedure is the following residual definition for `Map` in place of the definition in the original program:

```
FUNCTION  SumMap : (Integer -> Integer) -> Integer.
SumMap(f) => Sum([f(1),f(3),f(5),f(7),f(9)]).

FUNCTION  Abs : Integer -> Integer.
Abs(int) => IF int > 0 THEN int ELSE (-1 * int).

FUNCTION  Times10 : Integer -> Integer.
Times10(int) => int * 10.
```

```
FUNCTION  Sum : List(Integer) -> Integer.
Sum([]) => 0.
Sum([x|xs]) => x + Sum(xs).
```

On the other hand, if the function variable `f` is restricted to run-time values of function names only (i.e. no lambda expressions), then the type declarations of the functions in the program can be used to provide further specialisation of the program. For example, using the type declaration of the `Sum` function in the program above, it is possible to infer the type of the function `f`: `Integer -> Integer`. There are two functions that have a suitable type declaration in the original program: `Abs` and `Times10`. If these functions are applied to the term `Sum(Map([1,3,5,7,9],f))`, the following residual definition may be generated.

```
FUNCTION  SumMap : (Integer -> Integer) -> Integer.

SumMap(Abs)     => 25.
SumMap(Times10) => 250.
```

Function variables in the term which satisfy the restriction are annotated with a “ˆˆ” identifier. Before the algorithm enters its *extend*-generalise iterations, the possible values for the annotated function variables are determined based on their inferred types (or types supplied by the user). If a type cannot be inferred, the annotation is simply removed.

Particularly for large programs, it may be the case that this step can generate many possible instantiations of the original term. In order to avoid an unfeasibly large program, a bound b is placed on the number of terms that can be generated by this step. If the cardinality of the set of instantiated terms is less than b, then the terms are added as leaf nodes to the root of the m-tree at the beginning of the specialisation. Otherwise, the root is left unchanged, and the constraint-based partial evaluation proceeds as in Definition 4.1.2.

Definition 4.8.12 (*funcs*(t, P))
Let P be a program, $t = e[f]$ be a term with an annotated function variable f of type $A \to B$, where A, B are not type variables. Generate the set of all possible run-time instantiations of f, *funcs*(t, P) as follows:

$$funcs(t, P) = \{e[H] \,|\, H : A \to B \in P\}$$

where the cardinality of $funcs(t, P)$ is less than b.

Closure analysis for functional language specialisation has been studied extensively; this is a related method for computing an approximation of the set of functions to which an expression $\mathtt{e}_1$ in an application $\mathtt{e}_1$ $\mathtt{e}_2$ could evaluate [Ses88b].

4.8.5 Handling Non-linear Expressions

Most specialisation algorithms for functional programs restrict the transformation of non-linear expressions in order to guarantee an improvement in efficiency for the residual program. This restriction prevents duplicating computations during specialisation as in the following example from [Ses88a, Sør96].

Example 4.8.13 Consider the following program to convert lists into trees.

```
FUNCTION  ListToTree : List(a) -> Tree(a).
ListToTree([])     => Leaf.
ListToTree([x|xs]) => Br(ListToTree(xs)).

FUNCTION  Br : Tree(a) -> Tree(a).
Br(x) => Branch(x,x).
```

Partial evaluation of this program with respect to the term `ListToTree(x)` will result in the residual program containing the following statements:

```
FUNCTION  Ans : List(a) -> Tree(a).
Ans(x)         => FN_SP1(x).

FUNCTION  FN_SP1 : List(a) -> Tree(a).
FN_SP1([])     => Leaf.
FN_SP1([x|xs]) => Branch(FN_SP1(xs),FN_SP1(xs)).
```

Depending on the implementation of the functional logic language, the residual program is either the same or less efficient than the original program. In a lazy implementation of the rewriting-based language, the interpreter would maintain several references to the call to `ListToTree`, thus avoiding the duplicate computation of this term [Wad90]. This *sharing* is lost in the specialisation, and thus, the residual program is less efficient.

A post-processing transformation of the residual program can re-introduce the local definitions necessary to restore the efficiency of the residual program. This analysis examines all terms in the bodies of the residual program and adds the local definitions as necessary. For example, for the *E* language, the analysis defined below checks the terms from the outermost for duplication. In this case, assume that the infix function `WHERE` is a built-in function of *E*.

Definition 4.8.14 (Non-linearity transformation)
Let P be a residual program. Let t be a body of a schema statement in P. If there exist several occurrences of the subterm s in t, call these terms $s_1, \ldots s_n$, such that for all $1 \leq i, j \leq n$, s_i does not occur in s_j, then replace the body t with t' WHERE $z = s$ in P, where t' is equivalent to t with the occurrences of the terms $s_1, \ldots s_n$ are each replaced by the variable z not occurring in t.

The post-processing transformation results by repeatedly applying this transformation to all terms occurring in the bodies of the statements in the residual program until the program stabilises.

Example 4.8.15 The efficiency of the residual program is regained by introducing the local definition in the third statement of the program:

```
Ans(x)          => FN_SP1(x).
FN_SP1([])      => Leaf.
FN_SP1([x|xs]) => Branch(y,y) WHERE y = FN_SP1(xs).
```

The danger of duplicating computations by unfolding non-linear expressions has been addressed in many different ways. In the development of Mix, a static analysis called duplication analysis identifies the non-linear arguments of a term and restricts the unfolding of the term to its linear subterms [Ses88a]. Non-linearity is not a problem in basic deforestation, since linearity is a condition of treeless programs [Wad90]. In positive supercompilation, the non-linear arguments are generalised using let-expressions as above [Sør96].

4.9 Discussion

In this section, the algorithm for constraint-based partial evaluation is related to established techniques for the specialisation of declarative programs. The structure of the algorithm shares features with program specialisation techniques for both functional and logic languages. This is understandable as the language itself is composed of features from both languages.

The “generate and control” structure of the algorithm is the foundation of most automatic program transformation techniques. The algorithm is inspired by that of partial deduction [GB91], but the general framework is evident in partial evaluation, positive supercompilation, deforestation, and narrowing-based partial evaluation as well [JGS93, SGJ96, Wad90, AFV96a]. Unfolding in positive supercompilation and deforestation only occurs in single steps; therefore, the control of the algorithm is performed with each new term encountered during the transformation. On the other hand, the control for logic and functional logic program specialisation is two-tiered; one to control the unfolding, and one to control the set A. This will be discussed in further detail below.

The constraint-based procedure described in this thesis uses an m-tree structure to record the set of terms for which residual definitions have been generated during the computation. As noted in Section 4.2, Martens and Gallagher identified the positive effect that explicitly noting the relationships between terms of the set A had on the polyvariance in the residual program [LM96]. Similarly, Turchin uses a tree structure, called a graph of states, to record the terms generated during the driving step in supercompilation [Sør96]. This was the forerunner of process trees, structures used in supercompilation [GK93] and positive supercompilation [SGJ96] to record all the terms encountered during the transformation. M-trees, on the other hand, do not record all the terms encountered during the partial evaluation, but only store in the non-leaf nodes the terms for which residual definitions have been generated.

The algorithm for the partial evaluation described in this chapter uses constraints to represent additional information about a term. These constraints are used to prune branches of conditionals that are unreachable, to eliminate restart terms generated during *specialise*, and to further the unfolding of a term when the interpreter cannot evaluate a conditional expression. Very few automatic constraint-based techniques exist for program specialisation. As noted in Section 2.2, both generalized partial computation and supercompilation use constraints or predicates to represent negative information. In the case of supercompilation, the restricted REFAL language only requires restrictions [GK93]. Generalized partial computation uses user-defined libraries and disunification to solve constraints in an automatic transformation procedure [Tak91]. Positive supercompilation [SGJ96], narrowing-based positive supercompilation[AFV96a] and partial deduction [Kom81] all propagate positive information only by unification.

As noted earlier, the algorithm has the "generate-control" framework shared with many other automatic specialisers. On the other hand, since the unfolding is performed by the interpreter without direct intervention by the specialiser, control must be imposed on the unfolding to ensure its finiteness. This forms the local/global control distinction for procedures using finite unfolding to reduce terms during partial evaluation. Such a distinction does not exist for techniques such as positive supercompilation [SGJ96] or deforestation [Wad90], since they are single step transformations of the code.

Controlling unfolding during partial deduction has been studied extensively; solutions range from deterministic unfolding [GB91] to strategies using well-founded orderings [Mar94] or well-quasi orderings [SG95, LM96] over the terms. In this algorithm, the homeomorphic embedding well-quasi ordering is used to ensure finite partial computations, although the redexes of the terms are checked, not the terms themselves.

The global control presented in this chapter was inspired by the formulation of generalisation in supercompilation [SG95] and positive supercompilation. Features of the rewriting-based functional logic language necessitated the development of the most specific safe generalisation. Although the `LAMBDA` function should be regarded as “just another function” in the language, it is often not safe to use the msg of two lambda expressions in the partial evaluation of a program. The global control described in this thesis also handled the set of constraints associated with each term, either by testing the entailment of the sets or by widening the sets if generalisation of the terms is necessary.

The $divide$ and *split* function perform different operations in the algorithm: $divide$ strips the outermost function from the term when the outermost function cannot be restarted by *specialise* and *split* removes subterms of the term when they cannot be restarted by *specialise*. In techniques such as positive supercompilation [SG95], a splitting function performs both tasks in the generalisation step. Separating these operations allows the least disruption of the term overall.

The definition of the metric for selecting ancestors on a branch of the m-tree is also unique to this algorithm. This has a profound effect on the result of the specialisation, as will be shown for examples in Sections 5.6 and 5.3. This area needs to be studied further, as the global control is the most expensive operation during the partial evaluation. This hinders the application of the partial evaluator to practical program specialisation.

Recent work in conjunctive partial deduction [GJMS96, Leu97a] and narrowing-based partial evaluation [AAF$^+$98] have formulated the global control of the algorithm in terms of the best partition of the conjunction. Such a method for global control may be implemented to operate along with the extension proposed in 4.8.1.

4.10 Summary

This chapter began with the formalisation of the algorithm for the constraint-based partial evaluation of rewriting-based functional programs. The algorithm can be coarsely divided into two operations: an operation to extend and expand a tree of all possible computations of the term t in the program P, and a mechanism for ensuring the termination of the partial evaluator. The methodology was described for extracting the test from a conditional or case expression and representing this information by constraints. The constraints are used both in the generation of the restart terms for terms in the tree and during the generalisation of terms in the global control. Furthermore, the constraints associated with a term allow the elimination of unreachable branches in the computation.

The local control of the algorithm, ensuring finite unfolding during transformation, uses a well-quasi-ordering over the terms of the language to stop potentially non-terminating computations. The method of supplementing this ordering to avoid prematurely ending computations was presented. The global control uses a similar technique to ensure the branches of the tree structure are finite. This means that only finitely many restart steps will be performed, and the specialiser will terminate and return a residual program from the computation.

A function for extracting the program from the tree structure used in the algorithm was defined. Furthermore, the algorithm is correct in terms of the theoretical framework for the partial evaluation of functional logic programs from Chapter 3. Finally, several extensions to the algorithm were discussed, including improving the specialisation of Boolean expressions, avoiding non-linear expressions in the residual program, and transforming conditional expressions to further improve the efficiency of the residual program.

Chapter 5

Experiments with Constraint-based Partial Evaluation

This chapter contains an introduction to the implementation of the constraint-based partial evaluator for Escher programs and presents some experimental results on benchmark programs. The chapter begins with an overview of the Escher partial evaluator in Section 5.1. In Sections 5.2 to 5.7, the constraint-based partial evaluation of several example programs is examined in detail. The residual programs constructed in each case are compared with those generated from established specialisation techniques for declarative programs (§ 2.1.3, 2.1.4). In each section, key extensions of the algorithm are demonstrated, and the effect of program structure on the quality of the partial evaluation is discussed.

5.1 Implementation of the Constraint-based Partial Evaluator

In this section, features of the implementation of the Escher constraint-based partial evaluator are presented.

The implementation follows the structure of the algorithm as defined in Chapter 4. The system was developed for the implementation of Escher written in the Gödel logic programming language. The partial evaluator specialises arbitrary Escher programs with respect to arbitrary Escher terms. Features of the implementation include:

- An algorithm for constraint-based partial evaluation, including the local and global control

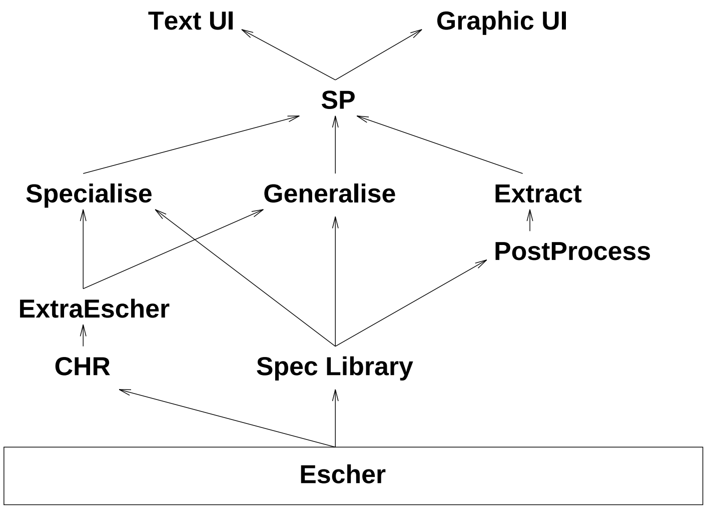


Figure 5.1: The architecture of the Escher constraint-based partial evaluator.

operations;

- Constraint solving for constraints of the union of the Herbrand, Boolean, and linear arithmetic domains;
- A choice of specialisation technique, including the advanced Boolean expression handling as presented in Section 4.8.1;
- A choice of ancestor selection, as discussed in Section 4.5.3;
- Post-processing operations;
- A choice of text-based interface or graphical user interface.

Given a compiled Escher program P, term t, and set of constraints C, the constraint-based partial evaluator returns a file containing the residual program P_t and the final m-tree, which can be displayed graphically using the GNU tool `xvcg` [San95a].

The architecture of the system is illustrated in Figure 5.1. The system is divided into ten modules. All the modules except the graphical user interface are implemented in the Gödel logic programming language. Descriptions of the functionality of the main system modules are given below.

SP Top level module of the system. The display predicates are defined in this module.

Specialise Procedures for the *extend, specialise*, *covered* and *split* functions (§ 4.3).

Constraints Constraint solving operations and predicates to call the relevant constraint solvers.

Generalise Procedures for the global control of the algorithm (§ 4.5).

Extract Procedures for the renaming of terms and the extraction of the residual program from the m-tree (§ 4.6).

CHR Module containing constraint handling rules, written as Escher schema statements, for simplifying and propagating constraints.

ExtraEscher Procedures for extending Escher computations with the constraint handling rules of the CHR module.

Spec Library Basic predicates required by all modules of the system.

Further description of the implementation, including the constraint solving operations of the implementation and an example session, are presented in the following sections.

5.1.1 General Overview

This section contains a brief introduction to the main components of the implementation. As shown in the previous section, the modular architecture of the system reflects the three main operations of the algorithm (Def. 4.1). Implementations of the *extend* function, the α operator, and the $\mathcal{R}_\sigma$ function form the three top-level predicates of the partial evaluator. In this section, the primary data structures and the general structure of the implementation are presented.

As defined in Section 4.1, the constraint-based partial evaluation algorithm manipulates a tree structure which stores the set of terms A and the residual statements generated during the specialisation. In the implementation, the data structure representing the m-tree is defined as follows. Unmarked leaves of the m-tree are represented using the function `LF`, with the following type:

```
FUNCTION  LF : Integer * Term * Integer * List(Term) * Trace -> Tree.
```

The expression `LF(n,term,index,hrb,chTree)` packages the following information in a node of the m-tree:

`n`**:** The index of the leaf node, `Integer` type.

`term`: The term stored in the leaf node, `Term` type. This type is defined by the Escher implementation.

`index`: The highest variable index of `term`, type `Integer`. This index is required by the Escher interpreter to ensure the introduction of new variables.

`hrb`: The constraints associated with the term `term`, type `List(Term)`. The constraints are kept as elements in a list, in order to keep the constraints of different types separate. That is, for this implementation, this argument is a list of three elements, storing the Herbrand constraints, arithmetic constraints, and Boolean constraints respectively.

`chTree`: The set of characteristic paths of the restart terms for the term `term`, type `Trace`. Characteristic paths have been used to improve the precision of global control in partial deduction [GB91, LM96]. Similarly, the indices of the partial computation are returned by the specialiser. As in the method of [LM96], this additional information can be supplied to the abstraction operator in order to improve the polyvariance of the residual programs.

Marked nodes (both leaf and non-leaf nodes) of the m-tree are packaged using the `B` function of the type:

```
FUNCTION  B : Mark * Integer * Term * Integer * List(Term) *
              Trace * List(Tree) -> Tree.
```

A marked node of the m-tree `B(mk,n,term,index,hrb,chTree,lvs)` shares most of its information with its associated `LF` node. That is, the literals `n`, `term`, `index`, `constraints`, and `chTree` are unchanged by the operations on the m-tree. The two elements introduced in the conversion from an unmarked node to a marked node are:

`mk`: The "type" of marked node, type `Mark`. This allows us to indicate if the node is a *Split* (§ 4.3.2), *Fold* (§ 4.5.1), or *Gen* (§ 4.5.2) node. If the node of the m-tree is simply a marked node, the mark `mk` of the node `Mk(bind)` stores the substitutions generated by *specialise* to generate the associated restart terms for `term` (§ 4.3.1).

`lvs`: The immediate children of the node, type `List(Tree)`. If the node is a marked leaf node, this is an empty list.

Therefore, the m-tree is composed of marked and unmarked nodes, until the partial evaluator terminates, at which point all nodes of the m-tree are marked. The definitions of the original program are read from the compiled Escher program and stored in a table data structure [HL94], indexed by the function name and arity. The iteration between the *extend* function and the α operator (implemented by the `generalise` predicate below) is the predicate `PE`. A counter `ch` indicates the number of unmarked leaf nodes that were added to the m-tree in the last iteration. If this counter is zero, the tree has been unchanged on the previous iteration, and the partial evaluation returns the current m-tree (§ 4.1), as indicated in the first clause of `PE`.

```
PREDICATE   PE :  Integer * Tree * Table(List(Term)) * List(Integer) *
                  Integer * Tree.

PE(0, tree, _, _, _, tree).

PE(ch, tree, program, env, n, final_tree) <-
  ch > 0 &
  Extend(tree, n, program, env, added_tree, n2, change) &
  Generalise(change, added_tree, n2, env, new_tree, n3, new_ch) &
  PE(new_ch, new_tree, program, env, n3, final_tree).
```

The argument `n` is the minimal unused m-tree node index and the argument `env` records the current settings of the partial evaluator. That is, the extensions of the algorithm presented in Section 4.8 can be invoked depending on the environment specified by the user.

The predicate `Extend` is the top-level predicate of the `Specialise` module. This predicate is the implementation of the *extend* function from the algorithm, responsible for extending and expanding the m-tree by restarting the Escher computations (§ 4.3). Its definition, from the `Specialise` module, is shown below.

```
Extend(LF(n,tm,k,c,p), nn, _, _, newNode, nn+1, 1)<-
    NonTrivComp(tm, k, tm2, k2) |
    Solve(tm2, k2, c, tm3) &
    BuildNode(LF(n,tm,k,c,p), Mk([]), [LF(nn,tm3,k2,c,p)], newNode).

Extend(LF(n,tm,k,c,p), nn, pgm, [app|_], newNode, nn2, ch)<-
    Restart(tm, app, k, pgm, bnd, mgu, ite) |
    Specialised(mgu, ite, bnd, LF(n,tm,k,c,p), nn, newNode, nn2, ch).

Extend(LF(n,tm,k,c,p), nn, _, _, newNode, nn2, ch)<- |
    SplitTerm(LF(n,tm,k,c,p), 1, nn, nds, lvs, nn2, ch) &
    BuildNode(LF(n,tm,k,c,p), Mk(nds), lvs, newNode).

Extend(B(mk,n,t,k,c,p,lfs), nn, pgm, env, B(mk,n,t,k,c,p,lfs2),
  nn2, ch)<- |
    Extend_lvs(lfs, nn, stdtms, env, lfs2, nn2, ch).
```

The first clause in the definition of `Extend` handles terms in the set A that have non-trivial computations in the program P. For these terms, it is not necessary to generate the set of associated restart terms (§ 3.2). Therefore, the result of the computation of the term `tm` in the program is computed (`tm2`), and the *solve* function (§ 4.3.1) is applied to `tm2` to obtain the label for the new leaf node, `LF(nn,tm3,k2,c,p)`.

If the term stored in the unmarked leaf node has a trivial computation, the set of restart terms must be generated. This case is implemented in the second clause of `Extend`. The predicate `Restart` computes the restart terms of the term `tm`. The minimal substitutions for generating the restart terms of `tm` are returned in the argument `bnd`. The `Specialised` predicate uses these substitutions to generate the associated residual statements. Otherwise, if the term has a conditional expression that must be processed by the *covered* function, this conditional is returned in the `ite` argument, and the constraint solving is invoked in `Specialised`. In this case, the `Restart` predicate returns the conditional subexpression in order to prevent having to search through the term twice: once for the selected term (§ 4.3.1) and once for the conditional subexpression for the *covered* function (§ 4.3.2).

Finally, if no restart terms can be generated for `tm` (see Section 4.3.2), the term `tm` is simply split by means of the `SplitTerm` predicate. This operation results from the third clause in the definition of `Extend` above. The last clause processes the marked nodes of the m-tree; these nodes are passed over in order to reach the unmarked leaf nodes of the m-tree.

The `Generalise` predicate occurring in the definition of the `PE` predicate above is the implementation of the global control of the algorithm. The predicates responsible for the global control comprise the `Generalise` module of the Escher partial evaluator. The main functionality of the `Generalise` module is contained in the `Genz` predicate, shown below.

```
Genz(leaf, bnch, nn, [app|env], tree, final, nn3, 1)<-
    SplitTerm(leaf, app, nn, nodes, lvs, nn2, _) |
    BuildNode(leaf, Mk(nodes), lvs, new_leaf) &
    ChangeTree(tree, new_leaf, bnch, _, new_tree) &
    Genz(new_leaf, bnch, nn2, [app|env], new_tree, final, nn3, _).

Genz(LF(n,term,k,c,p), bnch, nn, env, tree, final, nn3, 1)<-
    FoldTerm(term, bnch, m, sub) |
    Sub2Leaves(sub, k, c, nn, nn2, l, lvs) &
    new_node = B(Fd(m,l),n,term,k,c,p,lvs) &
    FoldTree(tree, m, new_node, bnch, tree2) &
    Genz(new_node, bnch, nn2, env, tree2, final, nn3, _).

Genz(LF(n,tm,k,c,p), bnch, nn, [app,gen|env], tree, final, nn3, 1)<-
    Embeds(tm, k, p, bnch, Lbl(m,p2,tm2,c2), msg, km, sub) |
    Sub2Leaves(sub, km, c, nn, nn2, nds, lvs) &
    ChooseLeaf(LF(n,tm,k,c,p), Lbl(m,p2,tm2,c2), leaf) &
    BuildNode(leaf, Gn(nds), [LF(nn2,msg,km,c2,p2)|lvs], node) &
    ChangeTree(tree, node, bnch, path2, tree2) &
    NewBranch(tm, tm2, [root | path2], bnch, newBnch) &
    Genz(node, newBnch, nn2+1, [app,gen|env], tree2, final, nn3, _).

Genz(LF(_,_,_,_,_), _, nn, _, tree, tree, nn, 0)<- |.
```

The `Genz` predicate alternates between the $divide$, $fold$, and gen functions of the global control (§ 4.5). The clauses for `Genz` shown above only manipulate the unmarked leaves of the m-tree. The marked nodes of the m-tree are added to the representation of the branch `bnch` by `Genz`.

The first clause shown above splits a term stored in an unmarked leaf node according to the $divide$ function (§ 4.5.4). The second clause of `Genz` processes the term in the unmarked leaf node if it is an instance of an ancestor in the branch. This is the top-level implementation of the $fold$ function of the global control (§ 4.5.1). The child nodes of the folded node are generated by the call to the predicate `Sub2Leaves`, as the terms of the instance substitution `sub` are stored in the new leaf nodes `lvs`.

The third clause shown above implements the gen function (§ 4.5.2). The predicate `Embeds` is true if the term in the unmarked leaf node violates the term ordering relation on the branch. If such a violation occurs, the most specific safe generalisation (§ 4.5.2) is computed and returned in the argument `msg`. Again, the terms of the substitution are stored in the new unmarked leaves of the branch; these are generated by the call to `Sub2Leaves`. Depending on whether the generalisation is a variable or a non-variable term, part of the branch will be deleted or not. The predicate `ChooseLeaf` returns the node which is the new *Gen* node. This node is used to form the new branch via `NewBranch` and change the m-tree accordingly via `ChangeTree`.

In this section, the top-level predicates for manipulating the primary data structure of the implementation have been presented. The constraint solving functionality of the Escher constraint-based partial evaluator is covered in the next section. Otherwise, a discussion of particular implementation details is beyond the scope of this thesis.

5.1.2 Constraint Solving

The Escher partial evaluator employs constraint solving to improve the decision-making of the specialiser. This section introduces the constraint solving features of the implementation. Constraint handling rules were a natural addition to the Escher implementation; some example rules are given later in this section. Initial implementations of the partial evaluator used constraint handling rules written in Escher for basic constraint solving; constraint handling rules are described briefly below. A widening operation is necessary for the global control of the algorithm (§ 4.5). The widening operators for the three constraint domains are discussed below.

Constraint Handling Rules

Constraint handling rules (CHRs) are a high-level language for building, extending, and combining constraint solvers [Frü94, FB95]. CHRs allow the *simplification* of user-defined constraints and the *introduction* of new, possibly redundant, constraints. A third type of rule, called *simpagation* rules, expresses subsumption and relative simplification. CHRs permit the rapid prototyping of constraint solvers; solvers for the standard constraint domains can be implemented using CHRs, although there is a penalty in terms of speed and complexity [FB95].

The Escher language was extended with some defined CHRs in the implementation of the partial evaluator. CHRs are guarded rules with multiple head atoms. The flexibility of the Escher language permitted the straightforward implementation of these rules. For example, the following rule simplifies the unsatisfiable conjunction of Herbrand constraints:

```
x & (v ~== w) & y & (v == w) & z => False
```

In terms of the linear arithmetic constraint domain, the following CHR replaces the two inequalities with a simpler equality constraint:

```
y & (x =< a) & w & (a =< x) & z => y & w & (x = a) & z
```

The following is an example of a *propagation* CHR, which introduces new constraints to the conjunction:

```
y & (x =< a) & w & (a =< b) & z =>
    y & (x =< a) & (a =< b) & w & (x =< b) & z
```

A check must be performed in this case to ensure the constraint `a =< b` does not already exist in the conjunction, or non-termination may result. Although the new constraint is logically redundant, it may permit the simplification of other constraints in the set (by one of the constraint simplification rules as shown above).

A CHR performing relative simplification is defined below:

```
y & (x < a) & w & (x =< a) & z => y & w & (x =< a) & z
```

Essentially, constraint handling rules express determinate information about predicates of the domains [Frü94]. The conjunctions in the heads of the statements are required to detect unsatisfiability of a conjunction of constraints. The CHRs provide a method for extending the predicate definitions of Escher with simple constraint solving capability.

Widening Operators

The global control of the algorithm ensures the finiteness of the resulting m-tree ($\S$ 4.5). A widening operation is required to approximate the set of constraints during the generalisation step ($\S$ 4.5.2).

The widening operator $\bigtriangledown$ is a mapping from $D \times D$ to D defined as follows [CC77, CH78]:

$$\forall C, C' \in D : C \Rightarrow^{E} C \bigtriangledown C'$$
$$\forall C, C' \in D : C' \Rightarrow^{E} C \bigtriangledown C'$$

where $C_1 \Rightarrow^{E} C_2$ if the set of constraints C_1 entails the set C_2.

The widening operators for the Herbrand and the linear arithmetic constraint domains are presented in this section. Widening of Boolean constraints is not necessary, as the Boolean constraint domain is finite.

There is a possibility of non-termination of the algorithm if arbitrary Herbrand constraints are permitted. Herbrand constraints can be approximated by either imposing an arbitrary depth-bound [CC77] or computing the most specific generalisation (msg) of the terms. For example, approximating the constraint `x == [a,b,c,d]` using a depth-2 bound would result in the constraint `SOME[z](x == [a,b|z])`, where `z` is a variable not occurring in the constraint. In this way, the infinite domain of terms is approximated by a finite domain, thus ensuring the finite convergence of all sequences in this domain.

Widening of linear arithmetic constraints is based on the convex polyhedron described by a system of linear equalities and inequalities, called linear restraints [CH78]. A polyhedron C, a subset of n-dimensional space, is *convex* if for any two vectors $x_1, x_2 \in C$ and $\lambda \in [0, 1]$, $\lambda x_1 + (1 - \lambda) x_2 \in C$. A finite system of linear restraints can be represented geometrically as a closed convex polyhedra. This polyhedron can be described using three sets: a set of points, a set of lines, and a set of rays.

For two systems of linear inequalities, S_1 and S_2, the approximation $S_1 \bigtriangledown S_2$ is defined as the convex polyhedron consisting in the linear restraints of Q_1 verified by every element of Q_2 [CH78]. In other words, for two sets of linear restraints, $S_1 = \{\beta_1, \ldots, \beta_n\}$ and $S_2 = \{\gamma_1, \ldots, \gamma_m\}$ with associated polyhedra H_1 and H_2, $H_1 \bigtriangledown H_2 = S'_1 \bigcup S'_2$, for S'_1, S'_2 defined as follows:

- For all $\beta_i \in S_1$, $\beta_i \in S'_1$ if it is satisfied by all points of H_2.
- For all $\gamma_j \in S_2$, $\gamma_j \in S'_2$ if there exists a $\beta_k \in S_1$ such that $((S_1 - \{\beta_k\}) \cup \{\gamma_j\})$ defines the same polyhedra as H_1.

In [Sağ98], an equivalent version of the widening operation was defined. This version of widening permitted a simpler method for testing the equality of two polyhedra.

The following example of the widening operation is from [CH78]:

Example 5.1.1 Let S_1 be the set of linear restraints: $\{-x_1 + 2x_2 \leq -2, x_1 + 2x_2 \leq 6, x_2 \geq 0\}$. Let S_2 be the set of linear restraints: $\{-x_1 + 2x_2 \leq -2, x_1 + 2x_2 \leq 10, x_2 \geq 0\}$. Then $S_1 \bigtriangledown S_2$ is the set: $\{-x_1 + 2x_2 \leq -2, x_2 \geq 0\}$. The constraint $x_1 + 2x_2 \leq 6$ is not satisfied by all points of H_2, the convex polyhedron corresponding to S_2 (e.g. $(10, 0)$). Likewise, the constraint $x_1 + 2x_2 \leq 10$ cannot be an element of the set S'_2 as defined above, since the convex polyhedra are not equal.

5.1.3 Example Session

In this section, an example session using the partial evaluator with the text-based interface is presented. In order to maintain consistency throughout this work, the example specialisation in this section is the partial evaluation of the double-append benchmark, using the definition of `Concat` from Example 3.2.5.

Loading Programs and the Partial Evaluator

As noted earlier, the system requires pre-compiled Escher programs. The compilation of an Escher program is straightforward; the Escher program is compiled into a Gödel module called `User`. This Gödel module is invoked by the system upon loading the Escher partial evaluator. Clearly, each Escher program must be compiled in this way before it can be specialised.

Suppose the Escher program has the name “M3”. This program is compiled into the Gödel module `User.loc`, which must also be compiled. The command `;l Sp.` loads the Escher partial evaluator. The initial system interaction is shown below.

```
[Special] <- Compile("M3").
Yes
[Special] <- ;c User.
Reading file "User.exp" ...
Reading file "User.loc" ...
Loading the language of module "Kernel" ...
Parsing module "User" ...
Compiling module "User" ...
Module "User" compiled.
[Special] <- ;l Sp.
Loading module "Sp" ...
Loading module "Special" ...
Loading module "Extract" ...
Loading module "SpecLib" ...
Loading module "Escher" ...
Loading module "Compile" ...
Loading module "SPCompile" ...
...
Loading module "User" ...
Loading module "Specialise" ...
Loading module "Constraints" ...
Loading module "ExtraEscher" ...
Loading module "CHRs" ...
Loading module "PostProcess" ...
Loading module "Generalise" ...
```

The modules loaded by the system are the modules comprising the Escher system and those making up the Escher partial evaluator (including the `User` module containing the target program).

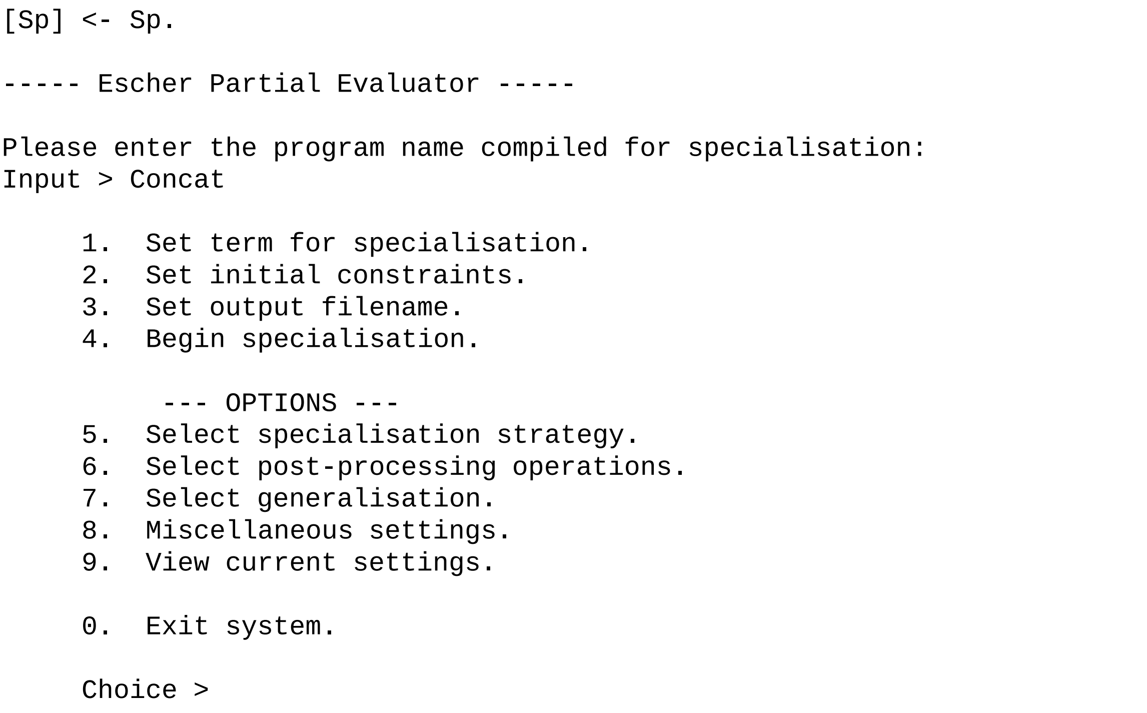
Starting the Partial Evaluation

The Escher constraint-based partial evaluator is invoked by the query `Sp` (or `SP`). This results in the menu of options shown in Figure 5.2.

Selecting the first option allows the user to fix the term of the partial evaluation. For example, during the specialisation of the double-append example, the user sets the term as follows:

```
     Choice >1

Please enter the term for specialisation:
Input >Concat(Concat(x,y),z)
```



```
[Sp] <- Sp.

----- Escher Partial Evaluator -----

Please enter the program name compiled for specialisation:
Input > Concat

     1.  Set term for specialisation.
     2.  Set initial constraints.
     3.  Set output filename.
     4.  Begin specialisation.

          --- OPTIONS ---
     5.  Select specialisation strategy.
     6.  Select post-processing operations.
     7.  Select generalisation.
     8.  Miscellaneous settings.
     9.  View current settings.

     0.  Exit system.

     Choice >
```

Figure 5.2: The main menu of the Escher constraint-based partial evaluator.

```
You entered "Concat(Concat(x,y),z)".
Is this correct? (1 = yes, 0 = no)
(1 or 0) ?1
```

Similarly, menu options 2 and 3 allow the user to define the initial set of constraints and the output filename. In this case, the initial constraint set is the empty set, and the output filename is “SPConcat”.

By selecting menu option 4, the partial evaluation begins. After the computation has completed, the system returns an m-tree and the associated residual program. In this case, since the partial evaluation of the double-append example is relatively simple, the partial evaluator returns the result quickly.

```
Time: 70 ms
Writing tree to SPConcat...
@@@@@@@@@@@ PROGRAM @@@@@@@@@@@@:
Ans(x, y, z) =>
  FN_SP1(x, y, z).

FN_SP1([], y, z) =>
  FN_SP2(y, z).

FN_SP1([u_3 | x_4], y, z) =>
  [u_3 | FN_SP1(x_4, y, z)].

FN_SP2([], z) => z.

FN_SP2([u_5 | x_6], z) =>
  [u_5 | FN_SP2(x_6, z)].
```

The m-tree returned by the partial evaluator is in the form readable by the GNU tool, `xvcg` [San95a]. The m-graph for the double append example as viewed using this tool is shown in Figure 5.3.

The Optional Settings

The remainder of the menu options are optional switches. For example, menu option 5 offers the users the possibility of changing the specialisation approach:

```
     Choice >5
```

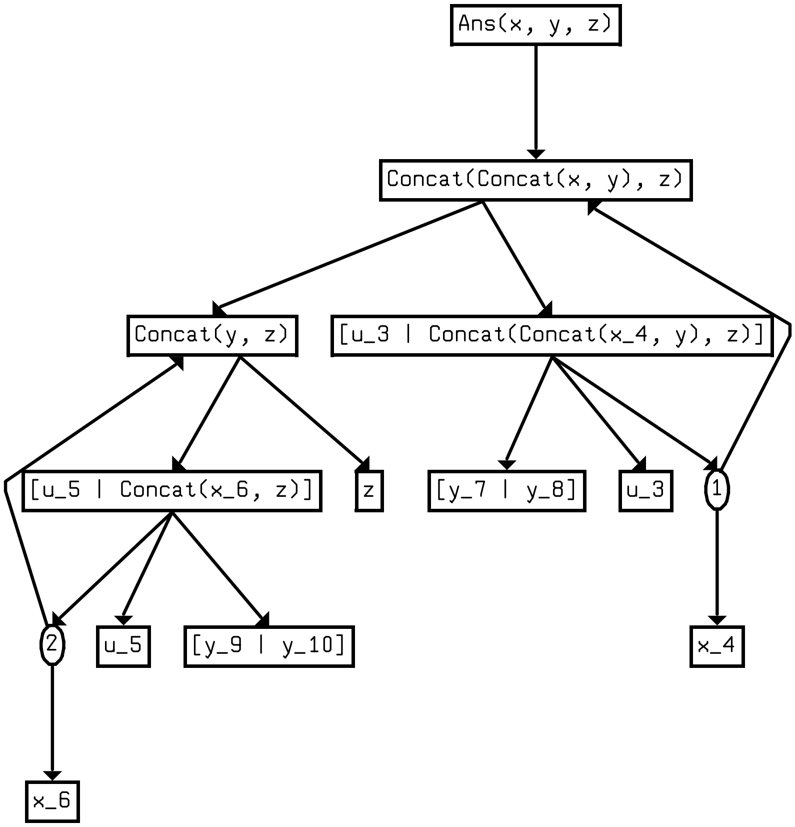


Figure 5.3: The m-tree returned by the Escher constraint-based partial evaluator, as viewed using the GNU tool, xvcg.

```
Options:
1.  Basic specialisation.
2.  Advanced specialisation of Boolean expressions.
3.  Restricted function variable specialisation.
```

As discussed in Section 4.8.1, it is possible to alter the algorithm in order to obtain better specialisation of Boolean expressions (Option 5.2). Likewise, by restricting function variables to be instantiated solely with function names at run-time (§ 4.8.4), better specialisation of higher-order expressions can be achieved by selecting Option 5.3.

Menu option 6 allows the user to select the type of post-processing performed by the specialiser.

```
     Choice >6

Options:
0.  No post-processing.
1.  Post-unfolding.
2.  Optimisation of conditional statements.
3.  Local definition introduction.
4.  All post-processing operations.

Choice >
```

Three main operations are offered: Option 6.1, traditional post-unfolding (§ 4.6); Option 6.2, optimisation of conditional statements (§ 4.8.2); and Option 6.3, local definition introduction (§ 4.8.5). The termination of the post-unfolding operation is guaranteed by restricting the unfolding to f-functions that appear only twice in the program, not occurring in the same schema statement.

Menu option 7 allows the user to change the ancestor selection strategy for folding or generalising a branch of the m-tree (§ 4.5.3).

```
     Choice >7

Options:
1.  Bottom-up generalisation.
2.  Top-down generalisation.
3.  Advanced selection metric.
```

There are three alternatives: the ancestor nearest the unmarked leaf node can be chosen (bottom-up, Option 7.1), the ancestor nearest the root node can be chosen (top-down, Option 7.2), or the selection metric can be used to determine the “best” ancestor for folding or generalisation (Option

7.3). The default setting is the bottom-up approach. Examples in Sections 5.3 and 5.6 below show how the ancestor selection approach can affect the result of the partial evaluation.

Miscellaneous options are available by selecting menu option 8:

```
     Choice >8

Options:
1.  Turn m-tree display on
2.  Turn size metric optimisation on
3.  Turn characteristic trees on
0.  Cancel.
```

Option 8.1 allows the user to view the m-tree that is constructed on each *extend-generalise* iteration during the program transformation (§ 4.1). These m-trees are displayed in vcg format.

The second option introduces an optimisation of the partial evaluator using the size metric (§ 2.3.4). As noted earlier (§ 4.4), the homeomorphic embedding relation requires depth-wise searching through each term on a branch to note if an embedding has occurred. For practical-sized terms, this is expensive to compute. In order to improve the efficiency of the specialiser, the simple property of the homeomorphic embedding relation is incorporated into the algorithm:

Lemma 5.1.2 For all terms t_1, t_2, if $t_1 \trianglelefteq t_2$, then $|t_1| \leq |t_2|$.

Proof By cases using the definition of the homeomorphic embedding relation. □

If this option is selected, the sizes of terms are stored in the nodes of the m-tree. Then, the gen function applies the contrapositive of Lemma 5.1.2: if $|t_1| > |t_2|$, then $t_1 \ntrianglelefteq t_2$.

As discussed in Section 5.1.1, a method for improving global control of partial deduction using characteristic trees was originally formalised in [GB91] and extended in [LM96] to characteristic atoms. In this case, the user has the option to incorporate this trace information into the generalisation operation. In the basic implementation, if this option is chosen, terms are generalised if the ordering on the branch is violated and the two terms have the same characteristic trees.

Finally, the selection of menu option 9 simply allows the user to view the current settings of the program specialiser. For the double-append example, the partial evaluator is set as follows.

```
     Choice >9

The current settings are:
```

```
Program : Concat
Term : Concat(Concat(x,y),z)
Constraints: None
Output file : SPConcat
Specialisation strategy : 1
Post-processing : 3
Generalisation : Top down
M-tree display : Off
Size metric optimisation : Off
Trace terms : Off
```

The graphical-user interface offers the same functionality as the text-based interface with the minor addition of a parser to transform the terms into the required syntax of the Escher system.

This presentation of the operation of the Escher constraint-based partial evaluator completes the discussion of the system implementation.

5.2 Specialisation of a Pattern Matcher

The *KMP-test* is a measure of the quality of the residual programs generated by a program specialiser. It tests whether the partial evaluator can generate the well-known, efficient Knuth-Morris-Pratt algorithm [KMP77] from an inefficient, simple pattern matching program.

In this section, the constraint-based partial evaluator on this benchmark is shown to pass the KMP-test for certain implementations of the naive pattern matching program. That is, the efficient program is constructed from the inefficient program by partial evaluation with respect to a fixed pattern. The constraint-based partial evaluator performs this transformation for both a simple tail-recursive pattern matcher and a pattern matcher using nested calls. On the other hand, the specialiser does not pass the KMP-test given a simple pattern matching program using Boolean expressions.

5.2.1 Specialising a Tail-recursive Pattern Matcher

A naive tail-recursive pattern matching program is presented in Figure 5.4. It has a multiplicative time complexity of $\mathcal{O}(|p||s|)$, where p is the input pattern and s is the input string. The aim is to generate a KMP pattern matching program with complexity $\mathcal{O}(|s|)$ by specialising this program with respect to a fixed pattern.

Partially evaluating the tail recursive pattern matching program (Figure 5.4) with respect to the term

```
FUNCTION  Match : List(a) * List(a) -> Boolean.

Match(p, s) => Loop(p, s, p, s).

FUNCTION  Loop : List(a) * List(a) * List(a) * List(a) -> Boolean.

Loop([], [], op, os) =>  True.
Loop([], [s | ss], op, os) =>  True.
Loop([p | pp], [], op, os) =>  False.
Loop([p | pp], [s | ss], op, os) =>
  IF    p = s
  THEN  Loop(pp, ss, op, os)
  ELSE  Next(op, os).

FUNCTION  Next : List(a) * List(a) -> Boolean.

Next(op, []) =>  False.
Next(op, [s | ss]) =>
  Loop(op, ss, op, ss).
```

Figure 5.4: Tail-recursive naive pattern matching program.

`Match([1,1,2], u)`, representing a fixed pattern “1”, “1”, “2”, results in the residual program shown in Figure 5.5. In this example, the function names automatically generated by the specialiser have been replaced with descriptive identifiers.

It can be shown that this is a KMP pattern matching program; the specialised pattern matcher never has to go “backwards” to test elements of the list previous to the one currently being tested. In this case, the constraint-based information propagation is essential for achieving this result. For example, in the second schema statement defining the function `Loop_12` (Figure 5.5), if the test `s_16 == 1` fails, the program starts the matching with the *next* element of the list, instead of just removing the head of the string and beginning the matching process again. This is a result of propagating the negative constraint `s_16 ~== 1` during the partial evaluation. In Figure 5.6, the expression in the dashed box has the outermost conditional testing the value of `s_16`. The set of constraints resulting from adding the constraint `s_16 == 1` to the current set is unsatisfiable. Therefore, only the term in the *else*-branch of the conditional is added to the m-tree, and the redundant test on `s_16` is removed by the specialisation.

```
FUNCTION  Match : List(Integer) -> Boolean.

Match_112(u) =>
    Loop_112(u).

FUNCTION  Loop_112 : List(Integer) -> Boolean.

Loop_112([]) => False.
Loop_112([s_9 | ss_10]) =>
    IF    s_9 = 1
    THEN  Loop_12(ss_10, s_9)
    ELSE  Loop_112(ss_10).

FUNCTION  Loop_12 : List(Integer) * Integer -> Boolean.

Loop_12([], s_9) => False.
Loop_12([s_16 | ss_17], s_9) =>
    IF    s_16 = 1
    THEN  Loop_2(ss_17, s_9, s_16)
    ELSE  Loop_112(ss_17).

FUNCTION  Loop_2 : List(Integer) * Integer * Integer -> Boolean.

Loop_2([], s_9, s_16) => False.
Loop_2([s_24 | ss_25], s_9, s_16) =>
    IF    s_24 = 2
    THEN  True
    ELSE  Loop_12([s_24 | ss_25], s_16).
```

Figure 5.5: The residual program generated by partially evaluating the naive pattern matching program wrt the pattern `[1,1,2]`. Passing the negative information as constraints results in an optimised version of the `Loop_12` function.

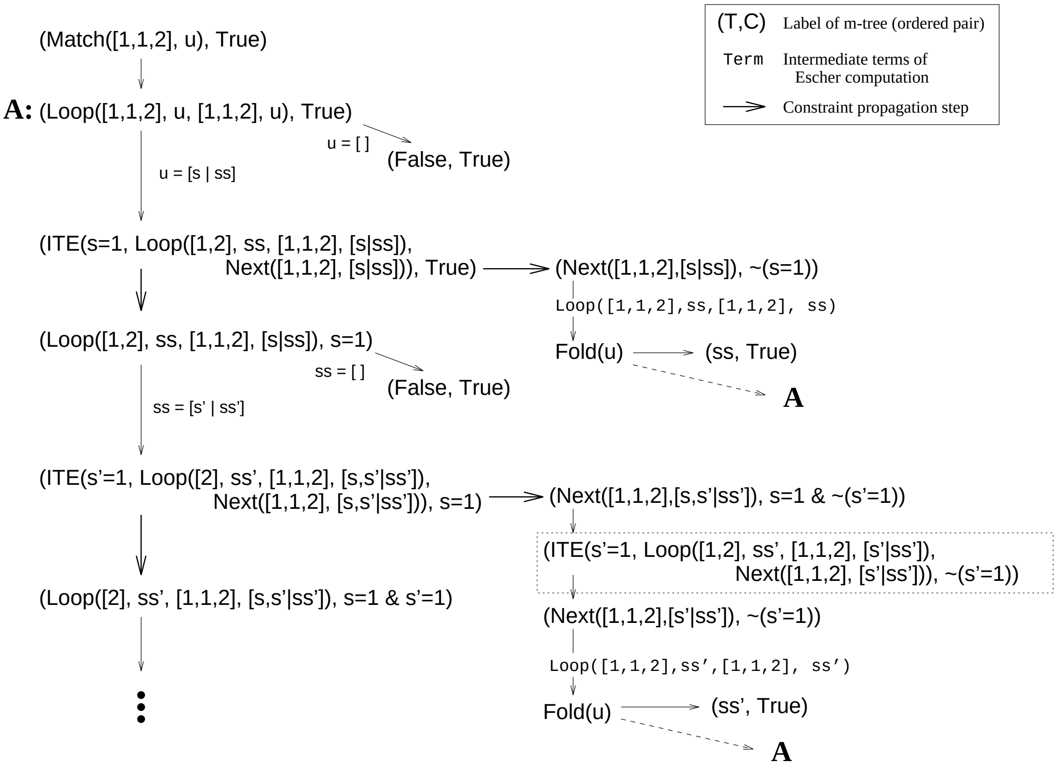

(T,C) Label of m-tree (ordered pair)
Term Intermediate terms of Escher computation
Constraint propagation step
(Match([1,1,2], u), True)
A: (Loop([1,1,2], u, [1,1,2], u), True)
u = [ ]
(False, True)
u = [s | ss]
(ITE(s=1, Loop([1,2], ss, [1,1,2], [s|ss]),
Next([1,1,2], [s|ss])), True)
(Next([1,1,2],[s|ss]), ~(s=1))
Loop([1,1,2],ss,[1,1,2], ss)
Fold(u)
(ss, True)
A
(Loop([1,2], ss, [1,1,2], [s|ss]), s=1)
ss = [ ]
(False, True)
ss = [s' | ss']
(ITE(s'=1, Loop([2], ss', [1,1,2], [s,s'|ss']),
Next([1,1,2], [s,s'|ss'])), s=1)
(Next([1,1,2],[s,s'|ss']), s=1 & ~(s'=1))
(ITE(s'=1, Loop([1,2], ss', [1,1,2], [s'|ss']),
Next([1,1,2], [s'|ss'])), ~(s'=1))
(Loop([2], ss', [1,1,2], [s,s'|ss']), s=1 & s'=1)
(Next([1,1,2],[s'|ss']), ~(s'=1))
Loop([1,1,2],ss',[1,1,2], ss')
Fold(u)
(ss', True)
A

Figure 5.6: Section of the m-tree for the specialisation of the naive pattern matching program wrt `Match([1,1,2], u)`. Note the use of negative information propagation at the `IF-THEN-ELSE` term in the dotted box; this leads to the removal of the unnecessary test in `Loop_12`, as described in Section 4. The definition of `Loop_112` in the residual program is extracted from the node marked "A".

```
FUNCTION  Match : List(a) * List(a) -> Boolean.
Match(p,s) =>
    IF    Prefix(p,s)
    THEN  True
    ELSE  Next(s,p).

FUNCTION  Next : List(a) * List(a) -> Boolean.
Next([],p)     => False.
Next([s|ss],p) => Match(p,ss).

FUNCTION  Prefix : List(a) * List(a) -> Boolean.
Prefix([],ss)     => True.
Prefix([p|ps],ss) => Loop(ss, p, ps).

FUNCTION  Loop : List(a) * a * List(a) -> Boolean.
Loop([], p, ps) => False.
Loop([s|ss], p, ps) =>
    IF    p = s
    THEN  Prefix(ps,ss)
    ELSE  False.
```

Figure 5.7: General naive pattern matching program.

5.2.2 Specialising a General Pattern Matcher

In the previous section, it was shown that the constraint-based partial evaluator can pass the KMP-test given a tail-recursive pattern matcher. On the other hand, there are many ways to implement a naive pattern matching program in Escher. Functional logic languages allow the user to write a program in either a logic or a functional style; furthermore, the expressive nature of the language offers even more flexibility. In this section, a study of the effect of the program structure on the quality of the residual program in this case is undertaken by repeating the KMP-test for other implementations of the naive pattern matching program.

The program shown in Figure 5.7 is also a simple pattern matching program [Sør96]. Nested conditional expressions are used to determine if the pattern `p` is a substring of the string `s`.

The call to `Prefix` generates an expression with nested conditional statements. For example, using the program of Figure 5.7, the restart term `Match([1,2],[s|ss])` simplifies to the term

```
  IF    (IF 1 = s THEN Prefix([2],ss) ELSE False)
  THEN  True
  ELSE  Next([s|ss],[1,2])
```

Using the basic algorithm of Chapter 4, this term will be split into its immediate subterms, since the condition of the outermost conditional expression cannot be represented in any of the constraint domains of the partial evaluator. This results in minimal specialisation of the program, and the residual program fails the KMP-test.

On the other hand, extending the algorithm with the procedure to handle nested conditional expressions as described in Section 4.8.2, the partial evaluator generates a residual program that passes the KMP-test. That is, the above expression is not split during the partial evaluation. Instead, the constraint `1 = s` is extracted, and the two subterms of the innermost conditional expression are used to evaluate the outermost conditional statement. In this case, the residual program is identical to the program in Figure 5.5.

One can argue that the programs in Figures 5.4 and 5.7 are written in a functional style, and this style is not necessarily one that would be used in a functional *logic* programming setting. Unfortunately, the residual program resulting from specialising the following “logical” implementation of the general pattern matcher does not pass the KMP-test.

```
FUNCTION  Match : List(a) * List(a) -> Boolean.

Match(p,s) =>
    Prefix(p,s) \/ Next(s,p).

FUNCTION  Next : List(a) * List(a) -> Boolean.

Next([],p)     => False.
Next([s|ss],p) => Match(p,ss).

FUNCTION  Prefix : List(a) * List(a) -> Boolean.

Prefix([],ss)     => True.
Prefix([p|ps],ss) =>
    SOME[r,rs](ss = [r|rs] &
              ((p = r & Prefix(ps,rs)) \/ (p ˜= r & False))).
```

The residual program generated by the partial evaluator extended with the advanced specialisation of Boolean expressions is shown below.

```
FUNCTION  Ans : List(Integer) -> Boolean.
```

```
Ans(x) =>
    (SOME [rs_8] (x = [1,1,2 | rs_8])) \/ FN_SP4(x).

FUNCTION  FN_SP4 : List(Integer) -> Boolean.

FN_SP4([]) => False.

FN_SP4([s_11|ss_12]) =>
    (SOME [rs_16] (ss_12 = [1,1,2 | rs_16])) \/ FN_SP4(ss_12).
```

The Boolean expression `s = 1` cannot be added to the constraint store, since the variable is destroyed after the substitution within the Boolean expression is performed. Therefore, the negative constraint cannot be used to specialise the `Next` predicate. Instead, the pattern is simply unfolded in the call to `Prefix` in the residual program. Since the specialised matcher still requires a "backwards step", it is not a KMP pattern matching program.

5.2.3 Comparison with Other Techniques

The KMP-test, specialisation of a naive pattern matching program to obtain a Knuth-Morris-Pratt matcher, is a popular example for demonstrating the power of program transformation schemes.

Program specialisation techniques that pass the KMP-test include generalised partial computation [FN88], and supercompilation [Leu95]. In fact, any technique that allows the propagation of the negative binding information will pass the KMP-test. For generalised partial computation and supercompilation, this information is represented explicitly.

In addition, partial deduction [Smi91], partial evaluation of narrowing-based functional logic programs [AFJV97], and partial evaluation based on Mix pass the KMP-test [CD89]. For example, specialising the tail-recursive pattern matching program using the SP partial deduction system [Gal91], the residual program generated given the fixed pattern `[a,a,b]`is the KMP-matcher shown below.

```
match([a,a,b],[X1|X2]) :- match1_1(X1,X2).

match1_1(X1,[X2|X3]) :- a \== X1, match1_1(X2,X3).
match1_1(a,[X1|X2]) :- match1_2(X1,X2).

match1_2(X1,[X2|X3]) :- a \== X1, match1_1(X2,X3).
match1_2(a,[X1|X2]) :- match1_3(X1,X2).

match1_3(X1,X2) :- b \== X1, match1_2(X1,X2).
match1_3(b,X1) :- true.
```

The partial evaluation of narrowing-based functional logic programs passes the KMP-test in a similar manner. Moreover, by extending partial deduction with more general unfold/fold rules, it is possible to transform the following specification of a pattern matcher into a KMP pattern matcher [PPR97].

```
match(P,S) :- append(S1,_,S),append(_,P,S1).
```

Generally, partial deduction of the above program achieves some speedup, but does not obtain a KMP pattern matcher.

The Mix partial evaluator can generate a KMP-matcher, but a binding time improvement is necessary to change the tail-recursive pattern matching program to the less simple matcher shown below [JGS93, CD89].

```
KMP(p,d) => Loop(p,d,p,[],[]).

Loop([],d,pp,f,ff)     => True.
Loop([p|ps],d,pp,f,ff) =>
    IF    f = []
    THEN  (IF   d = []
           THEN  False
           ELSE  (IF   p = Head(d)
                  THEN  Loop(ps,Tail(d),pp,[],Concat(ff,[p]))
                  ELSE  (IF   ff = []
                         THEN  KMP(pp,Tail(d))
                         ELSE  Loop(pp,d,pp,Tail(ff),Tail(ff)))))
    ELSE  (IF   p = Head(f)
           THEN  Loop(ps,d,pp,Tail(f),ff)
           ELSE  Loop(pp,d,pp,Tail(ff),Tail(ff))).
```

The Mix partial evaluator does not pass the KMP-test for the tail-recursive pattern matcher of Section 5.2.1. However, after the program has been “improved” to the program above, a KMP matcher can be generated by this technique.

5.3 Elimination of Intermediate Data Structures

Throughout this thesis so far, the simple double-append example has been used to illustrate the techniques and theory behind the partial evaluation procedure for rewriting-based functional logic programs. Not only is the double-append example simple, but it also allows the demonstration of the quality of residual programs produced by the specialisation procedure in question. In particular, the double-append example illustrates the ability of the partial evaluator to eliminate intermediate data structures (see Example 3.2.12).

Several example programs which are optimised by the automatic elimination of intermediate data structures are presented in this section. Since the constraint-based partial evaluation procedure is an automatic procedure, it is not possible to guarantee that *every* intermediate list which can be eliminated by deforestation can be removed by this procedure. For some examples, the control of the algorithm is too conservative to permit the complete elimination of intermediate data structures. A summary of the speed-up and size of the residual programs for the examples in this section is included in Table 5.4.

5.3.1 Example: Summing Squares

An example of the elimination of intermediate data structures by constraint-based partial evaluation is the specialisation of the following program, which uses an intermediate list for the computation of the sum of squares of a sequence of numbers. In this example, taken from [Wad90], the original program is partially evaluated with respect to the term `SumSqUpTo(n)`.

The result of specialising the SumSquares program (Figure 5.8) with respect to `SumSqUpTo(n)` depends on the ancestor selection strategy selected in the algorithm (see Section 4.5.3). After two iterations of the algorithm (§ 4.1), the partial evaluator has constructed the m-tree shown in Figure 5.9.

The α operator is applied to the m-tree of Figure 5.9. Let β be the right branch of the m-tree. There are four stored in nodes of β:

t_1 : `Ans(n)`

t_2 : `Sum(Squares(ITE(1 > n, [], [1 | UpTo(1+1,n)])))`

t_3 : `Sum(Squares([1 | UpTo(1+1,n)]))`

t_4 : `1+Sum(Squares(ITE(2 > n, [], [(1+1) | UpTo((1+1)+1,n)])))`

First, the α operator will split the term t_4 into three new unmarked leaf nodes, storing the terms:

t_5 : `z_1 + z_2`

```
FUNCTION   SumSqUpTo : Integer -> Integer.
SumSqUpTo(n) =>
  Sum(Squares(UpTo(1, n))).

FUNCTION   Sum : List(Integer) -> Integer.
Sum([]) => 0.
Sum([n | ns]) => n +  Sum(ns).

FUNCTION   Sq : Integer -> Integer.
Sq(n) => n * n.

FUNCTION   Squares : List(Integer) -> List(Integer).
Squares([]) => [].
Squares([n | ns]) => [Sq(n) |  Squares(ns)].

FUNCTION   UpTo : Integer -> List(Integer).
UpTo(m, n) =>
  IF   m > n
  THEN []
  ELSE [m | UpTo(m+1, n)].
```

Figure 5.8: The SumSquares benchmark program.

t_6 : `1`

t_7 : `Sum(Squares(ITE(2 > n, [], [1+1 | UpTo(1+1,n)])))`

If the numerical terms in t_2, t_3, and t_7 are written in successor notation, it is clear that the term t_7 embeds both the terms t_2 and t_3. On the other hand, these numerical constants have type `Integer`; in general, a conversion to successor notation will not be possible or practical. The definition of strict homeomorphic embedding is extended to handle integer constants by adding the following rule to Definition 4.4.4:

$i_1 \trianglelefteq i_2$ if $|i_1| \leq |i_2|$ for integers i_1, i_2.

The relation $|i_1| \leq |i_2|$ is a well-quasi relation on the set of integers (§ 4.4). Selecting the ancestor closest to the unmarked leaf node in the branch (bottom-up selection) results in the following residual program, P_{BU}.

```
FUNCTION  Ans : Integer -> Integer.
```

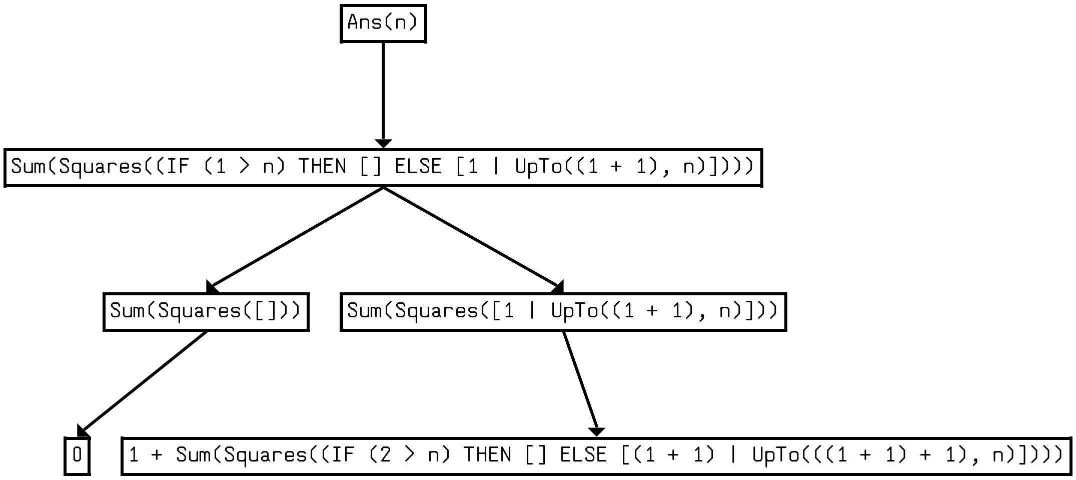

Ans(n)
Sum(Squares((IF (1 > n) THEN [] ELSE [1 | UpTo((1 + 1), n)])))
Sum(Squares([]))
Sum(Squares([1 | UpTo((1 + 1), n)]))
0
1 + Sum(Squares((IF (2 > n) THEN [] ELSE [(1 + 1) | UpTo(((1 + 1) + 1), n)])))

Figure 5.9: The m-tree returned after two applications of the *extend* function. Note that the term stored in the leaf node of the right branch of the m-tree embeds *two* terms stored in the ancestor nodes in the branch.

```
Ans(n) =>
    IF    1 > n
    THEN  0
    ELSE  FN_SP11([1 | FN_SP29(1, n)]).

FUNCTION  FN_SP11 : List(Integer) -> Integer.
FN_SP11([]) => 0.
FN_SP11([n_5 | ns_6]) => (n_5 * n_5) + FN_SP11(ns_6).

FUNCTION  FN_SP29 : Integer * Integer -> List(Integer).
FN_SP29(x_8, x_9) =>
    IF    (x_8 + 1) > x_9
    THEN  []
    ELSE  [(x_8 + 1) | FN_SP29((x_8 + 1), x_9)].
```

In this case, the intermediate data structure has not been removed by partial evaluation. The msg of terms t_3 and t_7 is the term `Sum(Squares(x))`. This term is renamed by the partial evaluator to `FN_SP11(x)`, and its residual definition is included in the program. The term `FN_SP29(m,n)` is a renaming of the term `UpTo(m+1,n)`.

On the other hand, using the metric defined in Section 4.5.3, the constraint-based partial evaluator returns the following program, P_{TD}, on completion.

```
FUNCTION  Ans : Integer -> Integer.
```

```
Ans(n) => FN_SP12(1, n, 1).

FUNCTION  FN_SP12 : Integer * Integer * Integer -> Integer.

FN_SP12(x_6, x_7, x_8) =>
    IF x_6 > x_7
    THEN 0
    ELSE (x_8 * x_8) + FN_SP12((x_8 + 1), x_7, (x_8 + 1)).
```

Clearly, the same residual program results if the ancestor closest to the root node is selected (the top-down approach of Section 4.5.3). Despite the lack of static data, the intermediate lists constructed during computations in the original program have been optimised away in the residual program. In this case, the msg is computed using the terms t_2 and t_7. The residual statement defining the function FN_SP12 is the resultant generated from the term Sum(Squares(UpTo(x,n))) during the partial evaluation. This result is not possible without the extension to handle nested conditional expressions in the *covered* function (§ 4.8.2). The condition must be extracted from the term

Sum(Squares(ITE(x_4 > x_5, [], [x_6 | UpTo(x_6+1,x_5)]))).

Otherwise, the conditional expression will be extracted by the *split* function (§ 4.3.2). The partial evaluator would return a residual program similar to P_{BU}.

There is still some inefficiency in the residual program P_{TD}, resulting from the repeated arguments of FN_SP12. As shown in Figure 5.9, the condition of the expression is simplified in Escher while the terms of the branches of the conditionals are not evaluated until a branch can be selected. Therefore, the leaf node of β has a condition 2 > n, while the subterm in the else-branch of the conditional contains the expressions 1+1. This causes the repeated argument in the residual program. These redundant arguments may be eliminated by post-processing analysis [LS96].

Furthermore, for the residual program P_{TD}, the post-processing introduction of local definitions (§ 4.8.5) should be applied in order to ensure the efficiency of this residual program. For example, in the schema statement for the function FN_SP12, the repeated variable x_8 should be removed from the main body of the statement and referenced in a local definition.

5.3.2 Example: Double-flip

In this example, a program containing a function for "flipping" a binary tree is partially evaluated in order to remove unnecessary data structures. This example is from [Wad90]. By partially evaluating the program with respect to two calls of the Flip function, the residual program should simply return the binary tree passed as input at run-time. The original flip function is shown below.

```
FUNCTION  Leaf : a  -> Tree(a);
          Branch : Tree(a) * a * Tree(a) -> Tree(a).

FUNCTION  Flip : Tree(a) -> Tree(a).

Flip(Leaf(x))       => Leaf(x).
Flip(Branch(x,i,y)) => Branch(Flip(y),i,Flip(x)).
```

Specialising the above program with respect to the term `Flip(Flip(x))` using the constraint-based partial evaluator results in the following program.

```
FUNCTION   Ans : Tree(a) -> Tree(a).

Ans(x) => FN_SP1(x).

FUNCTION   FN_SP1 : Tree(a) -> Tree(a).

FN_SP1(Leaf(x_2)) => Leaf(x_2).
FN_SP1(Branch(xt_2, i_3, yt_4)) =>
    Branch(FN_SP1(xt_2), i_3, FN_SP1(yt_4)).
```

The intermediate tree constructed after the first application of `Flip` is not constructed using the specialised definition of double-flip in the residual program. This is equivalent to the program generated by deforestation [Wad90].

5.3.3 Example: Queens

Next, consider the elimination of intermediate data structures from a “listful” implementation of a program solving the 10-queens problem (Figure 5.10). That is, the program will return the possible positions of n queens on a 10×10 chess board such that every queen is safe. A queen is *safe* if there is no other queen on the same row, column, or diagonal. This program is adapted from [Mar96]. The full program is contained in the Appendix, Section A.1.2. The main functions of the implementation are shown in Figure 5.10.

There are several lists in this program that are eliminated by higher-order deforestation [Mar96]. The constraint-based partial evaluator eliminates these intermediate data structures as well. For example, the call to `UpTo(1,10)` is unfolded by partial evaluation. The intermediate lists constructed by the calls to `Zip` and `From` are also removed by the partial evaluator. The main disadvantage of the constraint-based approach in this case is the restriction in the revised most specific safe generalisation algorithm that does not permit the inspection of a term in a lambda expression (§ 4.5.2). This causes the residual program to be larger than is strictly necessary.

```
FUNCTION  Queens : Integer -> List(List(Integer)).
Queens(n) =>
  IF    n = 0
  THEN  0
  ELSE  Map(Filter(ListCmp(n), Safe),
            LAMBDA[x](Concat(Fst(x),[Snd(x)]))).

FUNCTION  Safe : (List(Integer), Integer) -> Boolean.
Safe(<p,n>) =>
    FoldR(Map(Zip(From(1),p),
      LAMBDA[k](SOME[i,j,m](k = (i,j) & m = Length(p)+1 &
                          ~(j = n) & ~((i+j) = (m+n)) &
                          ~((i-j) = (m-n))))), &, True).

FUNCTION  ListCmp : Integer -> List((List(Integer),Integer)).
ListCmp(n) =>
  Join(Map(UpTo(1,10),LAMBDA[m](Map(Queens(n-1),LAMBDA[p](p,m))))).
```

Figure 5.10: A “listful” implementation of the 10-queens problem. The associated functions are given in Section A.1.2.

5.3.4 Higher-order Deforestation

In [Wad90], Wadler enumerates some higher-order terms which can be transformed using deforestation extended with higher-order macros. Higher-order macros are non-recursive definitions which may contain function variables in the right-hand side.

In constraint-based partial evaluation, it is not necessary to identify these higher-order function definitions; all functions are treated similarly by the interpreter during unfolding.

Consider the SumSquares program (Figure 5.8) extended with the following higher-order function definitions for `Map`, `Fold`, and `SumH`.

```
Map : List(a) * (a -> b) -> List(b).

Map([],_)     => [].
Map([a|as],f) => [f(a) | Map(as,f)].

Fold : List(b) * (a -> b -> a) * a -> a.

Fold([],_,a)     => a.
Fold([b|bs],f,a) => Fold(bs,f,f((a,b))).
```

```
SumH : List(Integer) -> Integer.

SumH => LAMBDA[xs](Fold(xs,+,0))
```

The Escher constraint-based partial evaluator is applied to this program (the SumSquares program with the three definitions above). Specialising the program with respect to the term

```
     SumH(Map(Square,UpTo(1,n)))
```

results in the following residual program.

```
FUNCTION  Ans : Integer -> Integer.
Ans(n) =>
    FN_SP11(n, 1, 0).

FUNCTION  FN_SP11 : Integer -> Integer.
FN_SP11(x_6, x_7, x_8) =>
    IF x_7 > x_6
    THEN x_8
    ELSE FN_SP11(x_6, (x_7 + 1), (x_8 + (x_7 * x_7))).
```

This is also an example of a specialisation where the top-down or the metric approach is necessary to find the “best” ancestor with which to generalise the growing term (§ 4.5.3).

5.3.5 Comparison with Other Techniques

Deforestation was originally developed to perform the elimination of intermediate data structures (§ 2.1.3). This type of program specialisation is also possible by supercompilation, positive supercompilation (§ 2.1.3), conjunctive partial deduction (§ 2.1.4), and the partial evaluation of narrowing-based functional logic programs (§ 2.3.2). Using a continuation-passing style transformation, partial evaluation based on Mix can also perform the intermediate data structure elimination [CD91]. In general, partial deduction cannot eliminate these structures.

5.4 Specialisation of Boolean Expressions

In this section, the extension of the algorithm to specialise Boolean expressions (§ 4.8.1) is evaluated. The program shown in Figure 5.11 computes the permutations of a list. It is originally from [Llo95].

```
FUNCTION  Nil : One -> List(a);
          Cons : a * List(a) -> List(a).

FUNCTION  Concat : List(a) * List(a) -> List(a).
Concat([],x) =>  x.
Concat([u|x],y) =>  [u|Concat(x,y)].

FUNCTION  Split : List(a) * List(a) * List(a) -> Boolean.
Split([],x,y) =>  x = [] & y = [].
Split([x|y],v,w) =>
    (v = [] & w = [x|y]) \/
    SOME [z] (v = [x|z] & Split(y,z,w)).

FUNCTION  Perm : List(a) * List(a) -> Boolean.
Perm([],l) =>  l = [].
Perm([h|t],l) =>
    SOME [u,v,r] (Perm(t,r) & Split(r,u,v) & l = Concat(u, [h|v])).
```

Figure 5.11: Program for computing the permutations of a list.

Specialising the program in Figure 5.11 with respect to the term `Perm([1|x],y)` using the constraint-based partial evaluator with the basic specialisation approach (i.e. no special handling of Boolean expressions) results in a residual program with no speed-up. The Boolean expression is divided immediately by the *divide* function (§ 4.5), illustrated in Figure 5.12. Residual definitions are generated for the individual terms `Perm(t,r)`, `Split(r,u,v)`, and `Concat(u,[1|v])`, resulting in no increase of efficiency.

On the other hand, if the constraint-based partial evaluator with the extension for handling Boolean expressions is used, the initial Boolean expression is not divided. Instead, the partial evaluator attempts to generate a residual definition for the entire Boolean expression. The difference between the two residual programs is clear from the schema statement defining the `Ans` function. Without the extension for handling Boolean expressions, the partial evaluator generates the following schema statement for rewriting `Ans(x,y)`.

```
Ans(x,y) =>
    SOME [u_3,v_3,r_3] ((FN_SP5(x,r_3) & (FN_SP8(r_3,u_3,v_3) &
                         (y = FN_SP12(u_3,v_3))))).
```

This schema statement is simply a renaming of the schema statement for rewriting

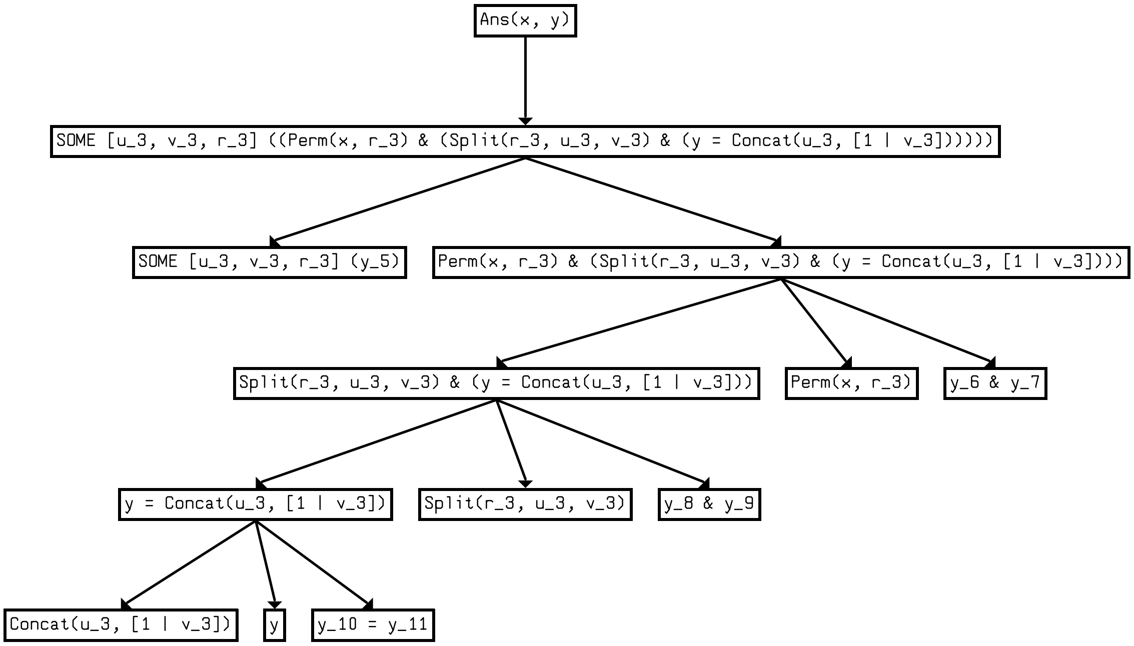

Ans(x, y)
SOME [u_3, v_3, r_3] ((Perm(x, r_3) & (Split(r_3, u_3, v_3) & (y = Concat(u_3, [1 | v_3])))))
SOME [u_3, v_3, r_3] (y_5)
Perm(x, r_3) & (Split(r_3, u_3, v_3) & (y = Concat(u_3, [1 | v_3])))
Split(r_3, u_3, v_3) & (y = Concat(u_3, [1 | v_3]))
Perm(x, r_3)
y_6 & y_7
y = Concat(u_3, [1 | v_3])
Split(r_3, u_3, v_3)
y_8 & y_9
Concat(u_3, [1 | v_3])
y
y_10 = y_11

Figure 5.12: Specialisation of the program computing the list permutations with respect to the term `Perm([1|x],y)` using the basic specialisation approach as described in Chapter 4. The *divide* function splits terms with outermost Boolean operators. This results in a loss of potential specialisation.

$length$	Original	Residual	Residual w/Ext
2	28	28	22
3	87	87	72
4	345	345	296
5	1714	1714	1531
6	10282	10282	9586

Table 5.1: The number of reductions to compute the permutation of a list in the original program, Program 5.11, the residual program generated by the standard constraint-based partial evaluator, and the residual program generated by the specialiser extended with advanced Boolean expression handling.

`Perm(Cons(h,t),y)` from the original program. On the other hand, using the constraint-based partial evaluator with the extension for specialising Boolean expressions results in the residual program with the definition for `Ans(x,y)` as shown below.

```
Ans(x,y) =>
    SOME [u_3,v_3,r_3] (FN_SP37(x,r_3,u_3,v_3,(y = FN_SP40(u_3,v_3)))).
```

The definition of the function `FN_SP37` in the residual program represents the residual definition for the term `Perm(x,r_3) & Split(r_3,u_3,v_3) & w`. The call to `Concat` has been split from the term in order to ensure termination of the transformation. The difference between the programs in terms of the number of reductions for lists of given length is illustrated in Table 5.1.

5.4.1 Comparison with Other Techniques

The specialisation of Boolean expressions has been addressed both in conjunctive partial deduction (CPD) [LSdW96, GJMS96] and narrowing-based functional logic partial evaluation (NPE) [AFV96a]. As discussed in Section 2.1.4, CPD extends traditional partial deduction by allowing conjunctions of atoms in the set **A**. The global control of CPD requires determining the “best” partition of the conjunctions in order to ensure termination of the specialiser [GJMS96]. Likewise, recent work in NPE has addressed the control of partial evaluation. Namely, conjunctions in narrowing-based functional logic languages are partitioned in a method inspired by [GJMS96].

The extension of the constraint-based partial evaluation procedure to handle Boolean expressions does not involve a specialised version of the local and global control as in these procedures. Boolean expressions are the focus of logic programming languages, and occur very often in narrowing-based functional logic programs. For a rewriting-based functional logic language, such an extension of the control is optional, but not necessary.

5.5 Specialisation of the 91-function

The specialisation of McCarthy's 91-function is discussed in this section. The 91-function is a highly recursive function; typically programs containing such functions are difficult to specialise, because the recursive behaviour of the functions causes the global control to generalise terms early.

In particular, the techniques used by generalized partial computation (§ 2.1.3) to obtain the optimal specialisation of the 91-function are closely evaluated. As originally defined in [FN88], generalized partial computation (GPC) is not an automatic procedure. Both GPC and constraint-based partial evaluation use constraint-based information propagation. However, the powerful theorem proving capability of GPC allows the procedure to generate the optimal residual program in this case, while automatic techniques are not able to do so. By examining the transformation by GPC in detail, features may be identified that could improve the specialisation precision of automatic partial evaluators. The material of this section was previously published in [LG98b].

McCarthy's 91-function uses nested recursion to return 91 for any $x \leq 100$.

```
FUNCTION  F : Integer -> Integer.
F(x) =>
    IF    x > 100
    THEN  x - 10
    ELSE  F(F(x + 11)).
```

Application of the GPC rules, as defined in [FN88], eliminates all the recursion in the 91-function, whereas even with constraint solving, the same transformation cannot be achieved with constraint-based partial evaluation (CPE). Generalized partial computation transforms the 91-function into the following function:

```
F(x) =>
    IF    x > 100
    THEN  x - 10
    ELSE  91.
```

The GPC transformation process is shown in Figure 5.13 (illustrated in Escher syntax). The first function `H1` represents unfolding the term `F(x)` using one computation step. The term in the else-branch of the conditional, `F(F(x+11))`, is extracted and a residual definition of the term is generated by the specialiser. In addition, the expression `x =< 100` is identified with the term. GPC is defined for a call-by-value language; therefore, the innermost call to `F` is reduced first, then the outermost call to `F` is propagated through the conditional. The term in the else-branch of the conditional, `F(F(F(x+11+11)))` is folded twice, and the term in the then-branch is extracted to be specialised further.

Associated with the term `F(x+1)` is the expression `89 < x =< 100`. Evaluating this term with its associated information allows the condition of the conditional statement to be reduced to `x = 100`, since `x` is between 89 and 101. Substituting `100` for `x` in the then-branch of the conditional expression results in the term `91`. The term in the else-branch is folded to `H2` and then evaluated again to obtain `H3(x+1)`.

At this stage of the program transformation, some recursion removal rules are applied to the definitions. These recursion removal rules are assumed to be generated by the theorem prover. The rules are defined as follows:

R1: if `H(x) => IF x = b THEN a ELSE H(x+1)` and `x =< b`, then `H(x) => a`.

R2: if `H(x) => IF c >= x > c-d THEN a ELSE G(H(x+d))`, where `d > 0` and `G(a) => a`, then `H(x) => a` for `x` such that `x =< c` and `G` does not contain `x` as a free variable.

The application of rule R1 (as indicated in Figure 5.13) transforms the conditional expression `ITE(x = 100, 91, H3(x+1))` to `91`. Then, the rule R2 is applied to the definition of `H2` to obtain the final residual definition.

The constraint-based partial evaluator [LG98a] transforms the 91-function into the program shown in Figure 5.14.

When the residual programs generated by CPE and GPC are compared, one can identify the similar unfolding/folding approach of both techniques. However, several actions of GPC cannot be performed in CPE. Some of these are enumerated below.

- In GPC, information from the set of associated constraints/expressions can be reintroduced into the terms of the transformation.

  For example, in Figure 5.13, the current information (set of constraints) is integrated into the statements at two positions, both underlined. Using the information in the expression `89 <`

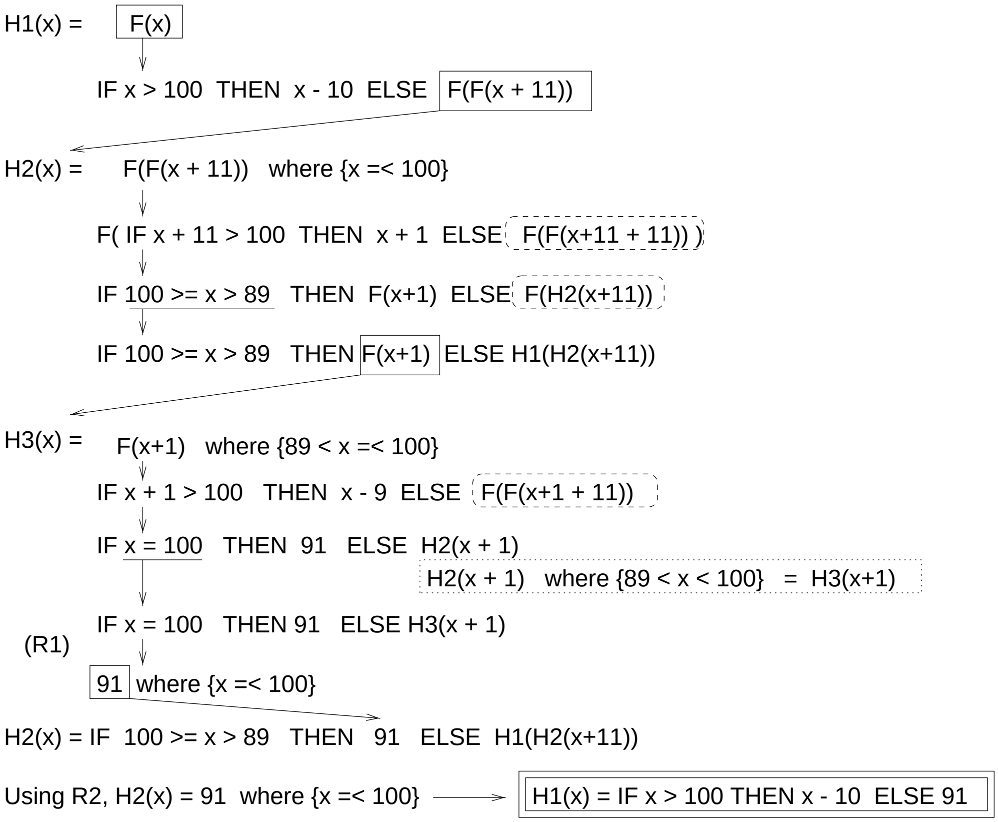

H1(x) = F(x)
IF x > 100 THEN x - 10 ELSE F(F(x + 11))
H2(x) = F(F(x + 11)) where {x =< 100}
F(IF x + 11 > 100 THEN x + 1 ELSE F(F(x+11 + 11)))
IF 100 >= x > 89 THEN F(x+1) ELSE F(H2(x+11))
IF 100 >= x > 89 THEN F(x+1) ELSE H1(H2(x+11))
H3(x) = F(x+1) where {89 < x =< 100}
IF x + 1 > 100 THEN x - 9 ELSE F(F(x+1 + 11))
IF x = 100 THEN 91 ELSE H2(x + 1)
H2(x + 1) where {89 < x < 100} = H3(x+1)
IF x = 100 THEN 91 ELSE H3(x + 1)
(R1)
91 where {x =< 100}
H2(x) = IF 100 >= x > 89 THEN 91 ELSE H1(H2(x+11))
Using R2, H2(x) = 91 where {x =< 100}
H1(x) = IF x > 100 THEN x - 10 ELSE 91

Figure 5.13: Transformation of the 91-function by GPC. The dashed boxes indicate that a fold operation has been performed. $R1$ and $R2$ are names for the two recursion removal rules.

```
FUNCTION   Ans : Integer -> Integer.
Ans(x) =>
   IF    x > 100
   THEN  x - 10
   ELSE  FN_SP1(x).

FUNCTION   FN_SP1 : Integer -> Integer.
FN_SP1(x) =>
   IF    x > 89
   THEN  FN_SP3(x)
   ELSE  (IF   FN_SP1(x+11) > 100
         THEN  FN_SP1(x+11) - 10
         ELSE  FN_SP1(FN_SP1(x+11))).

FUNCTION   FN_SP3 : Integer -> Integer.
FN_SP3(x) =>
   IF    x > 99
   THEN  x - 9
   ELSE  FN_SP1(x+1).
```

Figure 5.14: The result of specialising the 91-function using constraint-based partial evaluation.

`x =< 100`, the condition in the transformation of `F(x+1)` was replaced with the condition `x = 100`. In the other example, the condition of the conditional statement is extended with the information from the associated constraint set.

With the addition of this feature in CPE, the new definitions of `H21` and `H31` in Figure 5.14 are:

```
H21(x) => IF 100 >= x > 89  ...
H31(x) => IF x = 100  ...
```

The advantage of this transformation depends on the implementation of the rewriting-based functional logic language in question.

- The constraint `x = 100` is propagated via unification to obtain the value `91` in Figure 5.13. It is not possible to perform the substitution of terms based on an equality test in the condition of a conditional in CPE. Such instantiations may adversely affect the correctness of the transformation.

- In the above example, the term `F(F(x+1+11))` is unfolded to obtain the term `H3(x+1)`. In CPE, this term would not be able to be unfolded further, as is violates the ordering relation on the terms.

	Original Program		Residual Program	
x	reds	time	reds	time
100	19	20 ms	9	$<$10 ms
95	1120	10300 ms	69	90 ms
90	36817	860820 ms	179	430 ms
85	$>$ 301200	—	1418	9130 ms
80	—	—	3728	31030 ms
75	—	—	26366	359580 ms
70	—	—	74876	1175090 ms
65	—	—	461633	9738930 ms

Table 5.2: Run-times of the original 91-function program and the residual program generated by constraint-based partial evaluation for given values of n.

- The call-by-value semantics of the GPC permits the construction of the term `H1(H2(x+11))`. The same structure occurs in the definition of `FN_SP1` in the residual program (Fig. 5.14), but it cannot be identified during the transformation. This may be able to be identified by a post-processing folding operation.

- In GPC, the theorem prover can generate recursion removal rules to improve the quality of the residual program. The rules are "invented" using information about the well-founded ordering on the natural numbers. These rules allow the elimination of the recursion from the specialised definitions. For specialisation with CPE, these rules might be accessed from a library. For example, in GPC with constraints, equations describing properties for each relevant data type and primitive function are imported before specialisation [Tak92]. However, this is not a general solution.

Together, these features cause a great difference in the specialisation obtained by the two program transformers. Some of these may be able to be integrated directly, and some as post-processing steps. Having a library of axioms corresponding to implemented constraint domains might be a first step towards integrating the power of theorem proving in CPE. For example, the form of the first recursion removal rule, R1, is quite generic, and appears in many programs (e.g. programs with counters). The incorporation of an "equivalence replacement" step using an automatic theorem prover in CPE would provide further flexibility.

On the other hand, even the basic specialisation that is possible by CPE has a large impact on the run-time of the program. Table 5.2 shows the difference in efficiency between the original 91-function and the program of Figure 5.14.

5.5.1 Comparison with Other Techniques

Few references to the specialisation of the 91-function exist in published literature. Traditional partial deduction obtains no specialisation of the 91-function. It is unclear if any specialisation of the function could be obtained by positive supercompilation; the function cannot be expressed straightforwardly in the first-order functional language for which positive supercompilation is defined [SGJ96]. On the other hand, reference to the function exists in [JBE94]; analysis of the 91-function using interval constraints derives the interval `[91,91]` for `X` given the initial constraint `X =< 100`. The optimal specialisation of the 91-function by an automatic technique remains the subject of current research.

5.6 Experimenting with the Ackermann function

The Ackermann function, like the 91-function, is an example of a highly-recursive function. It is the simplest example of a well-defined total function that is computable, but not primitive recursive [Rog67]. Its recursive structure makes the function difficult to transform by most automatic program specialisers. The function, in Escher syntax, is shown below.

```
Ack(m,n) =>
  IF   m = 0
  THEN n + 1
  ELSE (IF   n = 0
        THEN  Ack(m-1,1)
        ELSE  Ack(m-1,Ack(m,n-1))).
```

The partial evaluation of this example demonstrates the effect of the ancestor selection (§ 4.5.3) on the quality of the residual program. The m-tree shown in Figure 5.15 is generated after two iterations of the partial evaluator specialising the Ackermann function with respect to the term `Ack(2,n)`. The terms of the right hand branch of the m-tree are the following:

t_1 : `Ans(n)`

t_2 : `ITE(n=0, Ack((2-1),1), Ack((2-1),Ack(2,(n-1))))`

t_3 : `Ack((2-1),Ack(2,(n-1)))`

t_4 : `ITE(ITE(n=1, Ack((2-1),1), Ack((2-1),Ack(2,((n-1)-1)))),`
`Ack(((2-1)-1),1), Ack(((2-1)-1), Ack((2-1), Ack(2,(n-1))-1)))`

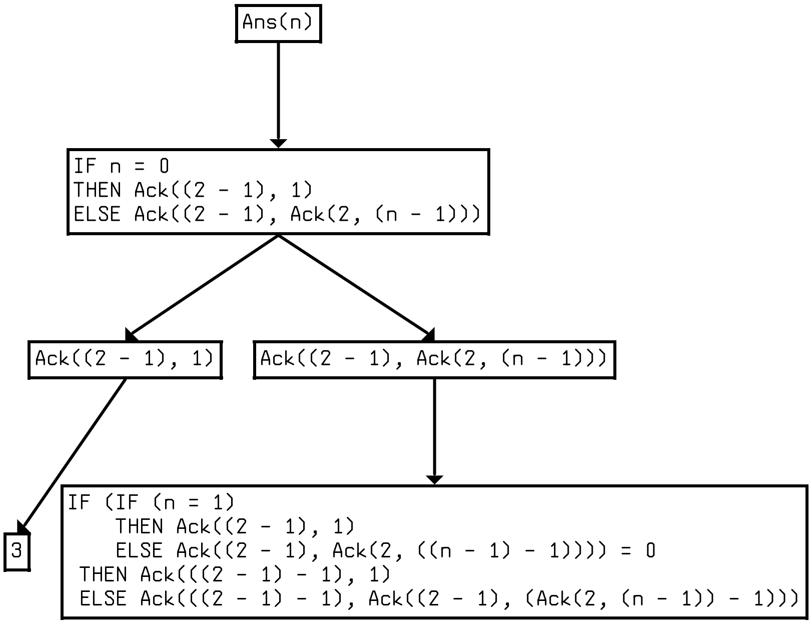

Ans(n)
IF n = 0
THEN Ack((2 - 1), 1)
ELSE Ack((2 - 1), Ack(2, (n - 1)))
Ack((2 - 1), 1)
Ack((2 - 1), Ack(2, (n - 1)))
3
IF (IF (n = 1)
THEN Ack((2 - 1), 1)
ELSE Ack((2 - 1), Ack(2, ((n - 1) - 1)))) = 0
THEN Ack(((2 - 1) - 1), 1)
ELSE Ack(((2 - 1) - 1), Ack((2 - 1), (Ack(2, (n - 1)) - 1)))

Figure 5.15: The m-tree constructed after two iterations of the partial evaluator during the specialisation of the Ackermann function with respect to the term `Ack(2,n)`.

The term t_4 stored in the leaf node of the right-hand side branch embeds both ancestor terms t_2 and t_3. Programs of different quality result from the ancestor selection strategy chosen by the user. For example, using the bottom-up selection strategy (§ 4.5.3), the partial evaluator will return the residual program shown in Figure 5.16 (the type declarations have been omitted for the sake of simplicity).

In this case, the call to `Ack(2,n)` is unfolded once, resulting in the residual definition of the `Ans` function. The term t_3 is generalised in order to ensure the termination of the specialisation. This results in the first argument of t_3 being removed by the partial evaluator. Therefore, the function `FN_SP13` is the residual definition for the term `Ack(x-1,Ack(x,y-1))`.

On the other hand, using the partial evaluator with either the top-down selection strategy or the selection metric will result in the residual program shown in Figure 5.17.

In this case, the generalisation operator will replace t_2 with the msg of t_2 and t_4. The term which is renamed to `FN_SP10` in this residual program is

```
ITE(x_2=0, Ack(x_3-1,1), Ack(x_3-1,Ack(x_3,x_4-1)))
```

```
Ans(n) =>
    IF n = 0
    THEN 3
    ELSE FN_SP13(2, n).

FN_SP13(x_4, x_5) =>
    IF x_4 = 1
    THEN (IF (x_5 = 1)
          THEN 2
          ELSE FN_SP13(x_4, (x_5 - 1))) + 1
    ELSE (IF  (IF   x_4 = 0
               THEN ((x_5 - 1) + 1)
               ELSE (IF (x_5 = 1)
                     THEN FN_SP13((x_4 - 1), 1)
                     ELSE FN_SP13(x_4, (x_5 - 1)))) = 0
          THEN FN_SP62(x_4)
          ELSE FN_SP13((x_4 - 1), (IF (x_4 = 0)
                      THEN ((x_5 - 1) + 1) ELSE (IF (x_5 = 1)
                      THEN FN_SP80(x_4) ELSE FN_SP13(x_4, (x_5 - 1)))))).

FN_SP62(x_8) =>
    IF   ((x_8 - 1) - 1) = 0
    THEN 2
    ELSE FN_SP13(((x_8 - 1) - 1), 1).

FN_SP80(x_4) =>
    IF   x_4 = 1
    THEN 2
    ELSE FN_SP13((x_4 - 1), 1).
```

Figure 5.16: The residual program obtained by specialising the Ackermann function with respect to the term `Ack(2,n)` using the bottom-up approach to selecting ancestors in the m-tree.

```
Ans(n) =>
    FN_SP10(n, 2, n).

FN_SP10(x_2, x_3, x_4) =>
    IF x_2 = 0
    THEN FN_SP22(x_3, 1)
    ELSE FN_SP13(x_3, x_4).

FN_SP22(x_7, x_8) =>
    IF   x_7 = 1
    THEN x_8 + 1
    ELSE (IF (x_8 = 0)
         THEN FN_SP22((x_7 - 1), 1)
         ELSE FN_SP22((x_7 - 1), FN_SP22(x_7, (x_8 - 1)))).

FN_SP13(x_3, x_4) =>
    IF x_3 = 1
    THEN FN_SP10((x_4 - 1), x_3, (x_4 - 1)) + 1
    ELSE (IF (IF (x_3 = 0)
             THEN x_4
             ELSE (IF (x_4 = 1)
                   THEN FN_SP13((x_3 - 1), 1)
                   ELSE FN_SP13(x_3, (x_4 - 1)))) = 0
          THEN FN_SP92(x_3)
          ELSE FN_SP13((x_3 - 1), (IF (x_3 = 0) THEN x_4 ELSE
                       (IF (x_4 = 1) THEN FN_SP112(x_3) ELSE
                       FN_SP13(x_3, (x_4 - 1)))))).

FN_SP92(x_7) =>
    IF ((x_7 - 1) - 1) = 0
    THEN 2
    ELSE FN_SP13(((x_7 - 1) - 1), 1).

FN_SP112(x_3) =>
    IF x_3 = 1
    THEN 2
    ELSE FN_SP13((x_3 - 1), 1).
```

Figure 5.17: The residual program obtained by specialising the Ackermann function with respect to the term `Ack(2,n)` using the top-down or metric approach to selecting ancestors in the m-tree.

	Original Program		Program P_{TD}		Program P_{BU}	
n	reds	time	reds	time	reds	time
0	30	30 ms	24	10 ms	3	<10 ms
1	183	870 ms	129	460 ms	85	260 ms
2	1220	15720 ms	876	8020 ms	593	5740 ms
3	10137	272890 ms	7358	148490 ms	5070	105280 ms
4	104029	4614300 ms	76218	2844700 ms	53297	2234400 ms

Table 5.3: Example run-times of the original Ackermann program, the residual program of Figure 5.17, P_{TD}, and the residual program of Figure 5.16, P_{BU}.

In either case, the quality of the residual program is not as great as can be obtained using a semi-automatic unfold/fold-based specialiser (see [JGS93] for the optimal residual program). The global control of the constraint-based partial evaluation algorithm prevents the full unfolding of this example, in order to avoid non-termination of the specialiser. However, it should be noted that the two residual programs of Figures 5.16 and 5.17 differ in efficiency as well. Notably, the program obtained using the bottom-up approach to finding ancestors in the m-tree is better in terms of efficiency and size than the program obtained using either the top-down or the metric approach! Example timings are presented in Table 5.3.

The partial evaluator with the top-down selection strategy removes the constant `2` from the term t_2 by the generalisation. This is illustrated in Figure 5.18.

The static data is removed from the term by generalisation before it can be adequately used by the partial evaluator. On the other hand, the bottom-up strategy allows the specialiser to generate a definition for `Ack(2,n)` before it is generalised.

Therefore, while in the deforestation examples (§ 5.3), using the metric permitted the elimination of the intermediate data structures, using the metric in the partial evaluation of the Ackermann function results in a program that shows linear speedup, but is not the best solution. It would seem that this metric is too conservative; allowing the specialiser to unfold the term once before it is generalised results in noticeable speedup of the program. On the other hand, always permitting extra unfolding is not a solution to the problem. Perhaps the metric should ensure that the static data is used at least once during the partial evaluation; however, the method for integrating such a metric is an area for future work (§ 7.1.3).

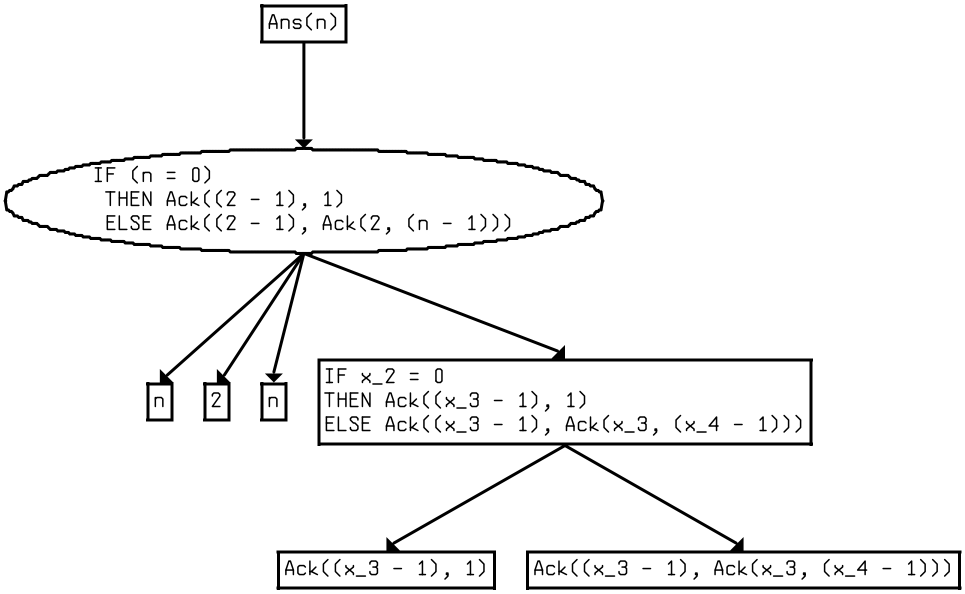

Ans(n)
IF (n = 0)
THEN Ack((2 - 1), 1)
ELSE Ack((2 - 1), Ack(2, (n - 1)))
n
2
n
IF x_2 = 0
THEN Ack((x_3 - 1), 1)
ELSE Ack((x_3 - 1), Ack(x_3, (x_4 - 1)))
Ack((x_3 - 1), 1)
Ack((x_3 - 1), Ack(x_3, (x_4 - 1)))

Figure 5.18: The base of the m-tree constructed by partial evaluating the Ackermann function with respect to the term Ack(2,n) using the top-down or metric selection strategy. The elliptical node contains the generalised term.

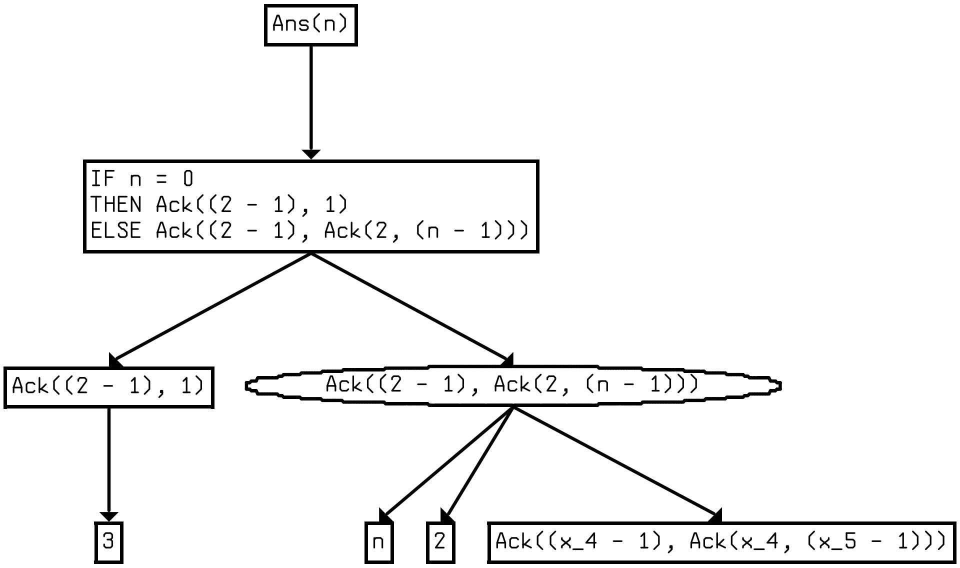

Ans(n)
IF n = 0
THEN Ack((2 - 1), 1)
ELSE Ack((2 - 1), Ack(2, (n - 1)))
Ack((2 - 1), 1)
Ack((2 - 1), Ack(2, (n - 1)))
3
n
2
Ack((x_4 - 1), Ack(x_4, (x_5 - 1)))

Figure 5.19: The base of the m-tree constructed by partial evaluating the Ackermann function with respect to the term Ack(2,n) using the bottom-up selection strategy. The elliptical node contains the generalised term.

5.6.1 Comparison with Other Techniques

Using the SP partial evaluator for logic programs [Gal91], the following residual program can be obtained by the specialisation of the Ackermann function with the first argument fixed to two.

```
ack(2,X1,X2) :- ack2_1(X1,X2).
ack2_1(0,3) :- true.
ack2_1(X1,X2) :-
      X1=\=0,
      X3 is X1-1,
      ack2_1(X3,X4),
      ack2_2(X4,X2).
ack2_2(0,2) :- true.
ack2_2(X1,X2) :-
      X1=\=0,
      X3 is X1-1,
      ack2_2(X3,X4),
      X2 is X4+1.
```

This program is on average two times faster (using the speedup metric defined in Section 5.8) than the program in Figure 5.16, P_{BU}. The unfold/fold program transformation generates a program with a similar structure [JGS93]. In addition, positive supercompilation achieves this result, but it is necessary to isolate the nested call to `Ack` to ensure termination of the specialisation [Sør96]. This approach seems to be the best technique for avoiding the nested conditional expression in the term t_4. By handling local definitions as in positive supercompilation [SGJ96], that is transforming the term in the local definition separately, the transformation can obtain a better residual program. As an aside, Welinder demonstrated in [Wel97] that the partial evaluation of the Ackermann function using a relatively small set of program transformation rules generates a residual program with the same structure as the logic program above. As a final aside, Welinder notes that the claim that partial evaluation makes the Ackermann function "faster" should be interpreted as "requires less reductions", as the function grows too fast to be of any practical use.

5.7 Specialising an Interpreter

This section presents results from several experiments involving the specialisation of an interpreter for a simple imperative language. The programs in this section are adapted from a benchmark example from [Leu]. There are two goals in these experiments:

- To test the constraint-based partial evaluator on the specialisation of an interpreter, and

- To study further the effects of the program structure on the quality of the residual programs.

The expressive nature of the Escher language permits several styles of implementation of this interpreter. A “functional” implementation of the interpreter is shown in Figure 5.20.

One may note that the `Store` function is not defined in the program. The implementation of the `Store` function is key to the quality of the residual program. Different implementations of the `Store` function will be discussed later in this section.

Another observation of the program is that the `Power` function defines a program that does not require initial input values from the environment `e1`. This causes the specialisation of the program to be simpler than if there was an initial lookup of a variable’s value in the object program. However, this feature of the benchmark program was left unaltered; examples involving the partial evaluation of an interpreter that requires the initial environment are featured in Chapter 6.

An alternative implementation of the `Power` and `Exec` functions is shown in Figure 5.21. These functions are written in more “logical” style. The readability of the `Power` function improves by this conversion.

In these experiments, the two interpreters specified above will be partially evaluated with respect to the term `Power(2,5,e1)` and `Power(2,5,e1,e2)`, respectively. However, before this can occur, the `Store` function must be defined. A tail-recursive implementation of this function, `Store1`, is shown below.

```
FUNCTION  Store1 : List((String,Integer)) * String * Integer ->
                   List((String,Integer)).

Store1(e, k, v) => [(k,v) | Rem(e,k)].

Rem([], k) => [].
Rem([h|t], k) =>
   IF    k = Fst(h)
   THEN  t
   ELSE  [h | Rem(t,k)].
```

However, this implementation is not similar to the predicate defined in the benchmark example. Another possible implementation of the function which changes the values in the environment, `Store2`, is shown below.

```
FUNCTION  Store2 : List((String,Integer)) * String * Integer ->
                  List((String,Integer)).
```

```
CONSTRUCT Expr/0, Stmt/0.
FUNCTION  I : Integer -> Expr;
          V : String -> Expr;
          Add, Times, Less : Expr * Expr -> Expr;
          Null : One -> Stmt;
          Let : String * Expr -> Stmt;
          WhileDo : Expr * Stmt -> Stmt;
          Seq : Stmt * Stmt -> Stmt.

FUNCTION  Power : Integer * Integer * List((String,Integer)) ->
                  List((String,Integer)).

Power(base, pwr, e1) =>
   Exec(Seq(Seq(Let("X",I(1)), Let("Result",V("Base"))),
                WhileDo(Less(V("X"),V("Pwr")),
                    Seq(Let("X",Add(V("X"),I("1"))),
                    Let("Result",Times(V("Result"),V("Base")))))),
       Exec(Let("Pwr",I(pwr)), Exec(Let("Base",I(base)),e1))).

Exec(Null, env) => env.
Exec(Let(v,expr), env) =>
   Store(env, v, EvalExpr(expr,env)).
Exec(WhileDo(tst,lp), env) =>
   IF    EvalTest(tst,env)
   THEN  Exec(Seq(lp,WhileDo(tst,lp)), env)
   ELSE  Exec(Null, env).
Exec(Seq(st1,st2), env) =>
   Exec(st2, Exec(st1,env)).

EvalTest(Less(x,y),env) =>
   EvalExpr(x,env) < EvalExpr(y,env).

EvalExpr(I(x),env) => x.
EvalExpr(V(v),env) => Lookup(env,v).
EvalExpr(Add(x,y),env) =>
   EvalExpr(x,env) + EvalExpr(y,env).
EvalExpr(Times(x,y),env) =>
   EvalExpr(x,env) * EvalExpr(y,env).

Lookup([], k) => 0.
Lookup([s|ss], k) =>
   IF   k = Fst(s)
   THEN Snd(s)
   ELSE Lookup(ss, k).
```

Figure 5.20: An interpreter for a simple imperative language written in a functional style. Inferable type declarations have been omitted.

```
FUNCTION  Power : Integer * Integer * List((String,Integer)) *
                  List((String,Integer)) -> Boolean.
Power(base, pwr, e1, e4) =>
  SOME [e2, e3] (
  Exec(Let("Pwr",I(pwr)), e1, e2) &
  Exec(Let("Base",I(base)), e2, e3) &
  Exec(Seq(Seq(Let("X",I(1)),
        Let("Result",V("Base"))),
        WhileDo(Less(V("X"),V("Pwr")),
           Seq(Let("X",Add(V("X"),I("1"))),
           Let("Result",Times(V("Result"),V("Base")))))), e3, e4)).


Exec(Null, e, e2) => e2 = e.
Exec(Let(v,expr), e, e2) =>
   Store(e, v, EvalExpr(expr,env), e2).
Exec(WhileDo(tst,lp), e, e2) =>
   IF    EvalTest(tst, e)
   THEN  Exec(Seq(lp,WhileDo(tst,lp)), e, e2)
   ELSE  Exec(Null, e, e2).
Exec(Seq(st1,st2), e, e3) =>
   SOME [e3] (Exec(st1, e, e2) & Exec(st2, e2, e3)).
```

Figure 5.21: The implementation of the Power and Exec functions in a logical style. The rest of the functions of the interpreter are assumed to be unchanged from the implementation in Figure 5.20.

```
Store2([], k, v) => [(k,v)].
Store2([s|ss], k, v) =>
   IF    k = Fst(s)
   THEN  [<k,v> | ss]
   ELSE  [s | Store2(ss, k, v)].
```

Therefore, there are four implementations of the interpreter that can be specialised by the Escher partial evaluator. These programs will be referred to as follows.

- F_1: the functional implementation of the interpreter (Figure 5.20) with `Store1`;
- F_2: the functional implementation of the interpreter (Figure 5.20) with `Store2`;
- L_1: the logical implementation of the interpreter (Figure 5.21) with `Store1`;
- L_2: the logical implementation of the interpreter (Figure 5.21) with `Store2`;

Timing results and sizes for the residual programs are shown in the table below.

	Original		Residual	
Program	Size	Reds	Size	Reds
F_1	17	774	21	44
F_2	18	1835	282	3338
L_1	17	1216	21	44
L_2	18	1068	35	192

The residual program with the worst quality resulted from the program F_2; the size of the residual program in this case is nearly 16 times larger than the original interpreter. The increase in the number of reductions is a result of repeated terms in the residual programs; these should be eliminated by the local definition introduction post-processing operation (§ 4.8.5). The best residual programs, in terms of efficiency and size, resulted from the specialisation of the programs F_1 and L_1. The specialised F_1 program is shown below.

```
Ans(e) =>
    [<"Result", 32>,
     <"X", 5>,
     <"Pwr", 5>,
     <"Base", 2> | FN_SP48(FN_SP47(FN_SP34(FN_SP28(e))))].
```

The key to this efficient residual program lies in the implementation of the function for storing new values in the environment. Checking the values of the variables by a conditional expression requires the elements of the environment to be specified at compile-time, if good specialisation can occur. On the other hand, if the environment is completely unknown at specialisation time, the tail-recursive implementation of `Store1` allows the values to be stored in a non-ground environment. This permits the full unfolding of the `Power` function.

Therefore, the residual programs resulting from the specialisation of the interpreters with `Store2` are less efficient than those generated from interpreters with `Store1`. It should be noted that the residual program generated by partially evaluating program L_2 has the same structure as the original benchmark logic program from [Leu] when specialised using the SP system [Gal91].

5.8 Further Experimental Results

In this section, results from a variety of partial evaluation examples are presented. Table 5.4 incorporates both the results from Sections 5.2 to 5.7 with extra example specialisations, detailed in the Appendix, Section A.1.

Table 5.4 is composed as follows. The examples presented in the earlier sections of this chapter are listed at the beginning of the table. One should note that the longer time required by the specialised version of the MapSumSq residual program is a result of the implementation of the Escher system. Implementation of the laziness of the language would eliminate this overhead. The best specialisation is achieved from the highly recursive examples (91, Ackermann) or from those examples that can be fully unfolded (Neighbours, BubbleSort).

In Table 5.4, the measure of the speedup of the program was computed using the equation:

$$speedup(P) = \frac{reds_P(t)}{reds_{P_A}(t)}$$

where $reds_P(t)$ is the number of reduction steps required to reduce the term t to its normal form in P and $reds_{P_A}(t)$ is the number of reduction steps required to rewrite the term t to its normal form in P_A.

This differs from the typical computation of the speedup of the residual program. Usually, the speedup of a program is defined as the ratio of the run-time required to compute the term t in P to the run-time required to compute t in the specialised program P_A. The Escher implementation used in these experiments was a prototype implementation. Given this, it was decided that the number

	Original Pgm			Residual Pgm			
Benchmark	Reds	Time	Size	Reds	Time	Size	Speedup
Match	325	120	7	228	70	9	1.43
Concat	51	1050	2	35	730	5	1.46
SumSqUpTo	1581	45110	15	1489	42890	2	1.06
SumSqUpToTree	111	180	15	64	90	3	1.73
Flip	25	80	3	13	40	3	1.92
Queens	11268	533200	22	4989	461040	70	2.26
MapSumSq	1643	17930	15	1489	19400	2	1.10
Perm	87	440	6	72	230	13	1.21
Connect	96	230	2	11	10	1	8.73
Neighbours	496	1080	3	21	40	1	12.4
91 (n=90)	36817	860620	1	179	430	3	205
Ackermann (m=2)	1220	15720	1	593	5740	5	2.06
BubbleSort	48	40	5	7	<10	1	6.86
Lambda	16	20	2	7	10	1	2.29
Lookup	22	40	5	6	10	1	3.33
MapReduce	93	220	5	48	110	5	1.94
Palindrome	181	1350	4	178	1280	3	1.02
SortBy	337	2970	5	259	1920	5	1.30
Solve (FP)	774	5720	17	44	320	21	17.6
Solve (FLP)	1216	76480	17	44	320	21	27.6
Solve (LP)	1068	11030	18	192	880	35	5.56

Table 5.4: Results from the specialisation of several benchmark examples using constraint-based partial evaluation. The run-times of the programs are measured in milliseconds.

of reductions of the term would give a better indication of the performance of the residual program, thus avoiding Gödel-specific effects on the results. On the other hand, the time required for memory allocation or garbage collection is not reflected in the speedup measure. Therefore, the speedup resulting from improved memory usage in the deforestation examples may not be evident from the measure used above.

5.9 Discussion and Summary

This chapter began with the implementation of the constraint-based partial evaluator for the transformation of Escher programs. The partial evaluator is built “on top” of the Escher implementation in the Gödel logic programming language. The main data structure of the implementation is that representing the m-tree. The constraint solving features of the implementation were presented, including the constraint handling rules implemented as rewrite rules of the Escher language and the widening operations for the various constraint domains. Finally, an example session demonstrated the integration of the extensions of the algorithm, as presented in Section 4.8.

In the remainder of the chapter, several experiments were used to compare the quality of the residual programs generated by the constraint-based partial evaluator with those obtainable by other established program specialisation procedures. The constraint-based partial evaluator passes the KMP-test by converting a naive tail-recursive pattern matching program into a pattern matcher based on the efficient Knuth-Morris-Pratt algorithm. Various examples demonstrated the ability of the partial evaluator to perform deforestation, even with higher-order functions. The extension of the algorithm for the advanced specialisation of Boolean expressions was demonstrated in the partial evaluation of a list-permutation program. The power of the partial evaluator was studied by comparing its specialisation of the 91-function and the Ackermann function with the results of comparable specialisers. Finally, an examination of the effect of the program structure on the specialisation was performed during the specialisation of the interpreter benchmark. Timing results for a variety of example programs, detailed in the Appendix, were presented in Table 5.4.

This chapter contained multiple studies of the effect of the program structure on the residual program. The interpreter example (§ 5.7) demonstrated the notable difference in quality between residual programs that can occur given the implementation of *one* function in the program. A pre-processing analysis of the program may be developed to identify these functions and convert them into a specialisable form. Of course, the ideal program structure depends on the term t with which the program is being partially evaluated. Specialising the interpreter with respect to the term `Power(2,5,[<"Y",y>, <"Z",z>])` would have resulted in residual programs of much better quality for all the implementations.

Another area of future work identified in this chapter is the definition of the optimal ancestor selection metric. In Sections 5.3 and 5.6, the effect of the selection strategy on the quality of the residual program was studied. It seems that the selection strategy should ensure that the static data is used during the specialisation at least once; on the other hand, allowing extra terms in the set A indeterminately will increase the size of the residual programs without ensuring that the efficiency will improve.

In general, the Escher constraint-based partial evaluator performed well in comparison with the established techniques. The constraint-based partial evaluator has specialisation power comparable with conjunctive partial deduction and narrowing-based functional logic partial evaluation. The constraint solving permits more information propagation that is possible in positive supercompilation. The most dramatic speedup, of course, resulted from the specialisation of the 91-function. This could not have been achieved without the constraint solving integration of this procedure. On the other hand, the semi-automatic generalized partial computation remains the most powerful transformer in comparison.

Chapter 6

Compiled Simulation by Program Specialisation

This chapter presents the application of the Escher constraint-based partial evaluator to the problem of digital circuit simulation. The aim of this work is to automatically improve the efficiency of a general interpreted-code simulator by specialising it with respect to a particular digital circuit design.

Typical functional or logical testing of a digital circuit design requires repeated simulation of the circuit for a test bench of stimuli. Partially evaluating an interpreted-code simulator allows designers to have both flexibility and efficiency in a single, simple program. That is, interpreted-code hardware simulators are beneficial on their own when the circuit design is being changed often; time does not have to be spent compiling the design before it can be simulated. On the other hand, when the design is more stable, the interpreted-code simulator can be automatically specialised by partial evaluation for a current design in order to "compile" the design of the component into the simulator. Previous research in gate-level simulation reported residual programs could be generated which are up to 91 times faster than the original simulator [BW90].

In this chapter, the goal is to apply partial evaluation to a simulator for high-level Verilog modules. Functional testing and validation at the register-transfer level is growing in popularity as the complex circuits require more time to be transformed by standard logic synthesis tools. At this level, interpreted-code simulators are particularly inefficient; most simulation at this level is performed by compiled-code simulators. However, compiled-code simulators are difficult to write in comparison to interpreted-code simulators. By specialising an interpreted-code simulator with respect to a particular register-transfer or behavioural level design, one should be able to "compile" the design of the component into the simulator, thus avoiding the interpretive overhead.

A simulator for a synthesizable subset of Verilog `V0` was implemented in the Escher rewriting-based functional logic language. This simulator was simply an implementation of the event semantics for the `V0` language, defined by Gordon in [Gor95, Gor98]. Traditionally, event-driven gate-level simulators did not specialise well by partial evaluation [AWS91][1]. This theory was to be tested by the experimentation of this chapter. To the best of the author's knowledge, this is the first research to address the specialisation of high-level digital circuit simulators.

The chapter is organised as follows. A brief introduction to hardware description languages (HDLs) in Section 6.1 and hardware simulation in Section 6.1.2. Then, the simulation semantics for the Verilog HDL are reviewed in Section 6.2. In particular, language complexities are discussed in this section. Section 6.3 introduces the synthesizable subset of Verilog `V0` which will be the target language for the rest of the chapter. Event and cycle semantics have been defined for this language by Gordon [Gor95, Gor98]. A semantics-based interpreted-code simulator is implemented in the Escher language based on this definition (Section 6.4). The results of experiments of applying the Escher partial evaluator to the simulator given a particular circuit design are reported in Section 6.5. Finally, in Section 6.6, the feasibility of this approach to automatic compiled simulation is evaluated.

6.1 Introduction to Hardware Description Languages

Hardware description languages (HDLs) provide a formal text-based syntax for describing electronic circuits and systems. HDLs allow the formalisation of the abstract behaviour of the device independent of the hardware structure. Therefore, the behaviour of the component is not affected by structural issues and vice-versa. This flexibility is essential for VLSI design; traditional graphic-based CAD tools became ineffectual with the onset of million-gate designs in the 1980s [McL87]. Since then, the development of layout and logic synthesis tools have accelerated the integration of HDLs in the typical design flow.

Two main hardware description languages are the focus of VLSI design methodology: VHDL and Verilog [Smi97]. Both languages are suitable for designing application-specific integrated circuits (ASICs) and simulation, both are supported by the major EDA vendors, and both have IEEE standards. Both support the hierarchical design of systems and different levels of abstraction in a single model. On the other hand, Verilog is a significantly simpler language than VHDL. The constructs

[1] Typical speedups for specialised event-driven simulators tend to be around 2 times that of the original program.

of Verilog are based on the C programming language, whereas VHDL is an Ada-based design language. Verilog lacks user-defined datatypes, design reusability using packages, and concurrent procedure calls, all of which can be found in VHDL. Therefore, VHDL is a more comprehensive language, but the simplicity of Verilog may be the cause of its growing popularity.

The aim of this work is to show that an interpreter based on defined semantics can be specialised to obtain an efficient simulator for a specific circuit design. In order to keep this chapter self-contained, the following section introduces the design process using HDLs. Verification is performed on several levels during the design flow; the simulation of the designs at the different levels of abstraction is the topic of Section 6.1.2.

6.1.1 Designing Hardware using HDLs

The typical design flow for an ASIC or a field-programmable gate array (FPGA) is shown in Figure 6.1 [Pal96].

The design process usually begins with the specification of the system and a description of its behaviour. The behavioural description can be modelled in an HDL. Modelling the behavioral description in a HDL allows system-level verification to confirm the overall functionality of the design. Furthermore, different algorithmic foundations of the design can be tested at this level before the specification is manually described at the register-transfer level (RTL).

The core of the design process is the development of the register-transfer level (RTL) descriptions of the components of the system. These models describe the flow of data through the digital circuits. Implementing these designs in an HDL facilitates the translation of the RTL modules to the gate-level designs; logic synthesis tools perform this transformation automatically.

Figure 6.1 presents a design flow in which the verification of the design is performed at several abstraction levels. The functional verification and testing of the design is typically achieved by regression testing, a process to ensure the functionality of the design is unchanged during its development. The RTL description is simulated to verify the correct data flow with respect to the given specification.

At the gate-level, logic verification of the circuit design is performed. Given a netlist representation of the design, a gate-level simulator emulates the behaviour of every gate. Typically, RTL simulation is one to two orders of magnitude faster than gate-level simulation [San]. Although gate-level simulators can be implemented efficiently, their performance is dependent on the number of gates in the design. An additional advantage of testing and validation at the functional level is the lack

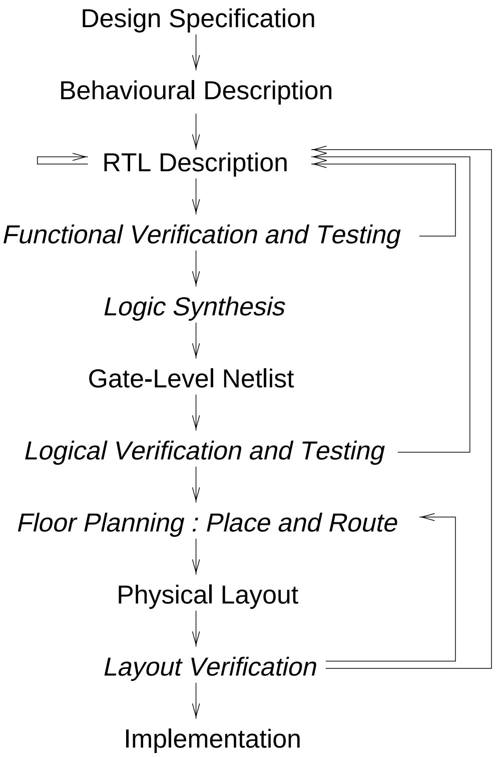


Figure 6.1: The typical design flow for digital circuit development from [Pal96]. The iteration of some of the steps are indicated by the looping arcs. Processes are noted in italic font; otherwise, the level of design representation is indicated.

of wasted effort of synthesising erroneous RTL descriptions. This should not be underestimated; synthesising complex designs takes a considerable amount of time.

Clearly, simulation of a HDL design at several levels of abstraction forms an important part of the design process. Modern simulation technology is reviewed in the next section, in order to survey the current trends in the industry.

6.1.2 The Simulation of HDL Descriptions

In this section, existing simulation technology for HDLs is reviewed. In particular, the types of simulators and their simulation speed are introduced, as these topics are relevant to the evaluation of the experiments with partial evaluation, presented later in Section 6.6.

There are two main types of simulators available: *event-driven* and *cycle-based* simulators.

Event-driven Simulation

In event-driven simulation, every active signal is calculated for every device in a given clock cycle. While this is a very thorough method of simulation of either behavioural, register-transfer level, gate or transistor representations, it suffers from poor performance in terms of speed.

Event-driven simulators can either be *compiled-code* or *interpreted-code* tools. Compiled-code takes a HDL description of a digital circuit design and compiles it into a more efficient structure in order to improve the simulation speed. For example, the NC-Verilog (Native Compiled) simulation tool developed by Cadence compiles Verilog modules into machine code. Of course, the simulator requires time to compile the code to the lower-level language. Alternatively, interpreted-code simulators offer the user the flexibility to modify the HDL descriptions without requiring recompilation of the simulator. This is an advantage during initial testing, but for the validation of the system, the slow speed of interpreted-code simulators can be prohibitive. The original Verilog simulator, Cadence’s Verilog-XL, is an example of an interpreted-code simulator.

Cycle-based Simulation

In cycle-based simulation, only the values at the end of a clock cycle are simulated. This means that the timing of the circuit is not exhibited in cycle-based simulation. This type of simulator is intended to be applied during the logic verification of the design; cycle-based simulators are faster than event-driven simulators, but usually their use has to be supplemented with a timing analyser to provide full validation of the design. They provide the speed of a compiled-code event-driven simulator with the flexibility of an interpreted-code simulator. Cadence's Cobra Cycle Simulator is an example of a cycle-based tool.

Therefore, in comparison, event-driven simulation evaluates the circuit design at every state change. This implicitly involves the simulation of the timing of the circuit. On the other hand, cycle-based simulation only demonstrates the logical design of the circuit; the state at the end of every clock cycle is returned by the simulation tool. In general, cycle-based simulators are more efficient than interpreted-code event-driven simulators, but compiled-code simulators tend to be the most efficient overall.

Simulation Speed

In the previous section, the types of simulators were compared in terms of their simulation speed. The overall speed of a simulator is a crucial issue for modern VLSI system development. As noted in [San], these systems usually require millions of simulation cycles in order to validate their design. Even a simulator performing a simulation a second will take far too long to achieve this task.

As discussed in Section 6.1.1, gate-level simulators, either event-driven or cycle-based, simulate the behaviour of each gate in the netlist. Therefore, the simulation time is positively related to the number of gates in the design; in fact, the simulation time increases exponentially [San].

Performing the simulation at higher levels of abstraction addresses this problem. Instead of simulating the activity of each gate in the netlist, RTL simulators perform the function of a number of gates. For example, consider the following descriptions of a 4-to-1 multiplexer from [Pal96]. The gate-level model of the circuit in Verilog is shown below:

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

output out;
input i0, i1, i2, i3, s1, s0;
wire s1n, s0n;
wire y0, y1, y2, y3;

not (s1n, s1);
not (s0n, s0);
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
or (out, y0, y1, y2, y3);

endmodule
```

The design consists of seven logic gates, and requires the definition of six internal wires. On the other hand, at the behavioural level, the design of a 4-to-1 multiplexer in Verilog is as follows:

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

output out;
input i0, i1, i2, i3, s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case({s1,s0})
   2’b00: out = i0;
   2’b01: out = i1;
   2’b10: out = i2;
   2’b11: out = i3;
   default: out = 1’bx;
  endcase
end

endmodule
```

The simulation results of the two modules are identical. However, simulating the behavioural model requires only the evaluation of a single case statement, while the gate-level simulation requires the activity of every gate to be enacted. Similarly, performing logical operations on buses can be simply expressed at the register-transfer level:

```
wire [7:0] out, in, in2;
assign out = in & in2;
```

Whereas, at the gate-level, the eight and-gates have to be simulated individually.

Experiments have shown that the gain in efficiency from using a compiled-code simulator rather than an interpreted-code simulator are highest at the behavioural level and RTL of abstraction [San]. Gate-level simulation can be performed adequately using interpreted-code simulation, since the behaviour of the gates can be calculated using efficient techniques, such as table look-ups. On the other hand, the behaviour of the devices at higher levels of abstraction are too complex to be implemented in such an efficient manner. The difference between compiled-code and interpreted-code simulation for the higher levels of abstraction can be as much as one to two orders of magnitude, whereas at the gate-level, there are usually no gains in speed between the two simulators.

6.2 Verilog IEEE Simulation Semantics

Verilog HDL, introduced in 1983 by Gateway Design Automation, is used by electronic designers for verification by simulation, timing analysis, and logic synthesis of VLSI digital circuits. As noted in Section 6.1, Verilog is a language based on the C programming language. However, while C is a sequential language, Verilog is a parallel language. Parallelism is essential for modelling the concurrent processes of hardware components. It is this feature of the language which complicates its defined semantics.

In this section, the scheduling semantics for the Verilog language are presented according to the IEEE Standard 1364 [IEE95]. This is intended to be a basic introduction to event-driven simulation of Verilog constructions; in general, the IEEE Standard is not considered a standard. Most designers employ the semantics of a particular simulator or tool as their standard [GG98].

6.2.1 Event Simulation

A Verilog module is composed of several threads of processes. A change of a value for a register or wire in a module is an *update event*. A process is defined to be sensitive to particular update events. When these signals change, the process are evaluated *non-deterministically*.

Some events have a defined evaluation order, and some have an indeterminate evaluation order. In the simulation semantics of the IEEE Standard, a stratified event queue orders the events according to their simulation time. Events are categorised into five types. They are ordered below according to their simulation order, with active events evaluated first during the simulation time.

- Active events: events which are simulated during the current simulation time;

- Inactive events: events which are simulated at the end of the current simulation time;
- Non-blocking assign updates: assignments that have been evaluated at a previous simulation time, but must be assigned during the current simulation time;
- Monitor events: any event which must be evaluated after active, inactive and non-blocking assign update events;
- Future events: events which are simulated at some future simulation time.

A simulation cycle is the evaluation of all active events. Non-determinism arises because active events are taken off the queue in no particular order. In addition, the evaluation of events may be interleaved; the simulator may decide to suspend a particular execution in order to simulate a different active event in the queue. The interleaving of event execution results in *race conditions*.

6.2.2 Race Conditions

Race conditions occur when two or more events have the same execution order and their evaluation order determines different behaviour of the design. Since the simulator non-deterministically selects the events to execute, the behaviour of the model will change depending on which event is selected first, but this selection strategy cannot be defined. An example of a race condition is shown below.

```
reg x;
initial x = 1;
initial x = 2;
```

At the end of simulation time 0, the simulator would be correct to return the value of either 1 or 2 for the register `x`. Different simulators return different results for such examples.

Clocked registers can also cause problems for simulation semantics [Bla]:

```
always @(posedge clk)    (1)
    q1 = d1;             (2)
always @(posedge clk)    (3)
    q2 = q1;             (4)
```

At the beginning of a clock cycle, the registers `q1` and `q2` may have the same or different values, depending on the order of the evaluation of the blocking assignments. Furthermore, the execution order during one clock cycle does not determine the execution order during a later clock cycle.

6.3 Semantics for Synthesizable Verilog

Defining a simulation semantics for Verilog is non-trivial, as the brief survey of the previous section illustrated. The non-determinism of the language and the ability of the simulator to interleave executions of events allow different behaviour of a design to be reported during and after each simulation time. In general, there is an overall problem with the definition of simulation semantics; the IEEE simulation semantics have been criticised as being too difficult to understand or implement, and most consider the particular semantics of the major simulation tools to be the industry standard [GG98].

These issues prompted the study of the event and cycle semantics for a subset of Verilog by Gordon [Gor95]. This language called `V0` was defined in [Gor95, Gor97] and later revised in [Gor98]. The event, trace, and cycle semantics are defined for the elements of this language. The event and cycle semantics are as defined in Section 6.1.2; trace semantics models the behaviour of a circuit at the end of each *simulation* time. Thus, like cycle-based simulation, individual events during a simulation cycle (§ 6.2) are not evaluated in trace semantics, but unlike the cycle semantics, the simulation does not depend on the clock cycle.

In this section, the syntax for the `V0` language is defined (§ 6.3.1) and the algorithm for the event-driven simulation of `V0` is presented (§ 6.3.2). The syntax and semantics of the language have been simplified in the new version of the draft paper [Gor98], but this research was undertaken before the latest edition was available.

6.3.1 Syntax of `V0`

In this section, the syntax of the `V0` language is defined according to [Gor95, Gor97]. Programs are a set of threads:

$$\begin{array}{l} \texttt{forever}\ \mathcal{S}_1 \\ \quad\vdots \\ \texttt{forever}\ \mathcal{S}_n \end{array}$$

These threads represent concurrent activities in the design. The $\mathcal{S}_1, \ldots \mathcal{S}_n$ are statements, composed of *expressions* ($\mathcal{E}$) and *timing controls*.

The syntax of expressions is not defined in detail. In general, the set of expressions contains variables $\mathcal{V}$, constants including high `1`, low `0` and undefined `X` (representing the Verilog constant `1'bx`) and expressions formed using binary operations over these expressions. The language `V0` does not account for the different datatypes of Verilog: a variable represents a wire if it occurs on the left hand side of a continuous assignment. Otherwise, it is a register.

Timing controls in `V0` have the form $@(\mathcal{T})$, where $\mathcal{T}$ is an *event expression* defined as follows:

$$\begin{array}{lcll}
\mathcal{T} & ::= & \mathcal{V} & \text{(Change of value)}\\
 & | & \texttt{posedge}\ \mathcal{V} & \text{(Positive edge)}\\
 & | & \texttt{negedge}\ \mathcal{V} & \text{(Negative edge)}\\
 & | & \mathcal{T}_1\ \texttt{or}\ \cdots\ \texttt{or}\ \mathcal{T}_n & \text{(Compound sensitivity list)}
\end{array}$$

Statements $\mathcal{S}$ are defined as follows. The variables $\mathcal{R}$ denote registers and $\mathcal{B}$ denote blocks.

$$\begin{array}{lcll}
\mathcal{S} & ::= & () & \text{(Empty statement)}\\
 & | & \mathcal{R}{=}\mathcal{E} & \text{(Blocking statement)}\\
 & | & \mathcal{R}{<=}\mathcal{E} & \text{(Non-blocking assignment)}\\
 & | & \texttt{begin}\{:\mathcal{B}\}\ \mathcal{S}_1;\ \cdots\ ;\mathcal{S}_n\ \texttt{end} & \text{(Sequencing block)}\\
 & | & \texttt{disable}\ \mathcal{B} & \text{(Disable statement)}\\
 & | & \texttt{if}\ (\mathcal{E})\ \mathcal{S}_1\ \{\texttt{else}\ \mathcal{S}_2\} & \text{(Conditional)}\\
 & | & \texttt{case}\ (\mathcal{E}) & \text{(Case statement)}\\
 & & \quad \mathcal{E}_1\ :\ \mathcal{S}_1 & \\
 & & \qquad \vdots & \\
 & & \quad \mathcal{E}_n\ :\ \mathcal{S}_n & \\
 & & \quad \{\texttt{default}:\mathcal{S}_{n+1}\} & \\
 & & \texttt{endcase} & \\
 & | & \texttt{while}\ (\mathcal{E})\ \mathcal{S} & \text{(While statement)}\\
 & | & \texttt{repeat}\ (n)\ \mathcal{S} & \text{(Repeat statement)}\\
 & | & \texttt{for}\ (\mathcal{R}_1 = \mathcal{E}_1;\ \mathcal{E};\ \mathcal{R}_2 = \mathcal{E}_2)\ \mathcal{S} & \text{(For statement)}\\
 & | & \texttt{forever}\ \mathcal{S} & \text{(Forever statement)}\\
 & | & @(\mathcal{T})\ \mathcal{S} & \text{(Timing control)}
\end{array}$$

Translating to Semantic Pseudo-code

The first step in the simulation process is the translation of the Verilog syntax to semantic pseudo-code [Gor97]. The pseudo-code consists only of the following six instructions:

$\mathcal{R} = \mathcal{E}$	blocking assignment
$\mathcal{R} <= \mathcal{E}$	non-blocking assignment
$@(\mathcal{T})$	timing control
$\texttt{go}\ n$	unconditional jump to instruction n
$\texttt{ifnot}\ \mathcal{E}\ \texttt{go}\ n$	conditional jump to instruction n
$\texttt{disable}\ \mathcal{B}$	disable block

The translation algorithm is shown below. The definition of the size of a statement $|S|$ is straightforward [Gor97]. In the following, the translation of the statement $\mathcal{S}$ is $[\![\mathcal{S}]\!]\ p$, where p is the index of the first instruction. A sequence of statements is denoted by $\langle \mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_N \rangle$ or $\mathcal{S}_1 \frown \langle \mathcal{S}_2, \ldots \mathcal{S}_N \rangle$. Finally, the expression $x[y \leftarrow z]$ denotes the replacement of y in x with z.

$$
\begin{array}{lcl}
[\![\mathcal{R} = \mathcal{E}]\!]\ p & = & \langle\ \mathcal{R} = \mathcal{E}\ \rangle \\
[\![\mathcal{R} <= \mathcal{E}]\!]\ p & = & \langle\ \mathcal{R} <= \mathcal{E}\ \rangle \\
[\![\langle\rangle]\!]\ p & = & \langle\rangle \\
[\![\langle \mathcal{S}_1,\ \mathcal{S}_2, \ldots, \mathcal{S}_n \rangle]\!]\ p & = & [\![\mathcal{S}_1]\!]\ p \frown [\![\langle \mathcal{S}_2, \ldots, \mathcal{S}_n \rangle]\!](p + |\mathcal{S}_1|) \\
\texttt{begin}\{:\mathcal{B}\}\mathcal{S}_1; \ldots;\ \mathcal{S}_n\texttt{end}]\!]\ p & = & [\![\langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle]\!]\ p\ [\texttt{disable}\ \mathcal{B}\ \leftarrow\ \texttt{go}\ p + |\langle \mathcal{S}_1, \ldots, \mathcal{S}_n \rangle|] \\
[\![\texttt{disable}\ \mathcal{B}]\!]\ p & = & \langle \texttt{disable}\ \mathcal{B}]\!] \\
[\![\texttt{if}\ (\mathcal{E})\ \mathcal{S}]\!]\ p & = & \langle \texttt{ifnot}\ \mathcal{E}\ \texttt{go}\ p + |\mathcal{S}| + 1 \rangle \frown [\![\mathcal{S}]\!](p + 1) \\
[\![\texttt{if}\ (\mathcal{E})\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2]\!]\ p & = & \langle \texttt{ifnot}\ \mathcal{E}\ \texttt{go}\ p + |\mathcal{S}_1| + 2 \rangle \\
 & & \quad \frown [\![\mathcal{S}_1]\!](p + 1) \\
 & & \quad \frown \langle \texttt{go}\ (p + |\mathcal{S}_1| + |\mathcal{S}_2| + 2) \\
 & & \quad \frown [\![\mathcal{S}_2]\!](p + |\mathcal{S}_1| + 2) \\
[\![\texttt{while}\ (\mathcal{E})\ \mathcal{S}]\!]\ p & = & \langle \texttt{ifnot}\ \mathcal{E}\ \texttt{go}\ p + |\mathcal{S}| + 2 \rangle \frown [\![\mathcal{S}]\!](p + 1) \frown \langle \texttt{go}\ p \rangle \\
[\![\texttt{forever}\ \mathcal{S}]\!]\ p & = & [\![\mathcal{S}]\!]\ p \frown \langle \texttt{go}\ p \rangle \\
[\![@(\mathcal{T})\ \mathcal{S}]\!]\ p & = & \langle @(\mathcal{T}) \rangle \frown [\![\mathcal{S}]\!](p + 1)
\end{array}
$$

The following example illustrates the translation of a statement to the semantic pseudo-code.

Example 6.3.1 Consider the following thread in `V0` :

```
forever
  begin
   @(posedge clk) total = data;
   @(posedge clk) total = total + data;
   @(posedge clk) total = total + data;
  end
```

This translates to the sequence of pseudo-code instructions shown below.

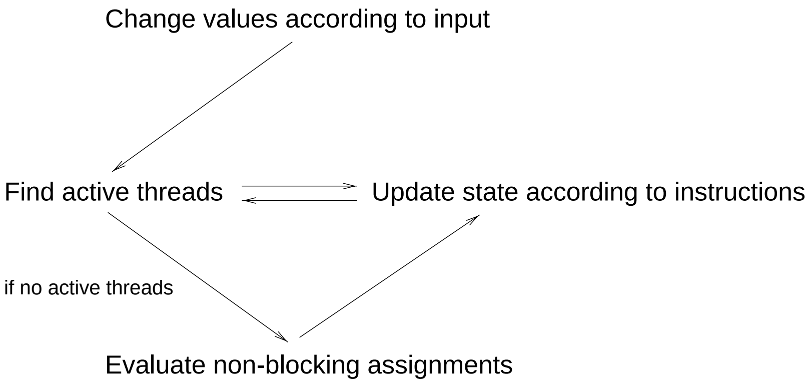

Change values according to input
Find active threads
Update state according to instructions
if no active threads
Evaluate non-blocking assignments

Figure 6.2: The flow of a simulation cycle for an event-based circuit simulator.

```
0:  @(posedge clk)
1:  total = data
2:  @(posedge clk)
3:  total = total + data
4:  @(posedge clk)
5:  total = total + data
6:  go 0
```

6.3.2 Event Semantics for `V0`

The event semantics are defined below. The simulation maintains a *state* consisting of the simulation time and an environment defining the values of the variables in the `V0` module and program counters.

Basically, a simulation cycle begins by updating the environment, which contains the values of the variables, according to the input defined for that simulation time. Then, threads are non-deterministically evaluated according to the new environment. For each thread, the timing control may or may not be satisfied by the new environment. If the timing control is satisfied, the active events on the thread are executed. These events may change the environment. If this happens, the timing controls on the threads are again examined to find more threads that are able to be evaluated. If no active threads exist, the non-blocking assignments are applied to the environment, and this environment is again used to find threads to evaluate. The simulation cycle ends when there are no active threads and no non-blocking assignments. This process is shown graphically in Figure 6.2.

Key to the event semantics is the definition of the satisfaction of a timing control; in the syntax of [Gor95], an event expression *fires* when it is satisfied, and this *enables* the thread. Otherwise, a thread is *waiting* if its current instruction is a timing control that cannot be satisfied. The notion of a firing event expression is defined below [Gor97].

$\mathcal{V}$ fires if the current value of $\mathcal{V}$ is different than the previous one;

`posedge` $\mathcal{V}$ fires if the current value of $\mathcal{V}$ is `1` and its previous value was not `1`;

`negedge` $\mathcal{V}$ fires if the current value of $\mathcal{V}$ is `0` and its previous value was not `0`;

$\mathcal{T}_1$ `or` ... `or` $\mathcal{T}_n$ fires if any $\mathcal{T}_i$, $1 \leq i \leq n$, fires.

The simulator begins at simulation time 0, where every variable has the value `X` (undefined). All program counters are initially set to 0. The event semantics defined below were adapted by the author from the original version in [Gor97]. The new version of the event semantics agrees with this formulation [Gor98], except non-blocking assignments are no longer considered to be an element of the `V0` language.

1 If there are no enabled threads then go to **3**, else non-deterministically choose an enabled thread and go to **2**.

2 Given an enabled thread, execute its current instruction as follows:

(a) $\mathcal{V} = \mathcal{E}$: change the $\mathcal{V}$ component of the state to the value $\mathcal{E}$ in the current state, note the changed variable $\mathcal{V}$, increment the program counter and then go to **2**;

(b) $\mathcal{V} <= \mathcal{E}$: add $\mathcal{V} <= \mathcal{E}'$ to the list of pending non-blocking assignments, where $\mathcal{E}'$ is the value of $\mathcal{E}$ in the current state (override any previously generated pending assignments to $\mathcal{V}$), increment the program counter and then go to **2**;

(c) `go` n: set the program counter to n and go to **2**;

(d) `ifnot` $\mathcal{E}$ `go` n: if $\mathcal{E}'$ is the value of $\mathcal{E}$ in the current state, then increment the program counter, otherwise set it to n, then go to **2**.

(e) $@(\ \mathcal{T}\ )$: go to **1**.

3 Increment the program counters of all threads whose current instruction is a timing control that fires and then go to **1**.

4 If there are no pending non-blocking assignments then go to **5**, else execute all pending non-blocking assignments (in any order and overriding any assignments in the state) then go to **3**.

5 Increment the simulation time, update the state with any changes from the inputs and go to **3**.

Note that in the above definition, race conditions can occur, since enabled threads are selected non-deterministically, while the interleaving of executing threads is not possible. A thread, once enabled, is evaluated until a timing control is reached. As an example, the following example follows the simulation of a simple adder for one simulation time unit.

Example 6.3.2 Consider the following combinational adder from [Gor97]:

```
forever @(b or c) a = b + c;
```

The corresponding pseudo-code instructions are the following:

```
 0:  @(b or c)
 1:  a = b + c
 2:  go 0
```

At the start of the simulation, the environment is defined as:
$\{\texttt{pc} \rightarrow 0, \texttt{a} \rightarrow \texttt{X}, \texttt{b} \rightarrow \texttt{X}, \texttt{c} \rightarrow \texttt{X}\}$

Suppose the value of `b` changes to `1` and `c` changes to `3` at simulation time 1. The new state is:
$\{\texttt{pc} \rightarrow 0, \texttt{a} \rightarrow \texttt{X}, \texttt{b} \rightarrow \texttt{1}, \texttt{c} \rightarrow \texttt{3}\}$

The timing control of the thread `pc` is satisfied by the change of the value of `b`. Therefore, the instruction counter for the thread is incremented: $\{\texttt{pc} \rightarrow 1, \texttt{a} \rightarrow \texttt{X}, \texttt{b} \rightarrow \texttt{1}, \texttt{c} \rightarrow \texttt{3}\}$

The blocking assignment corresponding to instruction 1 of the thread is evaluated according to the current state. This results in the new state: $\{\texttt{pc} \rightarrow 2, \texttt{a} \rightarrow \texttt{4}, \texttt{b} \rightarrow \texttt{1}, \texttt{c} \rightarrow \texttt{3}\}$

Finally, the unconditional jump according to instruction 2 resets the thread counter `pc` in the environment: $\{\texttt{pc} \rightarrow 0, \texttt{a} \rightarrow \texttt{4}, \texttt{b} \rightarrow \texttt{1}, \texttt{c} \rightarrow \texttt{3}\}$

The timing control in instruction 0 is not satisfied. Since there are no non-blocking assignments pending, the simulation cycle ends.

6.4 Semantics-based Simulator for V0

The event semantics for the V0 language were defined in the previous section. Based on this definition, an interpreted-code simulator for the V0 subset of synthesizable Verilog was implemented in the Escher functional logic language. The simulator is made up of seventy schema statements which define forty-two functions. These functions evaluate the components of a design expressed in the pseudo-code of the semantic definition.

Corresponding with the basic instructions for the pseudo-code (§ 6.3.1), the program has a defined type `Code` with the following free functions defined for packaging elements to type `Code`.

`V : String -> Code`	(variables)
`I : Integer -> Code`	(integers)
`X : One -> Code`	(undefined `1’bx`)
`Blk :  String * Code -> Code`	(blocking assignment)
`NBk :  String * Code -> Code`	(non-blocking assignment)
`Go :  Integer -> Code`	(unconditional jump)
`ING : Code * Integer -> Code`	(conditional jump)
`At :  Code -> Code`	(timing control)
`PosE, NegE : Code -> Code`	(event expressions)

A state contains two lists: a list of current values for the variables and program counters, and a list of current changes and non-blocking assignments pending for this simulation cycle.

The core of the evaluation in the program is performed by the `Eval` function, which performs any active events in the thread on the state, and returns the state upon encountering a timing control.

```
Eval(Blk(v,exp),m,th,state) =>
  Eval(XCode(Inst(th,+(m,1))),+(m,1),th,
   <[<v,EvalExp(exp,Fst(state))> | Fst(state)>,
    [<"ch",V(v)> | Snd(state)]>.

Eval(NBk(v,exp),m,th,state) =>
  Eval(XCode(Inst(th,m+1)),m+1,th,
   <Fst(state),[<"nbk",EvalExpr(exp,Fst(state))> | Snd(state)]>).

Eval(Go(n),m,th,state) =>
  Eval(XCode(Inst(th,n)),n,th,state).

Eval(ING(cond,n),m,th,state) =>
   IF    EvalCond(cond,Fst(state))
   THEN  Eval(XCode(Inst(th,m+1)),m+1,th,state)
   ELSE  Eval(XCode(Inst(th,n)),n,th,state)

Eval(At(t),m,th,state) =>
  <Store(Fst(state),Name(th),I(m)), Snd(state)>.
```

The threads of a Verilog design are stored in a list; these are not non-deterministically chosen in this implementation, but the order of the list is changed during the simulation. The functions to perform the firing tests were implemented as Boolean functions; for example, testing if a posedge event expression fires implemented by TestPos defined below.

```
TestPos(x, state) =>
  SOME [y,z] (LookupTwo(state, x, y, z) &
              y = I(1) & z ~= I(0)).
```

The main function is also a Boolean function of type:

```
FUNCTION   Sim: Integer * List(Thread) * List(Integer,(String,Code)) *
                List(List((String,Code))) -> Boolean.
```

The first argument to Sim is the maximum simulation time, the second is the circuit design, the third is the input, and the fourth is the list of states at each simulation time from time 0. The functions were implemented as type Boolean because the implementation was clearer, easier, and more efficient than the functional style implementation (see Section 5.7 for a discussion on implementing interpreters in Escher).

Both the Map and Filter higher-order functions are used in the implementation. The store function is tail-recursive, and the lookup function is as defined in Section 5.7.

6.5 Experiments with Simulating VO Modules

The implementation of the interpreted-code, event-driven simulator for modules in the VO language (§ 6.4) was partially evaluated with respect to several basic digital circuit designs.

Asynch single	combinational adder
Asynch disjoint	two disjoint adders
Asynch interact	two interacting adders
Asynch latch	latch inference example
One flipflop	one flip-flop
Two flipflops	two flip-flops in series
Flipflop & multiplexer	flip-flop with separate multiplexer
Flipflop & increment	flip-flop with built-in incrementer
Moore machine	simple Moore machine
State machine (Im)	Synopsys state machine from [Syn96]

	Simulator		Residual Program			Speedup	
	reds	time (ms)	reds	time (ms)	size	reds	time
Asynch single	2210	113660	1599	46120	280	1.38	2.46
Asynch disjoint	3977	291290	2872	146150	620	1.38	1.99
Asynch interact	4673	391250	2755	110650	620	1.70	3.54
Asynch latch	2187	131150	1313	32690	385	1.67	4.01
One flipflop	1985	113580	1410	35710	436	1.41	3.18
Two flipflops	3563	314670	2604	105300	778	1.37	2.99
Flipflop & multiplexer	5516	744920	3432	183340	808	1.60	4.06
Flipflop & increment	2005	118440	1416	36030	448	1.41	3.29
Moore machine	2745	315020	1606	70160	727	1.71	4.49
State machine (Im)	3253	568740	2212	172345	1352	1.47	3.30
Average Speedup						1.51	3.33

Table 6.1: Results of applying constraint-based partial evaluation to the specialisation of an interpreted-code, event-driven simulator for the V0 HDL. The original simulator contained 70 schema statements. The speedup is calculated for the run-times of the original and residual programs given the input data and the dynamic data respectively [JGS93].

Each of these examples were translated to the pseudo-code instruction language defined in Section 6.3.1. The Escher implementation of the V0 interpreter was partially evaluated with respect to each of these example components. The speedup for the resulting residual program in each case is shown in Table 6.1.

A few observations can be made from the results of the experiments. Firstly, the size of the residual program is directly related to the number of threads or the number of conditional jumps in the V0 component.

Twenty seven simulators were written in Escher in order to test the effect of the program structure on the quality of the residual program. Since all of the values of variables had to be recorded in the correct order during a simulation cycle, it was not possible to implement the interpreter in a traditional style. The environment had to be maintained as a list of ordered pairs, representing the variables and their value changes during the cycle. The input is undefined at compile-time. Since these input values must be recorded at the beginning of the sequence of changes on the environment, the environment is a variable at compile-time.

Overall, as was shown in Section 5.7, the logical-style implementation of the interpreter specialised the best under these circumstances. The residual programs suffered from many versions of simple functions such as Fst or Head. This resulted from the use of the tree structure in the algorithm. While the m-trees allowed greater polyvariance for the main functions, the level of polyvariance for the basic functions was too great.

6.6 Discussion and Summary

The previous research in this area includes [AWS91]; this report documents the specialisation of a cycle-based simulator for gate-level designs. The cycle-based simulation approach was chosen over an event-driven tool because the code size of the residual programs obtained by the specialisation of the event-driven tools was too large. In this work, the aim was to attempt the specialisation of an event-based interpreted-code simulator for high-level HDL designs. It was hoped that moving from the gate-level, where the speed of the simulator increases exponentially according to the number of gates in the netlist, would permit the controlled partial evaluation of the simulator.

In this chapter, it was shown that such an approach is feasible and shows a consistent speedup across a number of digital hardware component designs. However, further work is necessary to ensure the code size of the residual program is kept to a minimum. It may be the case that the use of the m-tree structure offers too much polyvariance; if so, a simple set-based partial evaluator might offer a solution.

Chapter 7

Summary and Further Work

Methodologies such as modular programming or programming in declarative languages facilitate the development of quality software products. However, improving the development lifecycle in such a way tends to have an adverse effect on the efficiency of the end product. By partial evaluation, it is possible to automatically improve the efficiency of programs by optimising away any unnecessary generality.

The functional logic programming paradigm is a recent development in the declarative programming domain. Functional logic languages offer users a highly expressive and flexible programming environment, but the advanced features demand greater computation time. The aim of this thesis was to develop a procedure for the partial evaluation of rewriting-based functional logic languages. The particular language of interest was the Escher language, which extends the Haskell functional language by providing logical variables and permitting partially instantiated run-time calls.

A further aim of this thesis was to study the effect of improved information representation in the transformation in terms of the quality of the residual programs. A partial evaluator must decide which program expressions should be reduced in order to generate the most efficient residual program. Theoretically, increasing the information available to the partial evaluator should improve this decision-making step. In turn, this should have a positive effect on the efficiency and size of the residual programs. The algorithm presented in this thesis used constraint solving to provide advanced information representation in an *automatic* context. For some programs, the representation of otherwise disregarded data noticeably improved the level of specialisation. This was studied in the context of other well-established program transformation procedures, in addition to the constraint-based approach described in this thesis.

Finally, the partial evaluator was applied to the simulation of high-level designs formalised in the Verilog hardware description language. The goal was to improve the efficiency of an interpreted-code simulator by automatically specialising it with respect to a particular circuit design. An event-based interpreted code simulator was implemented in Escher. This program is based on the event semantics defined for a synthesizable subset of Verilog [Gor95]. The simulator was partially evaluated with respect to a circuit design in order to remove the interpretive overhead. Results from the specialisation of the simulator with respect to several digital circuit designs showed an average speedup of nearly four times compared to the interpreted simulation. This is larger than reported previously for event-driven simulator specialisation, but is not as dramatic as the speedups that are possible for using cycle-based simulation tools as input [AWS91].

This chapter contains a summary of the key contributions of this thesis (§ 7.1). Future directions for research in partial evaluation related to the work described in this thesis are discussed in Section 7.2.

7.1 Summary

The aim of this thesis was to develop an advanced partial evaluator for rewriting-based functional logic programs. Methods for the partial evaluation of rewriting-based functional logic programs were not reported previous to the work described in this thesis. The focus language was Escher, an extension of Haskell offering set abstraction, logical variables, built-in Boolean operators, and a computational mechanism permitting the evaluation of partially-instantiated terms. The target language of this thesis, the *E* language, is a subset of the original Escher language; the syntax and semantics of *E* were defined in Chapter 2.

The solution to this problem was inspired by the ability of the language to rewrite partially instantiated terms. The data fixed at specialisation-time are represented in a term, while the run-time data are represented by variables; therefore, the term passed as input is partially instantiated. The theoretical framework of the partial evaluation procedure was adapted from partial deduction of logic programs. This framework was formalised in Chapter 3.

7.1.1 A Theoretical Framework for Partial Evaluation

In this thesis, a theoretical framework for the partial evaluation of rewriting-based functional logic programs was presented. Like the approach to partial deduction, a program is specialised with respect to a set of terms.

A residual definition is generated for each term in the set. The residual definition of a term having a non-trivial computation in the program is a resultant for that term, a partial computation converted into a rewrite rule. On the other hand, terms which are not reducible in the program must be *restarted*. Restart terms are the minimal instantiations of a term having a non-trivial definition in the program. This permits the construction of resultants for each of the *restart* terms for the term, thus constructing the residual definition.

A renaming operation was required in order to convert the resultants to schema statements in the language. This renaming operation incorporated a function to order the elements of the set of terms in order to make the renaming deterministic. The partial evaluation of a program with respect to a set of terms was defined as follows: for each element of the set of terms, generate a residual definition according to the schema statements of the object program. The union of the renamed set of rewrite rules is the resulting *residual* program.

Key to any program transformation is the proof that it is semantics-preserving:

- A program P reduces a term t in finitely-many steps to t' if and only if the specialised program P_A reduces the translated term t in finitely-many steps to t', or
- The computation of t in P is infinite if and only if the computation of the translated t in P_A is infinite.

The partial evaluation procedure formalised in this section was shown to be correct for several domains of programs and terms.

7.1.2 An Algorithm for Constraint-based Partial Evaluation

The theoretical foundations for the partial evaluation procedure developed in Chapter 3 provided a basis from which the algorithm was constructed. The algorithm describes a fully-automatic partial evaluator for rewriting-based functional logic programs. The partial evaluator takes as input a program, a term containing the fixed input data, and an initial set of constraints describing the environment.

The algorithm manipulates a tree structure containing the set of terms of the partial evaluation. The tree structure explicitly relates terms encountered during the transformation. This results in more precise residual programs, as unrelated terms are not generalised. A tree structure is also necessary to record the branching points in the partial evaluation; these are the junctions where the restart step has been applied. Each restart term is added as a child to the tree, and its computation in Escher is implicitly recorded.

The constraint solving operations allowed the pruning of branches in the tree structure. Information from conditional expressions that is representable in the constraint domains of the partial evaluator is stored in a set of constraints associated with the individual term. This information is used not only in the computation of restart terms, but also to eliminate unreachable terms in conditional expressions. The advanced specialisation of conditional expressions is particularly important for rewriting-based functional logic languages, as total functions with infinite domains cannot perform tests on arguments without a conditional expression in the body[1].

As in partial deduction, the control of the algorithm is partitioned into local control and global control. Local control ensures the finite unfolding of terms during the partial evaluation. The local control of this algorithm imposes a syntactic ordering on the redexes selected during a computation. If the ordering is violated, the computation stops. Redexes are identified using an indexing in order to improve the precision of the local control. The global control of the algorithm guarantees the finiteness of the tree structure constructed during the partial evaluation. The terms of a branch are ordered to ensure infinite branches cannot result without a violation of the ordering. The constraints associated with each term are widened, while the terms on a branch are generalised using the most specific safe generalisation.

A procedure to extract the residual program from the tree structure completed the definition of the algorithm. Extensions of the algorithm were presented at the end of Chapter 4. These extensions include the advanced handling of Boolean expressions, specialising nested conditional statements, improving the specialisation of higher-order functions using annotations on functional variables, and several post-processing operations.

7.1.3 A Partial Evaluator for the Escher Language

Chapter 5 introduced the implementation of the constraint-based partial evaluator for Escher programs. The implementation of the Escher language used in this research is a prototype written in the Gödel logic programming language. Since the Escher language could not support the implementation of the partial evaluator, the specialiser was developed in the Gödel language.

The structure of the algorithm (Chapter 4) is mirrored in the architecture of the Escher partial evalu-

[1] This results from the restriction to pattern and instance non-overlapping programs. For example, the function `F(x) => IF x = 1 THEN 2 ELSE F(x-1)` cannot be defined without the conditional expression, since the program containing the statement heads `F(1), F(x)` violates the restriction.

ator. Three main modules form the core of the system. These modules are responsible for the three main operations of the algorithm: extending the tree data structure, pruning the tree, and extracting the residual program from the tree. Constraint handling rules are implemented in Escher to provide simple constraint solving; advanced constraint solving is performed by built-in solvers. Furthermore, the implementation includes the extensions described in Chapter 4, including the advanced specialisation of Boolean expressions, specialisation of nested conditionals, and the post-processing operations.

The Escher constraint-based partial evaluator was tested using a set of benchmark programs. The partial evaluator passes the KMP-test, the conversion of a naive pattern matching program to a Knuth-Morris-Pratt pattern matcher. The constraint-based partial evaluator was also shown to be able to perform deforestation, the elimination of intermediate data structures from arbitrary higher-order rewriting-based functional logic programs. Several deforestation examples demonstrated the necessity for a defined ancestor selection strategy (§ 5.3). A term may be introduced on a branch of the tree structure which embeds more than one ancestor term stored on that branch. Typically, one of these ancestor terms is a better choice for generalisation than the others in terms of the resulting quality of the residual program. In this thesis, a metric for selecting ancestors was defined; however, a methodology for selecting the best ancestor is still an open problem, which could benefit from future research.

In particular, the specialisation of the 91-function example demonstrated the advantage of advanced information propagation by constraints. While the constraint-based partial evaluator could not generate the optimal program obtained by generalized partial computation, the residual program had over 200 times less reductions than the original program for the input `90`; after this value, timings for the original program could not be obtained.

Finally, in Chapter 5, a study of the effect of program structure was performed. In particular, a simple interpreter for an imperative language was specialised with respect to a program to compute the power of a number. This example demonstrated the wide variation between the residual programs based on the implementation of the original program. The implementation of the function to store values of the variables in the environment had a direct effect on the quality of the residual program. Further work in this area should include the detailed study of the relationship between program structure and the quality of the residual program. The flexibility of rewriting-based functional logic languages makes this an important issue; the partial evaluator must be able to specialise programs written in a functional style or a logical style equally well. This subject has been studied in the context of functional program specialisation before, particularly for the Mix partial evaluator [JGS93, Jon96b] and deforestation [CK96].

7.1.4 Semantics-based Digital Hardware Simulation

The implementation of the constraint-based partial evaluator was applied to the problem of digital circuit simulation. Hardware description languages (HDLs) allow the independent design of the behaviour and the structure of the circuit in a high-level language.

Although model checking and formal verification are becoming better accepted, most of the verification and testing of a design is performed by simulating the circuit on a given test bench and evaluating the results. In general, there are two types of circuit simulators: event-driven and cycle-based. Event-driven simulators exhibit every change of an active signal of the circuit. On the other hand, cycle-based simulators report the values of the signals at the end of each clock cycle. Thus, timing is abstracted away in the cycle-based simulation, but a cycle-based interpreted-code simulator is more efficient than one that is event-driven. On the other hand, event-driven *compiled-code* simulators are more efficient than either interpreted-code simulator. In particular, the high levels of HDLs (behavioural, register transfer) suffer from poor performance when interpreted-code simulation is used.

Therefore, the aim of this work was the automatic specialisation of an event-driven interpreted-code simulator with respect to a register-transfer or behavioural level HDL design to generate a more efficient simulator. The simulation semantics are complicated by the non-determinacy of the language. The event semantics for a subset of synthesizable Verilog was defined in [Gor95]. A simulator was implemented in the Escher language based on the definition in [Gor95]. The Escher constraint-based partial evaluator was applied to the simulator with various benchmark designs supplied as input. Specialisation in this case results in simulators with gains in efficiency of up to four times over the original program.

Previous work noted the difficulty of specialising event-driven simulators [AWS91]. Even in these experiments, the code size of the residual programs was considerably larger than the original program. On the other hand, most of these schema statements defined simple “packaging” functions that could be eliminated by a post-processing step.

7.2 Future Work

This thesis documents the study of the specialisation of rewriting-based functional logic languages. Unfortunately, the prototype Escher system did not permit the implementation of the partial evaluator in Escher. A practical implementation of Escher is currently under development. It would be interesting to study the self-applicability of the constraint-based partial evaluator by implementing it in the Escher language at some future date.

In the research performed related to this thesis, the author investigated the possibility of animating Z specifications by the specialisation of a Z interpreter, in the same manner as the Verilog simulation (Chapter 6). Unfortunately, the program synthesis required by this application is not possible by constraint-based partial evaluation. Therefore, this application was limited to Z specifications written in an algorithmic style; however, Z specifications are typically *not* written in an algorithmic style [HJ89]. This was regarded as too limiting to be of any practical use. On the other hand, the simulation of HDLs lends itself to the application of partial evaluation. The research described in this thesis should be extended to test the specialisation using more complex designs in a more robust language.

Outside the scope of functional logic program specialisation and Verilog simulation, there are three topics in partial evaluation that deserve future study. Firstly, in Section 7.2.1, the problem of controlling the size of the residual programs generated by automatic partial evaluators is discussed. Recent work in resource-bounded partial evaluation should be investigated in the context of on-line program specialisers. The investigation of the “power” of constraint-based partial evaluation motivates a future study of extending automatic program specialisers (§ 7.2.2). To what extent is automation really necessary? Can modern automatic theorem proving be incorporated into an automatic technique to yield better quality residual programs? Finally, in Section 7.2.3, initial research in language-independent specialisation is reviewed, as reported in the published papers [GL98, GL96].

7.2.1 Accounting for Residual Program Size

Controlling the size of residual programs remains a problem for automatic partial evaluators. Most partial evaluation procedures aim to reduce as much code as possible, while minimalising the amount of residualised code. However, in some cases, this approach can have a negative effect on the efficiency of the residual program. Blindly unfolding all the possible expressions typically results in a residual program too large to be of any use[2].

The concept of resource-bounded program specialisation was introduced in [DHM96] and has recently been addressed in terms of off-line partial evaluation in [Deb97]. These authors argue that practical partial evaluators should take into account machine-specific resources, such as the size of the information cache and registers, during specialisation. For example, as shown in [Deb97], if a

[2] Subsequent compilation may suffer, for example.

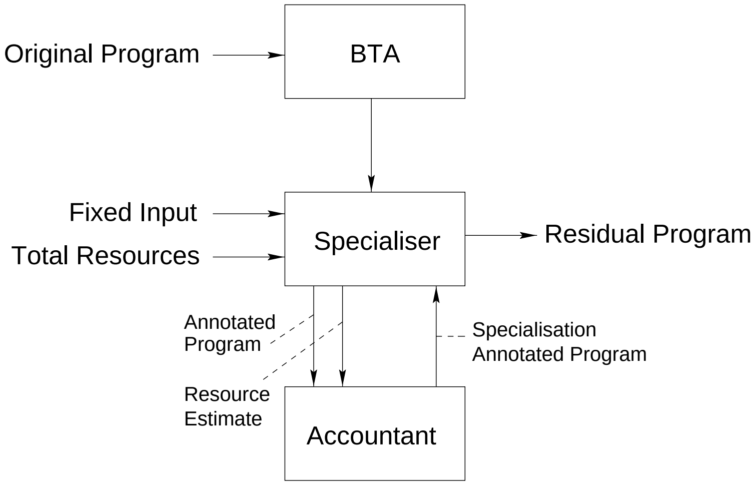


Figure 7.1: The structure of an off-line partial evaluator incorporating an accountant to control residual program size.

convolution-like program which takes two vectors of length n as input is specialised with respect to one of the input vectors, the quality of the residual program depends on the value of n. If $n \geq 7000$, the residual program is actually slower than the original program. The size of the body of the main loop grows linearly as n increases, and problems arise when the loop is too large to fit in the instruction cache of the machine. Debray notes in [Deb97] that performance tends to decrease as the size of the fixed input increases. Therefore, practical partial evaluation should incorporate a mechanism for *accounting* for the residual program size.

In this work, Debray defines a partial evaluator which incorporates an accountant component. The structure of this system is illustrated in Figure 7.1. The accountant interacts directly with the specialiser. The original program is annotated as usual by the binding time analyser. Then, the annotated program is passed to the accountant. The accountant calculates the cost and benefit of reducing the annotated expressions in terms of the fixed machine resources. Then, the accountant may convert some of the “reduce” annotations to “residualise” before passing the program to the specialiser for evaluation. After one iteration, the specialiser returns the new program code and the revised estimate of the machine resources to the accountant for analysis. The accountant may change some of the annotations based on the new data. This process continues until all the program expressions have been specialised.

Of course, determining the optimal cost versus benefit partition of the program code is an NP-complete problem. On the other hand, a heuristic for calculating the cost and benefit of program expressions is presented in [Deb97].

At a first glance, it would seem straightforward to incorporate such an accountant in the on-line partial evaluation algorithm as described in Chapter 4. In terms of the main algorithm from Definition 4.1.2, the accountant would be added to the generalisation operation α. After the folding and generalisation functions have been applied to the m-tree, the cost of specialising each term in the tree is calculated with respect to the available resources ($R - U_i$), where U_i is the amount of resources used during the i^{th} iteration. This will have to be evaluated with respect to the cost of including the definition of each user-defined function occurring in the term in the residual program, since the term must be split if the accountant does not permit its reduction. The main algorithm of the procedure in this case may have the following structure:

```
Input: a program P, a term t, set of constraints $C$, and machine-specific resource data R.
    $U_i := 0;$

repeat
    $(\mu_{i+1}, U_{i+1}) = \alpha(extend(\mu_i, P_0), U_i, R);$
    $i = i + 1;$
until $\mu_i = \mu_{i-1}$
return $\mu_i$
$P' := \mathcal{R}_\sigma(\mu_i).$
```

The m-tree structure in the algorithm for constraint-based partial evaluation (Chapter 4) also causes redundancy in the residual programs by generating several residual definitions of the same term. There is no referencing between individual branches of the m-tree; therefore, if there are ten branches containing the same term, ten different copies of the same residual definition will occur in the residual program. A naive approach to solving this problem might be to store the terms in a binary tree with their tree index. A check is performed during the generalisation step; if the term already occurs in the tree, the leaf is marked and the reference to the index is stored. This would be acceptable if nodes of the tree were not destroyed during the generalisation process. The referencing and dereferencing of the terms would require a great amount of computation time. In addition, the constraint sets complicate the matter further; a check would have to be performed that the sets are logically equivalent before the terms could be related.

7.2.2 Improving Decision-Making in Partial Evaluation

> As long as your methods are supposed to be good for proving everything, they're not likely to be good for proving anything. — S. Papert

Two characteristics of program specialisers seem to conflict: the quality of the decision-making of the procedure and its automation. For example, consider again the 91-function specialisation described in Chapter 5. Many automatic established techniques, including partial deduction and Mix-based partial evaluation, cannot obtain any specialisation of this simple program. On the other hand, the user-driven generalized partial computation, as defined in [FN88], achieves the optimal specialisation of this program. It seems that the automatic generalized partial deduction as defined in [Tak91, Tak92] cannot obtain the specialisation.

Three questions naturally arise from this study:

- How much specialisation is actually possible by an automatic program transformation procedure? Can the specialisation be improved in an automatic context?
- To what extent can user-driven program specialisation techniques be used in practical applications?

Pettorossi and Proietti have extended partial deduction in an automatic context by incorporating versions of the definition and the folding rule from the unfold/fold program transformation framework [PPR97]. The inclusion of these rules has a dramatic effect on the specialisation power of partial deduction. In particular, it was noted earlier in Chapter 5 that partial deduction extended with some user-defined rules could generate a KMP-pattern matching program from the simple specification of the naive pattern matcher shown below [PPR97]:

```
match(P,S) :- append(_,P,X), append(X,_,S).
```

Although traditional partial evaluation can obtain some efficiency gain for the above program, the residual program generated by traditional partial deduction does not pass the KMP-test.

Partial evaluation and the unfold/fold program transformation system were developed originally to perform two different tasks: in the first case, program specialisation, and in the second, program synthesis (Chapter 2). While partial evaluation is an automatic procedure, the unfold/fold program transformation requires interaction from the user. However, it is clear that the integration of some of the unfold/fold methodology in the context of partial evaluation can improve the quality of the residual programs.

Pettorossi and Proietti support the alternative view: the core of “new generation” program specialisation techniques should be the unfold/fold program transformation framework [PP98]. It seems likely that simply improving the flexibility of partial evaluators to handle, for example, the introduction of user-defined rules or the incorporation of a library of rules would have a noticeable effect on the quality of residual programs.

In this context, it may be possible to integrate automatic theorem proving technology. One should note the actual meaning of “automatic” theorem proving; even in these techniques, interaction from the user may be required to adjust the weightings of strategies or input the theorems in a particular order. Automated theorem proving generally can be classified into two disciplines: search-based theorem proving founded on resolution [Rob65] (such as Argonne’s OTTER system [Wos96]), and Bledsoe’s knowledge-based theorem proving [Ble77]. New directions in automated theorem proving, such as proof planning [Ker97], may provide program specialisers with the ability to generate rules during the transformation. The field of automatic theorem proving should be watched closely; it has been ten years since the original paper on generalized partial computation, but little research on the incorporation of automatic theorem proving in partial evaluation has been published since.

7.2.3 Language-Independent Program Specialisation

Most of the material of this section originates from initial work carried out by the author with Gallagher [GL96, GL98]. The aim of the research is to construct a language-independent framework for program specialisation. The core of the technique is based on the notion of a *trace*, the finite, terminating computation from an initial environment in a program.

Let Trace(Init,P) be the set of finite transitions, or *traces*, resulting from some initial environment Init and program P. Computations can be labelled in order to represent traces in a language of finite sequences [GL98]. A particularly convenient way to generate the word describing a computation is by means of trace terms [GL96]. For example, consider the logic program defining the append predicate:

```
append([],y,y).
append([x|xs],y,[x|z]) :- append(xs,y,z).
```

An extra argument is added to the heads of the statements and each literal in the body. The process for adding trace terms to a program is defined in [GL96]. This results in the program:

```
append([],y,y,a1).
append([x|xs],y,[x|z],a2(w)) :- append(xs,y,z,w).
```

The computation of the goal `append(A,B,[1,2],T)` returns the trace term {`a1, a2(a1), a2(a2(a1))`} in the fourth argument `T` of `append`. The same trace results from the computation of any goal with a two element list in the last argument of `append` and variable terms in the first two arguments. The partial evaluator can safely generate a single residual definition for these terms, since they share a similar state [GL98].

Clearly, in a functional program, matching prevents the embedding of trace terms in the heads of the statements as above. Instead, the trace can be represented with a list of identifiers (the determinacy of the language ensures only one computation exists).

For example, examine the program which finds the last element of a list:

```
last (x:xs) => check xs x        <last1/1>
check [] x  => x                 <check1/0>
check (z:zs) x => check zs z     <check2/1>
```

Three statement identifiers `last1/1`, `check1/0`, and `check2/1` are assigned to the statements in the program. Then, the word describing the trace of `last (A:B)` is `[last1/1, check2/1, check1/0]`.

The related concepts of neighbourhoods [Tur88], characteristic trees [GB91], characteristic atoms [Leu95, LM96] and trace terms [GL96] allow the representation and use of properties of the computation during the specialisation, for example, to improve the control of the procedure [GL98]. These are called *trace abstractions*, as they only represent the information necessary for the specialiser; this may not include the full representation of the trace in the program.

Computing trace grammars to perform language-independent specialisation has only been initially addressed in [GL96, GL98]. Further work is necessary to identify an approximation for the language of the trace representations, in order to guarantee the termination of the procedure.

Appendix A

Appendix

A.1 Benchmark Programs

These benchmark examples are referenced in Table 5.4 in Chapter 5. In the following examples, the CONSTRUCT declarations and some FUNCTION declarations of free functions have been omitted for the sake of brevity.

A.1.1 SumSqUpToTree

This benchmark is a member of the Dozens of Problems for Partial Deduction (DPPD) [Leu]. It is related to the original Summing Squares benchmark of Wadler [Wad90], but the lack of direct reference to integers in the following original programs allows complete deforestation of this program by constraint-based partial evaluation.

```
FUNCTION  Leaf: Integer -> Tree.
FUNCTION  Branch: Tree * Tree -> Tree.

FUNCTION  SumTRSqTR : Tree -> Integer.
SumTRSqTR(xt) =>
   SumTR(SquareTR(xt)).

FUNCTION  Sq : Integer -> Integer.
Sq(n) => n * n.

FUNCTION  SumTR : Tree -> Integer.
SumTR(Leaf(x)) => x.
SumTR(Branch(xt,yt)) => SumTR(xt) + SumTR(yt).

FUNCTION  SquareTR : Tree -> Tree.
SquareTR(Leaf(x)) => Leaf(Sq(x)).
SquareTR(Branch(xt,yt)) =>
   Branch(SquareTR(xt), SquareTR(yt))).
```

Partially evaluating the above program with respect to the term `SumTRSqTR(x)` results in the following residual program.

```
FUNCTION  Ans : Tree -> Integer.
Ans(xt) => FN_SP1(xt).

FUNCTION  FN_SP1 : Tree -> Integer.
FN_SP1(Leaf(x_2)) => (x_2 * x_2).
FN_SP1(Branch(xt_2, yt_3)) =>
    FN_SP1(xt_2) + FN_SP1(yt_3).
```

A.1.2 Queens

The following program is the full “Listful Queens” implementation, adapted from [Mar96]. The specialisation of this program is the subject of Section 5.3.3.

```
FUNCTION  Queens : Integer -> List(List(Integer)).
Queens(n) =>
  IF    n = 0
  THEN  0
  ELSE  Map(Filter(ListCmp(n), Safe),
            LAMBDA[x](Concat(Fst(x),[Snd(x)]))).

FUNCTION  Safe : (List(Integer), Integer) -> Boolean.
Safe(<p,n>) =>
    FoldR(Map(Zip(From(1),p),
      LAMBDA[k](SOME[i,j,m](k = <i,j> & m = Length(p)+1 &
                         ~(j = n) & ~((i+j) = (m+n)) &
                         ~((i-j) = (m-n))))), &, True).

FUNCTION  ListCmp : Integer -> List((List(Integer),Integer)).
ListCmp(n) =>
  Join(Map(UpTo(1,10),LAMBDA[m](Map(Queens(n-1),LAMBDA[p](p,m))))).

FUNCTION  Concat : List(a) * List(a) -> List(a).
Concat([],y) => y.
Concat([x|xs],y) => [x | Concat(xs,y)].
```

```
FUNCTION  Filter : List(a) * (a -> Boolean) -> List(a).
Filter([], _) => [].
Filter([a|as], f) => IF f(a) THEN [a | Filter(as)] ELSE Filter(as).

FUNCTION  FoldR : List(b) * (a -> b -> a) * a -> a.
FoldR([], _, a) => a.
FoldR([b|bs], f, a) => FoldR(bs, f, f(<a,b>)).

FUNCTION  From : Integer -> List(Integer).
From(n) => [n | From(n+1)].

FUNCTION  Fst : (a,b) -> a.
Fst(<a,b>) => a.

FUNCTION  Hd : List(a) -> a.
Hd([a|as]) => a.

FUNCTION  Join : List(List(a)) -> List(a).
Join(x) => FoldR(x, Concat, []).

FUNCTION  Length : List(a) -> Integer.
Length([]) => 0.
Length([p|ps]) => 1 + Length(ps).

FUNCTION  Map : List(a) * (a -> b) -> List(b).
Map([], _) => [].
Map([a|as],f) => [f(a) | Map(as,f)].

FUNCTION  Snd : (a,b) -> b.
Snd(<a,b>) => b.

FUNCTION  Tl : List(a) -> List(a).
Tl([a|as]) => as.

FUNCTION   UpTo : Integer -> List(Integer).
UpTo(m, n) => IF m > n THEN [] ELSE [m | UpTo(m+1, n)].

FUNCTION  Zip : List(a) * List(b) -> List((a,b)).
Zip([], _) => [].
Zip([a|as], b) =>
  IF   b = []
  THEN []
  ELSE [<a,Hd(b)> | Zip(as,Tl(bs))].
```

A.1.3 Connect

The Connect benchmark is originally presented in [Sør96] to compare the computational capability of driving with that of logic programming.

```
FUNCTION  Connect : City * City * List(City) -> Boolean.
Connect(x,y,z) =>
   IF   Flight(x,y)
   THEN z = []
   ELSE SOME [w,ws] (Flight(x,w) & Connect(w,y,ws) & z = [w|ws]).

FUNCTION  Flight : City * City -> Boolean.
Flight(x,y) =>
   (x = Vienna & y = Paris) \/
   (x = Vienna & y = Rome) \/
   (x = Rome & y = Paris) \/
   (x = Paris & y = London) \/
   (x = Paris & y = Copenhagen).
```

Specialisation of this program with respect to the term `Connect(Vienna,y,[w])` should result in a residual program which indicates the airports that can only be reached from Vienna via exactly one stopover city. The residual program generated by the constraint-based partial evaluator in this case is shown below.

```
FUNCTION  Ans : City * City -> Boolean.
Ans(y, w) =>
    IF   (y = Paris) \/ (y = Rome)
    THEN False
    ELSE SOME [ws_5] (
            IF  (y = London) \/ (y = Copenhagen)
            THEN (w = Paris) & (ws_5 = [])
            ELSE False)).
```

One can see that there is an unnecessary assignment in the nested then-branch. The post-processing unfolding has been performed on each branch of the conditional. However, the existential operator cannot be removed by this optimisation.

A.1.4 Neighbours

The Neighbours program was originally presented in [Llo95]. Searching in Escher is achieved by supplying a county or city to either function, in order to reduce the Boolean expression. This benchmark can be fully unfolded by partial evaluation. In the following example, the county names and city names are defined as functions with type `One -> County` and `One -> City`, respectively.

```
FUNCTION   Neighbours : County * County -> Boolean.
```

```
Neighbours(x, y) =>
    (x = Devon & y = Cornwall) \/
    (x = Devon & y = Dorset) \/
    (x = Devon & y = Somerset) \/
    (x = Avon & y = Somerset) \/
    (x = Avon & y = Wiltshire) \/
    (x = Avon & y = Gloucestershire) \/
    (x = Dorset & y = Wiltshire) \/
    (x = Somerset & y = Wiltshire) \/
    (x = Gloucestershire & y = Wiltshire) \/
    (x = Dorset & y = Somerset) \/
    (x = Dorset & y = Hampshire) \/
    (x = Hampshire & y = Wiltshire) \/
    (x = Hampshire & y = Berkshire) \/
    (x = Hampshire & y = Sussex) \/
    (x = Hampshire & y = Surrey) \/
    (x = Sussex & y = Surrey) \/
    (x = Sussex & y = Kent) \/
    (x = London & y = Surrey) \/
    (x = London & y = Kent) \/
    (x = London & y = Essex) \/
    (x = London & y = Hertfordshire) \/
    (x = London & y = Buckinghamshire) \/
    (x = Surrey & y = Buckinghamshire) \/
    (x = Surrey & y = Kent) \/
    (x = Surrey & y = Berkshire) \/
    (x = Oxfordshire & y = Berkshire) \/
    (x = Oxfordshire & y = Wiltshire) \/
    (x = Oxfordshire & y = Gloucestershire) \/
    (x = Oxfordshire & y = Warwickshire) \/
    (x = Oxfordshire & y = Northamptonshire) \/
    (x = Oxfordshire & y = Buckinghamshire) \/
    (x = Berkshire & y = Wiltshire) \/
    (x = Berkshire & y = Buckinghamshire) \/
    (x = Gloucestershire & y = Worcestershire) \/
    (x = Worcestershire & y = Herefordshire) \/
    (x = Worcestershire & y = Warwickshire) \/
    (x = Bedfordshire & y = Buckinghamshire) \/
    (x = Bedfordshire & y = Northamptonshire) \/
    (x = Bedfordshire & y = Cambridgeshire) \/
    (x = Bedfordshire & y = Hertfordshire) \/
    (x = Hertfordshire & y = Essex) \/
    (x = Hertfordshire & y = Cambridgeshire) \/
    (x = Hertfordshire & y = Buckinghamshire) \/
    (x = Buckinghamshire & y = Northamptonshire).

FUNCTION   IsIn : City * County -> Boolean.

IsIn(x, y) =>
    (x = Bristol & y = Avon) \/
    (x = Taunton & y = Somerset) \/
    (x = Salisbury & y = Wiltshire) \/
    (x = Bath & y = Avon) \/
    (x = Bournemouth & y = Dorset) \/
    (x = Gloucester & y = Gloucestershire) \/
    (x = Torquay & y = Devon) \/
    (x = Penzance & y = Cornwall) \/
    (x = Plymouth & y = Devon) \/
    (x = Exeter & y = Devon) \/
    (x = Winchester & y = Hampshire) \/
    (x = Dorchester & y = Dorset) \/
    (x = Cirencester & y = Gloucestershire) \/
    (x = Truro & y = Cornwall) \/
    (x = Cheltenham & y = Gloucestershire) \/
    (x = Shaftesbury & y = Dorset) \/
    (x = Sherbourne & y = Dorset).
```

Specialisation of the above program with respect to the term

```
     Neighbours(Devon,x)& IsIn(y,x)
```

results in the residual program shown below.

```
FUNCTION  Ans : County * City -> Boolean.

Ans(x, y) =>
    ((x = Cornwall) & (y = Penzance)) \/
   (((x = Cornwall) & (y = Truro)) \/
   (((x = Dorset) & (y = Bournemouth)) \/
   (((x = Dorset) & (y = Dorchester)) \/
   (((x = Dorset) & (y = Shaftesbury)) \/
   (((x = Dorset) & (y = Sherbourne)) \/
   ((x = Somerset) & (y = Taunton))))))).
```

A.1.5 Bubble Sort

This benchmark, originally from [JGS93], demonstrates the power of the partial evaluator to transform the naive bubble sort algorithm into a more powerful sorting algorithm. A naive implementation of the bubble sort algorithm implemented in Escher is shown below.

```
FUNCTION  BS : List(Integer) -> List(Integer).
BS(ls,n) =>
  IF    n = 0
  THEN  ls
  ELSE  BS(Swap(ls),n-1).

FUNCTION  Swap : List(Integer) -> List(Integer).
```

```
Swap([])     => [].
Swap([x|xs]) => Swap2(xs,x).

FUNCTION  Swap2 : List(Integer) * Integer -> List(Integer).
Swap2([],x) => [x].
Swap2([h|t],x) =>
  IF    h < x
  THEN  [h | Swap([x|t])]
  ELSE  [x | Swap([h|t])].
```

Given the above program and the term `BS([x1,x2,x3],3)` as input, the partial evaluator returns the following residual program.

```
FUNCTION  Ans : Integer * Integer * Integer -> List(Integer).

Ans(x1,x2,x3) =>
    IF   x2 < x1
    THEN (IF  (x3 < x1)
         THEN (IF   (x3 < x2)
               THEN [x3,x2,x1]
               ELSE [x2,x3,x1])
         ELSE [x2,x1,x3])
    ELSE (IF  (x3 < x2)
         THEN (IF   (x3 < x1)
               THEN [x3,x1,x2]
               ELSE [x1,x3,x2])
         ELSE [x1,x2,x3]).
```

A.1.6 Lambda

This benchmark program performs a simple test of the specialisation of higher-order expressions. It is adapted from an example in [JGS93].

```
FUNCTION  F : Integer -> Integer.
F(x) =>
  IF    x = 0
  THEN  H(LAMBDA[y](y+1)) + 42
  ELSE  H(LAMBDA[y](y+1)) + H(LAMBDA[y](y+x))

FUNCTION  H : (Integer -> Integer) -> Integer.
H(f) => f(17) + f(42).
```

The term `F(x)` simplifies in Escher to the following expression:

```
IF (x = 0) THEN (H(LAMBDA [y_1] ((y_1 + 1))) + 42)
ELSE (H(LAMBDA [y_1] ((y_1 + 1))) + H(LAMBDA [y_1] ((y_1 + x))))
```

However, by partial evaluation, the higher-order expressions of the program can be optimised away:

```
FUNCTION   Ans : Integer -> Integer.
Ans(x) =>
    IF   x = 0
    THEN 103
    ELSE 61 + ((17 + x) + (42 + x)).
```

A.1.7 Lookup Table

Another benchmark example from [JGS93], this lookup table program can be fully unfolded by partial evaluation.

```
FUNCTION   F : String -> Integer.
F(n) =>
  1 + Lookup(n, ["A","B","C"], [1,2,3]).

FUNCTION   Lookup : String * List(String) * List(Integer) -> Integer.
Lookup(n, ns, vs) =>
  IF    x = []
  THEN  v
  ELSE  (IF    n = Hd(ns)
        THEN   v
        ELSE   Lookup(n, x, Tl(vs)))
  WHERE  x = Tl(ns) & v = Hd(vs).

FUNCTION   Hd : List(a) -> a.
Hd([n|ns]) => n.

FUNCTION   Tl : List(a) -> List(a).
Tl([n|ns]) => ns.
```

The following residual program is computed by invoking the constraint-based partial evaluator with respect to the term `F(n)`.

```
Ans(n) =>
    1 + (IF (n = "A") THEN 1 ELSE (IF (n = "B") THEN 2 ELSE 3)).
```

A similar residual program can be obtained by fully unfolding the following Lookup program, written using Boolean expressions instead of conditionals. The Hd and Tl functions are as defined in the original Lookup program above.

```
FUNCTION  F2 : String * Integer -> Boolean.
F2(n,v) =>
   SOME [v1] (Lookup(n, ["A","B","C"], [1,2,3], v1) &
             v = v1 + 1).

FUNCTION  Lookup2 : String * List(String) * List(Integer) *
                    Integer -> Boolean.
Lookup2(n,ns,vs,v) =>
   SOME [w,ws] (ns = [w|ws] & ((n = w) & v = Hd(vs)) \/
                Lookup2(n, ws, Tl(vs), v)).
```

This program has a slightly different meaning than the original program; if the name does not equal one of the names stored in the program, the latter program returns False instead of the value 3 returned by the former Lookup program.

```
Ans(n, v) =>
    ((n = "A") & (v = 2)) \/
    ((n = "B") & (v = 3)) \/
    ((n = "C") & (v = 4))).
```

A.1.8 MapReduce

The following benchmark program is a member of the Dozens of Problems for Partial Deduction [Leu]. Like the Lambda benchmark program, it demonstrates the specialiser's ability to remove unnecessary higher-order functions, which appear often in practical functional programming.

```
FUNCTION   Map : List(a) * (a -> b) -> List(b).
Map([], f)     => [].
Map([x|xs], f) => [f(x) | Map(xs,f)].

FUNCTION   Reduce : List(a) * (a * b -> b) -> b.
Reduce([], f, base)   => base.
Reduce([x|xs], f, bs) =>
  f(x, Reduce(xs, f, bs)).

FUNCTION   ReduceAdd : List(Integer) -> Integer.
ReduceAdd(ls) => Reduce(ls, +, 0).
```

The above program is specialised with respect to the term `Map(Reduce(ls,+,0))` in order to generate the following residual program.

```
FUNCTION   Ans : List(List(Integer)) -> Integer.
Ans(x) => FN_SP1(x).

FUNCTION   FN_SP1 : List(List(Integer)) -> Integer.
FN_SP1([]) => [].
FN_SP1([x_3 | xs_4]) =>
    [FN_SP5(x_3) | FN_SP1(xs_4)].

FUNCTION   FN_SP5 : List(Integer) -> Integer.
FN_SP5([]) => 0.
FN_SP5([b_9 | bs_10]) =>
    b_9 + FN_SP5(bs_10).
```

A.1.9 SortBy

This benchmark program is an example from [Llo95].

```
FUNCTION  SortBy : (a * a -> Boolean) -> (List(a) -> List(a)).
SortBy(p) => LAMBDA [x] Sort(x,p).

FUNCTION  Sort :  List(a) * (a * a -> Boolean) -> List(a).
Sort([],p) => [].
Sort([x|xs],p) =>
  Insert(Sort(xs,p),p,x).

FUNCTION  Insert :  List(a) * (a * a -> Boolean) * a -> List(a).
Insert([],p,x) => [x].
Insert([y|ys],p,x) =>
    IF   p(x,y)
    THEN [x,y | ys]
    ELSE [y | Insert(ys,p,x)].
```

This program is specialised with respect to the term `SortBy(<)` to obtain the residual program containing the following schema statements.

```
Ans =>
    LAMBDA [x_3] (FN_SP3(x_3)).

FN_SP3([]) => [].
FN_SP3([x_7 | xs_8]) =>
    FN_SP7(x_7, FN_SP3(xs_8)).
```

```
FN_SP7([], x_7) => [x_7].
FN_SP7([y_13| ys_14], x_7) =>
    IF x_7 < y_13
    THEN [x_7, y_13 | ys_14]
    ELSE [y_13 | FN_SP7(x_7, ys_14)].
```

Bibliography

[AAF$^+$98] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In *Static Analysis Symposium '98*, LNCS, pages 262–277, 1998.

[AFJV97] M. Alpuente, M. Falschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 151–162. New York: ACM, 1997.

[AFV96a] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. R. Nielson, editor, *Proc. of European Symp. on Programming Languages, ESOP'96*, LNCS 1058, pages 45–61. Springer-Verlag, 1996.

[AFV96b] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. Technical Report TR DSIC-II/33/96, U. P. Valencia, 1996.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[Aug85] L. Augustsson. Compiling Lazy Pattern-Matching. In *Conference on Functional Programming and Computer Architecture*, LNCS 201, 1985.

[AWS91] W.Y. Au, D. Weise, and S. Seligman. Automatic Generation of Compiled Simulations through Program Specialisation. In *28th IEEE Design Automation Conference*, pages 205–210, 1991.

[BD77] R.M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BE86] D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional programming language. In *Proceedings of the European Symposium on Programming*, LNCS 213, pages 119–132, 1986.

[BGM88] P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[Bla] G. Blair. An Introduction to the Concepts of Timing and Delay in Verilog. `URL: www.ee.ed.ac.uk/~gerard/Teach/Verilog/mjta/Gateway/html/delays.html`.

[Ble77] W.W. Bledsoe. Non-Resolution Theorem Proving. *Artificial Intelligence*, 9:1–35, 1977.

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, Denmark, 1990.

[Bry86] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[Bry92] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BW90] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, 1977.

[CD89] C. Consel and O. Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30(2):79–86, 1989.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, LNCS 523, pages 496–519. Springer-Verlag, 1991.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[Chi90] W.-N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, United Kingdom, March 1990.

[Chi91] W.-N. Chin. Generalising deforestation to all first-order functional programs. In *Workshop on Static Analysis of Equational, Functional and Logic Programming Languages*, BIGRE 74, pages 173–181, 1991.

[Chi92] W.-N. Chin. Fully lazy higher-order removal. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 38–47, 1992.

[Chi94] W.-N. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.

[Chv83] V. Chvátal. *Linear Programming*. W.H. Freeman and Co., 1983.

[CK96] W.-N. Chin and S.-C. Khoo. Better Consumers for Program Specializations. *Journal of Functional and Logic Programming*, 4, 1996.

[Deb97] S. Debray. Resource-Bounded Partial Evaluation. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 179–192, New York, June12–13 1997. ACM Press.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69–116, 1987.

[DG89] J. Darlington and Y. Guo. Narrowing and unification in functional programming - an evaluation mechanism for absolute set abstraction. In *Proceedings of the Conference on Rewriting Techniques and Applications*, LNCS 355, pages 92–108, 1989.

[DGT96] O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation*, volume 1110 of *LNCS*, Dagstuhl Castle, Germany, 1996. Springer.

[DHM96] O. Danvy, N. Hentze, and K. Malmkjær. Resource-bounded partial evaluation. *ACM Computing Surveys*, 28(2):329–332, 1996.

[Dus97] D. Dussart. *Topics in Program Specialization and Analysis for Statically-Typed Functional Languages*. PhD thesis, Katholieke Universiteit Leuven, Belgium, May 1997.

[Ers78] A.P. Ershov. On the Essence of Compilation. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[FB95] T. Frühwirth and P. Brisset. High-Level Implementations of Constraint Handling Rules. Technical Report ECRC-95-20, ECRC, June 1995.

[FN88] Y. Futamura and K. Nogi. Generalized Partial Computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, page 133. North Holland, 1988.

[FNT91] Y. Futamura, K. Nogi, and A. Takano. Essence of Generalized Partial Computation. *Theoretical Computer Science*, 90(1):61–79, 1991. Also in D. Bjørner and V. Kotov: Images of Programming, North-Holland, 1991.

[Fri85] L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proceedings of IEEE International Symposium on Logic Programming*, pages 172–184, 1985.

[Frü94] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, pages 90–107. Springer, 1994.

[Fut71] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 25:45–50, 1971.

[FW88] A.B. Ferguson and P. Wadler. When will deforestation stop? In *Glasgow Workshop on Functional Programming*, pages 39–56, 1988.

[Gal91] J.P. Gallagher. *A System for Specialising Logic Programs*, 1991. University of Bristol, Computer Science Technical Report CS-91-32.

[GB91] J.P. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialisation. *New Generation Computing*, 9:305–333, 1991.

[GG98] M.J.C. Gordon and A. Ghosh. Language Independent RTL Semantics. In *IEEE CS Annual Workshop on VLSI: System Level Design*, 1998.

[GJ96] A.J. Glenstrup and N.D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, LNCS. Springer-Verlag, June 25–28 1996.

[GJMS96] R. Glück, J. Jørgensen, B. Martens, and M. H. Sørensen. Controlling conjunctive partial deduction. In H. Kuchen and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics and Programs*, Lecture Notes in Computer Science, pages 137–151. Springer-Verlag, 1996.

[GK93] R. Glück and A. V. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In G. Filè, P.Cousot, M.Falaschi, and A. Rauzy, editors, *Static Analysis. Proceedings*, LNCS 724, pages 112–123. Springer-Verlag, 1993.

[GK95] R. Glück and A. Klimov. Metasystem Transition Schemes in Computer Science and Mathematics. *World Futures*, 45:213–243, 1995.

[GL96] J.P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In Danvy et al. [DGT96], pages 115–136.

[GL98] J.P. Gallagher and L. Lafave. The Role of Trace Abstractions in Program Specialisation Algorithms. *ACM Computing Surveys: Special Issue on Partial Evaluation*, 1998. (To appear.).

[Glü96] R. Glück. On the Mechanics of Metasystem Hierarchies in Program Transformation. In M. Proietti, editor, *Proceedings of the Workshop on Logic Program Synthesis and Transformation*, LNCS 1048, pages 234–251, 1996.

[Gor95] M. Gordon. The Semantic Challenge of Verilog HDL. In *The Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, June 1995.

[Gor97] M. Gordon. Synthesizable Verilog: syntax and semantics. VFE Project, University of Cambridge, Version 0.7, June 1997.

[Gor98] M. Gordon. Event and Cycle Semantics of Hardware Description Languages. University of Cambridge, Version 1.4, January 1998.

[GS94] R. Glück and M. H. Sørensen. Partial Deduction and Driving are Equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, LNCS 844, pages 165–181. Springer-Verlag, 1994.

[GS96] R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In Danvy et al. [DGT96].

[GT89] R. Glück and V.F. Turchin. Experiments with a Self-applicable Supercompiler. Technical report, City University, New York, USA, 1989.

[Gur93] C.A. Gurr. *A Self-Applicable Partial Evaluator for the Logic Programming Language Gödel*. PhD thesis, University of Bristol, United Kingdom, August 1993.

[Ham91] G. Hamilton. Extending deforestation for first order functional programs. In *Glasgow Workshop on Functional Programming*, pages 134–145, 1991.

[Ham93] G. Hamilton. *Compile-Time Optimisation of Store Usage in Lazy Functional Programs*. PhD thesis, University of Stirling, United Kingdom, October 1993.

[Ham96] G. Hamilton. Higher order deforestation. In H. Kuchen and S.D. Swierstra, editors, *Programming Languages: Implementation, Logics and Programs*, LNCS 1140, pages 213–227, 1996.

[Han90] M. Hanus. Compiling Logic Programs with Equality. In *Proceedings of the Second International Workshop on Programming Langauge Implementation and Logic Programming*, LNCS 456, pages 387–401, 1990.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *J. Logic Programming*, 19,20:583–628, 1994.

[Han97] M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, LNCS 523, pages 448–472. Springer-Verlag, 1991.

[Hig52] G. Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the LMS*, 3(2):326–336, 1952.

[HJ89] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

[HJ91] G. Hamilton and S.B. Jones. Transforming programs to eliminate intermediate structures. In *Workshop on Static Analysis of Equational, Functional and Logic Programming Languages*, BIGRE 74, pages 182–188, 1991.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, Cambridge, MA, 1994.

[IEE95] IEEE. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, 1995. IEEE Standard 1364.

[JBE94] G. Janssens, M. Bruynooghe, and V. Englebert. Abstracting numerical values in CLP(H,N). In Manuel Hermenegildo and Jaan Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (PLILP'94)*, pages 400–414. Springer-Verlag, September 1994.

[JGS93] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. Series editor C. A. R. Hoare.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19,20:503–581, 1994.

[Jon96a] N.D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–504, September 1996.

[Jon96b] N.D. Jones. What Not to Do When Writing an Interpreter for Specialisation. In Danvy et al. [DGT96].

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, LNCS 202, 1985.

[JSS88] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 1(3/4):9–50, 1988.

[Ker97] M. Kerber. Proof planning: A practical approach to mechanised reasoning in mathematics. Technical Report CSRP-97-24, University of Birmingham, School of Computer Science, September 1997.

[Kle52] S. Kleene. *Introduction to Metamathematics*. D. van Nostrand, Princeton, New Jersey, 1952.

[Klo92] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, Oxford, 1992.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[Kom81] J. Komorowski. *A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1981. Linköping Studies in Science and Technology Dissertations 69.

[Kot85] L. Kott. Unfold/fold program transformations. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 411–434. Cambridge University Press, 1985.

[Kru60] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the AMS*, 95:210–225, 1960.

[LD94] J.L. Lawall and O. Danvy. Continuation-Based Partial Evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238. ACM, June 1994.

[Leu] M. Leuschel. Dozens of problems for partial deduction. A compilation of logic programs from various sources. Available from `http://www.cs.kuleuven.ac.be/~dtai/prototypes/dppd.html`.

[Leu95] M. Leuschel. Ecological Partial Deduction: Preserving Characteristic Trees without Constraints. In M. Proietti, editor, Logic Program Synthesis and Transformation. *Proceedings of LOPSTR'95*, LNCS 1048, pages 1–16, 1995.

[Leu97a] M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, Katholieke Universiteit Leuven, Belgium, May 1997.

[Leu97b] M. Leuschel. Extending Homeomorphic Embedding in the Context of Logic Programming. Technical Report CW 252, Department of Computer Science, Katholieke Universiteit Leuven, June 1997.

[LG97] L. Lafave and J.P. Gallagher. Information Propagation in Partial Evaluation by Constraints. In *Workshop on Constraint Programming for Reasoning about Programming*, April 1997.

[LG98a] L. Lafave and J.P. Gallagher. Constraint-based Partial Evaluation of Rewriting-based Functional Logic Programs. In *Seventh Annual Logic Program Synthesis and Transformation (LOPSTR '97)*, LNCS 1463, 1998.

[LG98b] L. Lafave and J.P. Gallagher. Extending the Power of Automatic Partial Evaluators. *ACM Computing Surveys: Special Issue on Partial Evaluation*, 1998. (To appear.).

[Llo] J.W. Lloyd. Programming in an Integrated Functional and Logic Language. Due for publication.

[Llo93] J.W. Lloyd. *Foundations of Logic Programming*. Artificial Intelligence. Springer-Verlag, 1993.

[Llo95] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[LM96] M. Leuschel and B. Martens. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. In Danvy et al. [DGT96], pages 263–283.

[Lom64] L.A. Lombardi. Incremental Computation. In F.L. Alt and M. Rubinoff, editors, *Advances in Computers*, pages 247–333. Academic Press, 1964.

[LR67] L.A. Lombardi and B. Raphael. LISP as the Language for an Incremental Computer. In E.C. Berkeley and D.G. Bobrow, editors, *The Programming Language LISP: Its Operation and Applications*, pages 204–219. MIT Press, 1967.

[LS91] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11(3&4):217–242, October 1991.

[LS96] M. Leuschel and M.H. Sørensen. Redundant argument filtering of logic programs. In J. Gallagher, editor, *International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 63–77, Stockholm, Sweden, August 1996.

[LS97] M. Leuschel and D. De Schreye. Constrained Partial Deduction and the Preservation of Characteristic Trees. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, June 1997. Accepted for publication in *New Generation Computing*.

[LSdW96] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction; Towards a Maximal Integration. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.

[Mar94] B. Martens. *On the Semantics of Meta-Programming and the Control of Partial Deduction in Logic Programming*. PhD thesis, K.U. Leuven, Belgium, 1994.

[Mar96] S.D. Marlow. *Deforestation for Higher-Order Functional Languages*. PhD thesis, University of Glasgow, United Kingdom, 1996.

[McL87] M.R. McLauchlan. *Computer Aided Tools for VLSI System Design*, chapter High level languages in design. IEE, 1987.

[MG95] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Stirling, editor, *International Conference on Logic Programming*, pages 597–613. MIT Press, 1995.

[MG97] M. Marinescu and B. Goldberg. Partial Evaluation Techniques for Concurrent Programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 47–62, New York, 1997. ACM Press.

[Mic86] A. Miczo. *Digital Logic Simulation and Testing.* Harper & Row, 1986.

[MNRA92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates. *Journal of Logic Programming*, 12:191–223, 1992.

[MW92] S.D. Marlow and P.L. Wadler. Deforestation for higher-order functions. In J. Launchbury, editor, *Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 154–165, 1992.

[NN88] H. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 98–106, January 1988.

[NPT96] A.P. Nemytykh, V.A. Pinchuk, and V.F. Turchin. A Self-Applicable Supercompiler. In Danvy et al. [DGT96].

[NW63] C.St.J.A. Nash-Williams. On well-quasi-ordering finite trees. *Proc. Cambridge Philosophical Society*, 59(4):833–835, 1963.

[Pal96] S. Palnitkar. *Verilog HDL : A Guide to Digital Design.* Prentice Hall, 1996.

[Pla93] D.A. Plaisted. Equational reasoning and term rewriting systems. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1 – Logical Foundations of *Oxford Science Publications*, pages 273–364. Clarendon Press, 1993.

[PP90] M. Proietti and A. Pettorossi. Synthesis of Eureka Predicates for Developing Logic Programs. In *Proceedings of the European Symposium on Programming*, LNCS 432, pages 305–325, 1990.

[PP94] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19-20:261–320, 1994.

[PP98] A. Pettorossi and M. Proietti. Program Specialization via Algorithmic Unfold/Fold Transformations. *ACM Computing Surveys: Special Issue on Partial Evaluation*, 1998. (To appear.).

[PPR97] A. Pettorossi, M. Proietti, and S. Renault. Enhancing Partial Deduction via Unfold/Fold Rules. In J. Gallagher, editor, *Logic Program Synthesis and Transformation (LOPSTR'96)*, LNCS 1207, pages 147–168. Springer-Verlag, 1997.

[Red85] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 138–151, 1985.

[Rob65] J.A. Robinson. A Machine-Oriented Logic based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rog67] H. Rogers. *Theory of recursive functions and effective computability*. McGraw-Hill, 1967.

[RS82] J.A. Robinson and E.E. Sibert. LOGLISP: Motivation, Design and Implementation. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 299–313. Academic Press, 1982.

[Ruf93] E. Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, USA, March 1993. Also released as technical report CSL-TR-93-563.

[Sağ98] H. Sağlam. *A Toolkit for Static Analysis of Constraint Logic Programs*. PhD thesis, University of Bristol, United Kingdom, March 1998.

[San] J. Sanguinetti. Simulation Speed in Hardware Description Languages.
URL: www.comit.com/˜rajesh/verilog/info/sim_speed.html.

[San95a] G. Sander. *VCG: Visualization of Compiler Graphs*. Universität des Saarlandes, Germany, February 1995.

[San95b] D. Sands. Proving the Correctness of Recursion-Based Automatic Program Transformations. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Sixth International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, LNCS 915, pages 681–695. Springer-Verlag, 1995.

[San95c] D. Sands. Total Correctness by Local Improvement in Program Transformation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 221–232. ACM Press, 1995.

[Sei96] H. Seidl. Integer constraints to stop deforestation. In *European Symposium on Programming*, LNCS 1058, pages 326–340, 1996.

[Ses88a] P. Sestoft. Automatic Call Unfolding in a Partial Evaluator. In D. Bjørner, A. P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, 1988.

[Ses88b] P. Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, Denmark, 1988.

[SG95] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In J. W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.

[SGJ94] M.H. Sørensen, R. Glück, and N. D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In *ESOP*. Springer-Verlag, 1994.

[SGJ96] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[Smi91] D. A. Smith. Partial Evaluation of Pattern Matching in Constraint Logic Programming Languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 62–71. ACM Press, 1991.

[Smi97] D.J. Smith. *HDL Chip Design, A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog*. Doone Publications, 1997.

[Sør93] M.H. Sørensen. A new means of ensuring termination of deforestation with an application to logic programming. In *Global Compilation Workshop*, 1993. Proceedings appeared as Penn State Technical Report.

[Sør94] M.H. Sørensen. A Grammar-based Data-flow Analysis to Stop Deforestation. In *CAAP*. Springer-Verlag, 1994.

[Sør96] M.H. Sørensen. Turchin's Supercompiler Revisited. Master's thesis, DIKU, Denmark, January 1996.

[SS88] K. Sakai and Y. Sato. Boolean Gröbner Bases. Technical report, ICOT Research Center Japan, June 1988.

[SS97] H. Seidl and M.H. Sørensen. Constraints to Stop Higher-Order Deforestation. In N.D. Jones, editor, *ACM Symposium on Principles of Programming Languages*, pages 400–413, 1997.

[Stu91] P.J. Stuckey. Constructive Negation for Constraint Logic Programming. In *Proceedings of the Logic in Computer Science Conference*, pages 328–339, 1991.

[Syn96] Synopsys, Inc. *HDL Compiler for Verilog Reference Manual*, 1996. Version 3.5.

[Tak91] A. Takano. Generalized partial computation for a lazy functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 1–11. New York: ACM, 1991.

[Tak92] A. Takano. Generalized partial computation using disunification to solve constraints. *Lecture Notes in Computer Science*, 656:424–428, 1992.

[Thi94] P. Thiemann. Higher-order redundancy elimination. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 73–84, 1994.

[TNT82] V. F. Turchin, R.M. Nirenberg, and D. V. Turchin. Experiments with a supercompiler. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 47–55. ACM, August 1982.

[Tur86] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Tur88] V.F. Turchin. The algorithm of generalization. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549, 1988.

[Wad88] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, LNCS 300, pages 344–358. Springer Verlag, 1988.

[Wad90] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wel97] M. Welinder. *Partial Evaluation and Correctness*. PhD thesis, DIKU, Denmark, 1997.

[Wos96] L. Wos. *The Automation of Reasoning: An Experimenter's Notebook with OTTER Tutorial*. Academic Press, 1996.