Debugging Techniques for Declarative Languages:
Profiling, Program Slicing, and Algorithmic Debugging

Josep Francesc Silva Galiana
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Memoria presentada para optar al título de:

Doctor en Informática

Director:

Germán Vidal Oriola

Valencia, 2007

A Julio y Nina, que me dieron a Vanessa
A mi padre, mi guía

Prologue

I enjoy reading books about neutron stars and black holes, and one of my best friends use to say that I am losing my time because I will never see a black hole, and it is stupid to study something that will never influence my live... One hundred years ago people used to read science fiction about the space and, nowadays, hundreds of satellites are rounding the Earth: what is a dream today will be a fact tomorrow. This is the way in which researchers must think, since much of their work is useless today but, possibly, another researcher will make it useful tomorrow. In fact, this is an annoying property of research: sometimes it is blind. When Fleming discovered penicillin after the first world war, he did not know that he was going to save millions of lives, and the scientific community did not realize about the importance of his discovery until the second world war. Most of the results of this thesis has been implemented and applied to a real language, thus producing useful technology; but others simply no. Some results are theoretical results that I have not been able to apply and I simply hope they will become useful in the future.

Some of the most important discoveries have been discovered accidentally or casually. When Archimedes discovered the Archimedes' principle he was having a bath; and Roentgen switched off the light and saw a glow from a drawer before he discovered X rays. Others, on the other hand, have been discovered after many years of hard work. Einstein spent five years to generalize the relativity theory; and Mendel had to plant and experiment for decades with thousands of pea before he arrived to a definite conclusion and stated the principles of genetics.

Unfortunately (or maybe fortunately), this thesis belongs to the second group. I started to study the principles of debugging techniques for declarative programming in March of 2002. I got a grant in the Technical University of Valencia to participate in the research of the ELP research group, and I went for the first time in the office of my advisor German Vidal. He just asked me: "*What do you like more, writing programs or correcting them?*" In that moment I did not realize that my answer was going to mark my future for at least the next four years...

Debugging Techniques of Declarative Languages

Abstract

The task of debugging can be arduous. A bug can be evident with a single glance, or it can be hidden in the deepest lurking place of our program. Nevertheless, surprisingly, debugging is one of the software processes that has been mostly abandoned by the scientific community, and the same debugging techniques used twenty years ago are still being used today.

The situation is not different regarding declarative languages. Or it is indeed worst; because declarative languages can pose additional difficulties due, for instance, to the lazy evaluation mechanism.

In this thesis, we revise the current debugging methods for declarative languages and we develop some new methods and techniques which are based on profiling, program slicing, and algorithmic debugging. In short, the main contributions of the thesis are:

- The definition of a profiling scheme for functional logic programs which is based on the use of *cost centers* and that allows us to measure different kinds of symbolic costs.
- The formulation of a new dynamic slicing technique based on *redex trails*, its application to debugging and its adaptation for the specialization of modern multi-paradigm declarative programs.
- The introduction of a new algorithmic debugging scheme which combines conventional algorithmic debugging with program slicing.
- The definition of three new strategies for algorithmic debugging.
- The development of a comparative study and a subsequent classification of program slicing techniques and algorithmic debugging strategies.

Part of the material included in this thesis has been previously presented in [Albert *et al.*, 2003; Braßel *et al.*, 2004a; Ochoa *et al.*, 2004a, 2005, 2006; Silva and Chitil, 2006; Silva, 2006].

Resumen

La depuración de programas puede ser una tarea árdua. Esto se debe fundamentalmente a que los errores pueden ser evidentes al primer vistazo, o pueden estar escondidos en las zonas más profundas y ocultas de nuestros programas. Sin embargo, sorprendentemente, la depuración es uno de los procesos software que ha sido menos tratado por la comunidad científica; de hecho, las mismas técnicas de depuración que se usaban hace veinte años, continúan utilizándose en la actualidad.

La situación no es diferente en el contexto de los lenguajes declarativos. O es incluso peor, porque estos lenguajes suelen presentar dificultades adicionales a la hora de depurarlos debido, por ejemplo, al mecanismo de evaluación perezosa.

En esta tesis se revisan los métodos actuales de depuración para lenguajes declarativos y se desarrollan nuevos métodos y técnicas basadas en el cómputo de costes, la fragmentación de programas y la depuración algorítmica. Brevemente, las principales contribuciones de la tesis son:

- La definición de un esquema de cómputo de costes para programas lógico funcionales que está basado en el uso de *centros de coste* y que permite medir diferentes tipos de costes simbólicos.
- La formulación de una nueva técnica de fragmentación dinámica de programas basada en *redex trails*, su aplicación a la depuración y su adaptación para la especialización de programas declarativos multi-paradigma modernos.
- La introducción de un nuevo esquema de depuración algorítmica que combina la depuración algorítmica convencional con la fragmentación de programas.
- La definición de tres nuevas estrategias para depuración algorítmica.
- El desarrollo de un estudio comparativo y una posterior clasificación de técnicas de fragmentación de programas y estrategias de depuración algorítmica.

Parte del material de esta tesis ha sido presentado en [Albert *et al.*, 2003; Braßel *et al.*, 2004a; Ochoa *et al.*, 2004a, 2005, 2006; Silva and Chitil, 2006; Silva, 2006].

Resum

La depuració de programes pot ser una tasca àrdua. Açò es deu fundamentalment a què els errors poden esdevindre evidents amb un sol colp d'ull o bé, poden estar amagats en els fragments més profunds i ocults del nostre codi. En canvi, sorprenentment, la depuració és un dels procesos del programari que han estat menys tractats per la comunitat científica; de fet, les mateixes tècniques de depuració que s'empraven fa vint anys, encara continuen utilitzant-se en la actualitat.

La situació no és diferent en el context dels llenguatges declaratius. O tal volta pitjor, perque aquestos llenguatges solen prenentar dificultats addicionals a l'hora de depurar-los per culpa, per exemple, del mecanisme d'avalluació mandrós.

En aquesta tesi es revisen els mètodes actuals de depuració per a llenguatges declaratius i es desenvolupen nous mètodes i noves tècniques basades amb el còmput de costs, la fragmentació de programes i la depuració algorismica. Breument, les principals contribucions de la tesis són:

- La definició d'un esquema de còmput per a programes lògic-funcionals que està basat amb l'ús de centres de cost i permet mesurar els diferents tipus de costs simbòlics.
- La formulació d'una nova tècnica de fragmentació dinàmica de programes basada en "redex-trails", la seua aplicació a la depuració declarativa i la seua adaptació per a l'especialització de programes declaratius multi-paradigma moderns.
- La introducció d'un nou esquema de depuració algorísmica que combina la depuració d'algorismes convencionals amb la fragmentació de programes.
- La definició de tres noves estratègies per a depuració algorísmica.
- El desenvolupament d'un estudi comparatiu i una posterior classificació de tècniques de fragmentació de programes i estratègies de depuració algorísmica.

Part del material d'aquesta tesi ha estat presentat a [Albert *et al.*, 2003; Braßel *et al.*, 2004a; Ochoa *et al.*, 2004a, 2005, 2006; Silva and Chitil, 2006; Silva, 2006].

Contents

1 Introduction **1**
1.1 Debugging Declarative Languages 1
1.2 Debugging Techniques 4
1.2.1 Traditional Debugging 4
1.2.2 Profiling 5
1.2.3 Algorithmic Debugging 6
1.2.4 Abstract Debugging 17
1.2.5 Tracing 18
1.2.6 Program Slicing 19
1.3 Structure of the Thesis 34

I Foundations **37**

2 Preliminaries **39**
2.1 Signature and Terms 39
2.2 Substitutions 39
2.3 Term Rewriting Systems 40
2.3.1 Narrowing 41
2.3.2 Needed Narrowing 42
2.4 Declarative Multi-Paradigm Languages 45

II Profiling **55**

3 Time Equations for Lazy Functional (Logic) Languages **57**
3.1 Introduction 57
3.2 Cost Augmented Semantics 59
3.3 The Program Transformation 61
3.4 The Transformation in Practice 67

3.4.1 Complexity Analysis . . . 67
3.4.2 Other Applications and Extensions . . . 69
3.5 Related Work . . . 71

4 Runtime Profiling of Functional Logic Programs 75
4.1 Introduction . . . 75
4.2 A Runtime Profiling Scheme . . . 77
4.3 Cost Semantics . . . 79
4.4 Cost Instrumentation . . . 81
4.5 Implementation . . . 89
4.5.1 Measuring Runtimes . . . 89
4.5.2 Extension for Symbolic Profiling . . . 90
4.6 Related Work and Conclusions . . . 91

III Program Slicing 93

5 Dynamic Slicing Based on Redex Trails 95
5.1 Introduction . . . 96
5.2 Combining Tracing and Slicing . . . 98
5.3 Building the Extended Trail . . . 103
5.3.1 A Semantics for Tracing . . . 103
5.3.2 Including Program Positions . . . 105
5.3.3 Logical Features . . . 114
5.4 Computing the Slice . . . 115
5.4.1 Logical Features . . . 130
5.5 Program Specialization Based on Dynamic Slicing . . . 131
5.5.1 Specialization Based on Slicing . . . 131
5.6 Implementation Issues . . . 135
5.6.1 Architecture . . . 138
5.6.2 The Slicer in Practice . . . 139
5.6.3 Benchmarking the Slicer . . . 143
5.7 Related Work . . . 143

IV Algorithmic Debugging 147

6 Combining Program Slicing and Algorithmic Debugging 149
6.1 Introduction . . . 149
6.2 The Idea: Improved Algorithmic Debugging . . . 150

6.3 Computation Graphs . . . 151
6.4 The Augmented Redex Trail . . . 152
6.5 Slicing the ART . . . 157
6.6 Combining Program Slicing and Algorithmic Debugging . . . 160
6.6.1 The Slicing Criterion . . . 160
6.6.2 The Slicing Algorithm . . . 163
6.6.3 Completeness of the Slicing Algorithm . . . 164
6.7 The Technique in Practice . . . 166
6.8 Related Work . . . 168

7 A Classification of Program Slicing and Algorithmic Debugging Techniques 171
7.1 Introduction . . . 171
7.2 Classifying Slicing Techniques . . . 172
7.3 Three New Algorithmic Debugging Strategies . . . 175
7.3.1 Less YES First . . . 175
7.3.2 Divide by YES & Query . . . 179
7.3.3 Dynamic Weighting Search . . . 180
7.4 Classifying Algorithmic Debugging Strategies . . . 183
7.4.1 Combining Algorithmic Debugging Strategies . . . 183
7.4.2 Comparing Algorithmic Debugging Strategies . . . 185

8 Conclusions and Future Work 191
8.1 Conclusions . . . 191
8.2 Future Work . . . 194

9 Acknowledgements 199

10 Agradecimientos 203

Chapter 1

Introduction

In this chapter, we offer a general overview of the state of the art in our area of interest: debugging of declarative languages. First, we briefly recall the main approaches and results found in the literature. Then, we concisely present the main goals of this thesis.

1.1 Debugging Declarative Languages

In 1998, Philip Wadler published in the *Journal of Functional Programming* a very controversial article entitled "*why no one uses functional languages*" [Wadler, 1998]. He identified the difficulty of debugging such languages as one of the main causes. In particular—he said—the main difficulty of debugging lazy languages is the lack of proper debugging environments:

> "*Constructing debuggers and profilers for lazy languages is recognized as difficult. Fortunately, there have been great strides in profiler research, and most implementations of Haskell are now accompanied by usable time and space profiling tools. But the slow rate of progress on debuggers for lazy languages makes us researchers look, well, lazy.*"

A good tracer, program slicer or algorithmic debugger able to help the programmer to understand what could be wrong during the execution of a program can considerably simplify the task of debugging; and, consequently, the time expended in this task.

Many efforts have been done to develop different environments for debugging declarative programs (see, e.g., [Wallace *et al.*, 2001; Caballero, 2005; MacLarty, 2005]) but the problem still remains. Unfortunately, none of these works have completely solved the problem. However, all these works are necessary to find a solution.

The techniques presented in this thesis constitute a step forward to solve some of the problems related to the current state of the art in debugging.

Languages with a *lazy* evaluation strategy (in the following, lazy languages) impose additional difficulties over languages with an *eager* evaluation strategy (in the following, eager languages); in general, these languages lack of suitable debugging tools or, if they exist, they are mostly useless for programmers and, in practice, they are hardly used due to their research-oriented nature and lack of stability [Wadler, 1998]. In the particular case of multi-paradigm languages, this lack is even worst because these languages are relatively young, and, thus, the maturity level of the debugging tools is sensibly lower.

Declarative multi-paradigm languages combine some of the main features from the pure functional and logic programming paradigms, as well as from other paradigms (e.g., concurrency). Integrated multi-paradigm languages allows us to see different forms of computation as different aspects of a unique computation mechanism. This versatility eases the task of programming allowing us to use in each aspect of a problem the most natural solution, which could be based on functional programming, logic programming, constraint programming, object-oriented programming, etc.

Among the most important multi-paradigm languages one can find Curry [Hanus, 1997, 2006a], Toy [López-Fraguas and Sánchez-Hernández, 1999], Mercury [MacLarty, 2005], and Oz [Muller *et al.*, 1995]. The techniques proposed in this work could be adapted to any functional logic language, as well as to pure logic and pure functional languages; however, to be concrete, the developments and implementations described have been based on the functional logic language Curry.

Curry is a language that combines elements from functional programming and logic programming:

Functional Programming (e.g., Haskell [Peyton-Jones, 2003]) is based on the idea that a computation consists on the *evaluation* (with referential transparency) of expressions. The main contributions from this paradigm are higher-order functions, function composition and the use of types.

Logic Programming (e.g., Prolog [Sterling and Shapiro, 1986]) defines computations as a series of *deductions*. The main contributions from this paradigm are the logical variables and the search mechanisms.

In Chapter 2 a revision of this language is presented.

This thesis reviews some of the most important debugging methods for multi-paradigm declarative languages; then it introduces new techniques to solve some of the limitations of such languages. In particular, this thesis focuses on three well-known techniques:

Program Slicing: It is a technique to extract slices from a given program. A *program slice* is formed by all those program's expressions that (potentially) affect the values computed at some point of interest for the programmer. This point is known as the *slicing criterion*. The task of computing program slices is known as program slicing. The original definition of program slice—introduced by Mark Weiser in 1979 [Weiser, 1979, 1984]—defined it as an executable program extracted from another program by deleting sentences in such a way that the slice reproduces part of the behavior of the original program. See Section 1.2.6 for a deeper explanation.

Profiling: Basically, it consists on the computation of the resources (time, memory, IO operations, etc.) that programs use during their execution. Despite the fact that this technique has classically used milliseconds to measure time and bytes to measure memory, in Chapter 4 we will see a different approach based on *symbolic* costs which is able to measure the number of elemental operations that a program performs during its execution. See Section 1.2.2 for a deeper explanation.

Algorithmic Debugging: This is a semi-automatic debugging technique which is based on the answers of the programmer to a series of questions generated automatically by the algorithmic debugger. After the programmer has answered some questions, the debugger is able to automatically find the bug in the source code. See Section 1.2.3 for a deeper explanation.

These techniques have been applied in many other contexts besides program debugging. In particular, program slicing has been applied, e.g., to program debugging [Lyle, 1984; Lyle and Weiser, 1987; Agrawal, 1991; Shimomura, 1992; Kamkar, 1993], testing [Laski, 1990; Bates and Horwitz, 1993; Kamkar *et al.*, 1993; Rothermel and Harrold, 1994; Binkley, 1995, 1998], parallelization [Choi *et al.*, 1991; Weiser, 1983], detection and removal of *dead code* [Jones and Métayer, 1989; Reps and Turnidge, 1996; Liu and Stoller, 2003], integration [Reps, 1991; Reps *et al.*, 1994; Binkley *et al.*, 1995], software safety [Gallagher and Lyle, 1993], understanding [Cutillo *et al.*, 1993; Lanubile and Visaggio, 1993; Merlo *et al.*, 1993] and software maintenance [Gallagher and Lyle, 1991; Platoff *et al.*, 1991; Ramalingam and Reps, 1991, 1992].

The techniques proposed in this thesis could also be used in the above mentioned contexts. Firstly, the main objective was the development of a profiler for the PAKCS programming environment [Hanus *et al.*, 2006] of Curry. The techniques proposed together with its implementation are described in Part II. Secondly, the purpose was to complement the current Curry tracer with new debugging capabilities based on slicing (the language Curry lacked of a program slicer). Part III describes how we have extended the standard semantics of Curry to be able to slice programs, and how

we have implemented it in a single tool which integrates the Curry's tracer, slicer and program specializer. Thirdly, we focused on algorithmic debugging. In particular, Part IV introduces some new algorithmic debugging strategies and compares them with previous strategies in order to establish a classification of algorithmic debugging strategies.

1.2 Debugging Techniques

In this section we review the state of the art in debugging techniques for declarative languages. Most of the techniques discussed here have also been adapted to—or come from—the imperative paradigm. We present here the most relevant techniques that, in some sense, are related with the techniques later developed in the thesis. In particular, those techniques that will be extended in the forthcoming chapters will be explained in detail in their respective chapters. We omit here details about the applications of these techniques to particular languages or implementations.

1.2.1 Traditional Debugging

One of the most common debugging practices still seem to be the use of print statements. This technique, also called "*echo debugging*" or "*print debugging*", basically consists in adding print statements in "key" parts of the program. With these print statements, the programmer can check if some condition is satisfied, or he can also observe the value of some variables of interest in some parts of the execution.

This kind of debugging can be done in every programming language without the help of a debugging tool: it comes for free with the language. However, its main drawback is that erroneous values are often observed long after the bug, and the programmer has to iterate over the program locating the different print statements over the source code which can be an arduous task. Moreover, in every iteration, the programmer has to add the print statement, recompile the program and run it again in order to see the value printed by the new print statement.

$$\textit{modify source code} \rightarrow \textit{compile} \rightarrow \textit{execute}$$

Some debugging tools (see, e.g., [Somogyi and Henderson, 1999]), give the possibility to inspect the state of the compiler at some point of the execution. The programmer can insert *breakpoints* in the program—breakpoints allow us to see at different times of a computation part of the computation state— and, during the execution, the compiler shows the values of some specified variables at these points. In some tools, it is also possible to specify some breakpoint conditions that the compiler evaluates. If the conditions are satisfied, then the compiler stops at this breakpoint; otherwise, the compiler ignores this breakpoint and the execution continues.

Some compilers incorporate mechanisms to statically identify potentially sources of bugs. For instance, by identifying if a parameter is passed through a set of functions without being used; or if an argument of a function never influences the result of the function, etc. This kind of static analysis is known as *Anomaly Detection* [Osterweil and Fosdick, 1976].

1.2.2 Profiling

The York functional programming group describes the necessity of profiling tools as follows [York Functional Programming Group, 2006]:

> "*Functional programs often display elegance and simplicity; yet their computational behavior may be complex, depending on the subtle interplay of several mechanisms in the process of evaluation. Solutions that are concise as defining formulae can give rise in their execution to surprisingly large demands for machine resources.*"

Profiling is a method to identify sections of code that consume large portions of execution time. In a typical program, most execution time is spent in relatively few sections of code. To improve performance, the greatest gains result from improving coding efficiency in time-intensive sections. In particular, profilers can analyze code and detect implementation inefficiencies, which are yet another form of a bug.

Profiling methods include:

Program counter (PC) sampling: This technique periodically interrupts the program being executed and logs the value of the PC. After execution, the logs can be analyzed in order to determine which parts of the code consume the most time.

Basic block counting: These techniques modify the program by inserting profiling code at strategic locations. The augmented/instrumented code can calculate the number of times it is executed.

Compiler profilers: This technique basically consist in the extension of a compiler with profiling capabilities. While executing a program, the compiler logs the number of resources (memory, time, disk space, etc.) needed by each portion of code.

There are different kinds of profilers depending on the purpose of the programmer. Firstly, the information provided by the profiler is rather different (for example, CPU usage, call counts, call cost, memory usage, and I/O operations); and, moreover, this information can be collected at different granularity levels (for example, at a

module level or at a function level). In addition, it is important to specify whether the profiler must consider the shared libraries used by the application or not. And, secondly, the method used to profile the programs determines whether the program being profiled must be modified or not. For instance, compiler profilers can profile a program without modifying it, while basic block counting profilers need to instrument programs by some kind of program transformation.

One of the most important profilers for functional languages is the Haskell profiler [Sansom and Peyton-Jones, 1997] which was the first to introduce the notion of *cost centers* (a key piece in our work that we will deeply explain in Chapter 4). Roughly, a cost center is a container that the programmer can set in the program by means of a special operator, and which is used to accumulate the cost of executing one part of the program (e.g., an expression). Another important characteristic of the Haskell profiler is the possibility to measure symbolic costs. This means that this profiler not only computes space and time costs, but it is also able to compute the number of elemental operations performed during a computation. For instance, a symbolic profiler can compute the number of function unfoldings performed during a computation. The reader is referred to Chapter 4 for details about symbolic profiling.

Apart from the use of an augmented compiler to measure costs—this is the case, e.g., of [Sansom and Peyton-Jones, 1997] and [Albert and Vidal, 2002]—another frequent approach to profiling is the definition of a program transformation. Approaches based on a program transformation are, for instance, [Métayer, 1988; Sands, 1990; Rosendahl, 1989]. We will discuss and compare them to our approach in Chapter 3.

1.2.3 Algorithmic Debugging

Algorithmic debugging is a debugging technique which relies on the programmer having an *intended interpretation* of the program. In other words, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Therefore, algorithmic debuggers compare the results of subcomputations with what the programmer intended. By asking the programmer questions or using a formal specification the system can identify precisely the location of a program's bug.

Essentially, algorithmic debugging is a two-phase process: An *execution tree* (see, e.g., [Naish, 1997a]), ET for short, is built during the first phase. Each node in this ET corresponds to an equation which consists of a function call with completely evaluated arguments and results[1]. Roughly speaking, the ET is constructed as follows: The root node is the *main* function of the program; for each node n with associated function f, and for each function call in the right-hand side of the definition of f which has been evaluated, a new node is recursively added to the ET as the child of n. This

[1] Or as much as needed if we consider a lazy language.

notion of ET is valid for functional languages but it is insufficient for other paradigms as the imperative programming paradigm. In general, the information included in the nodes of the ET includes all the data needed to answer the questions. For instance, in the imperative programming paradigm, with the function (or procedure) of each node it is included the value of all global variables when the function was called. Similarly, in object-oriented languages, every node with a method invocation includes the values of the attributes of the object owner of this method (see, e.g., [Caballero, 2006]). In the second phase, the debugger traverses the ET asking an oracle (typically the programmer) whether each equation is correct or wrong. At the beginning, the *suspicious area* which contains those nodes that can be buggy (a buggy node is associated with a buggy rule of the program) is empty; but, after every question, some nodes of the ET leave the suspicious area. When all the children of a node with a wrong equation (if any) are correct, the node becomes buggy and the debugger locates the bug in the function definition of this node [Nilsson and Fritzson, 1994]. If a bug symptom is detected then algorithmic debugging is complete [Shapiro, 1982]. It is important to say that, once the execution tree is built, the problem of traversing it and selecting a node is mostly independent of the language used (a clear exception are those strategies which use information about the syntax of the program); hence many algorithmic debugging strategies can theoretically work for any language.

Unfortunately, in practice—for real programs—algorithmic debugging can produce long series of questions which are semantically unconnected (i.e., consecutive questions which refer to different and independent parts of the computation) making the process of debugging too complex.

Furthermore, questions can also be very complex. For instance, during a debugging session with a compiler, the algorithmic debugger of the Mercury language [MacLarty, 2005] asked a question of more than 1400 lines.

Therefore, new techniques and strategies to reduce the number of questions, to simplify them and to improve the order in which they are asked are a necessity to make algorithmic debuggers usable in practice.

During the algorithmic debugging process, an oracle is prompted with equations and asked about their correctness; it answers "YES" when the result is correct or "NO" when the result is wrong. Some algorithmic debuggers also accept the answer "I don't know" when the programmer cannot give an answer (e.g., because the question is too complex). After every question, some nodes of the ET leave the suspicious area. When there is only one node in the suspicious area, the process finishes reporting this node as buggy. It should be clear that algorithmic debugging finds one bug at a time. In order to find different bugs, the process should be restarted again for each different bug.

```
main = sqrtest [1,2]

sqrtest x = test (computs (listsum x))

test (x,y,z) = (x==y) && (y==z)

listsum [] = 0
listsum (x:xs) = x + (listsum xs)

computs x = ((comput1 x),(comput2 x),(comput3 x))

comput1 x = square x

square x = x*x

comput2 x = listsum (list x x)

list x y | y==0      = []
         | otherwise = x:list x (y-1)

comput3 x = listsum (partialsums x)

partialsums x = [(sum1 x),(sum2 x)]

sum1 x = div (x * (incr x)) 2
sum2 x = div (x + (decr x)) 2

incr x = x + 1
decr x = x - 1
```

Figure 1.1: Example program

Let us illustrate the process with an example[2].

Example 1 *Consider the buggy program in Figure 1.1 adapted to Haskell from [Fritzson et al., 1992]. This program sums a list of integers [1,2] and computes the square of the result with three different methods. If the three methods compute the same result the program returns True; otherwise, it returns False. Here, one of the three methods—the one adding the partial sums of its input number—contains a bug. From*

[2]While almost all the strategies presented here are independent of the programming paradigm used, in order to be concrete and w.l.o.g. we will base our examples on the functional programming paradigm.

```
Starting Debugging Session...
(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(3)  test [9,9,8] = False? YES
(4)  computs 3 = [9,9,8]? NO
(5)  comput1 3 = 9? YES
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

Figure 1.2: Debugging session for the program in Figure 1.1

this program, an algorithmic debugger can automatically generate the ET of Figure 1.3 (for the time being, the reader can ignore the distinction between different shapes and white and dark nodes) which, in turn, can be used to produce a debugging session as depicted in Figure 1.2. During the debugging session, the system asks the oracle about the correctness of some ET nodes w.r.t. the intended semantics. At the end of the debugging session, the algorithmic debugger determines that the bug of the program is located in function "`sum2`*" (node 23). The definition of function "*`sum2`*" should be:* `sum2 x = div (x*(decr x)) 2`

Algorithmic debugging strategies are based on the fact that the ET can be pruned using the information provided by the oracle. Given a question associated with a node n of the ET, a NO answer prunes all the nodes of the ET except the subtree rooted at n; and a YES answer prunes the subtree rooted at n. Each strategy takes advantage of this property in a different manner.

A correct equation in the tree does not guarantee that the subtree rooted at this equation is free of errors. It can be the case that two buggy nodes caused the correct answer by fluke [Davie and Chitil, 2006]. In contrast, an incorrect equation does guarantee that the subtree rooted at this equation does contain a buggy node [Naish, 1997a]. Therefore, if a program produced a wrong result, then the equation in the root of the ET is wrong and thus there must be at least one buggy node in the ET. We will assume in the following that the debugging session has been started after discovering a bug symptom in the output of the program, and thus the root of the

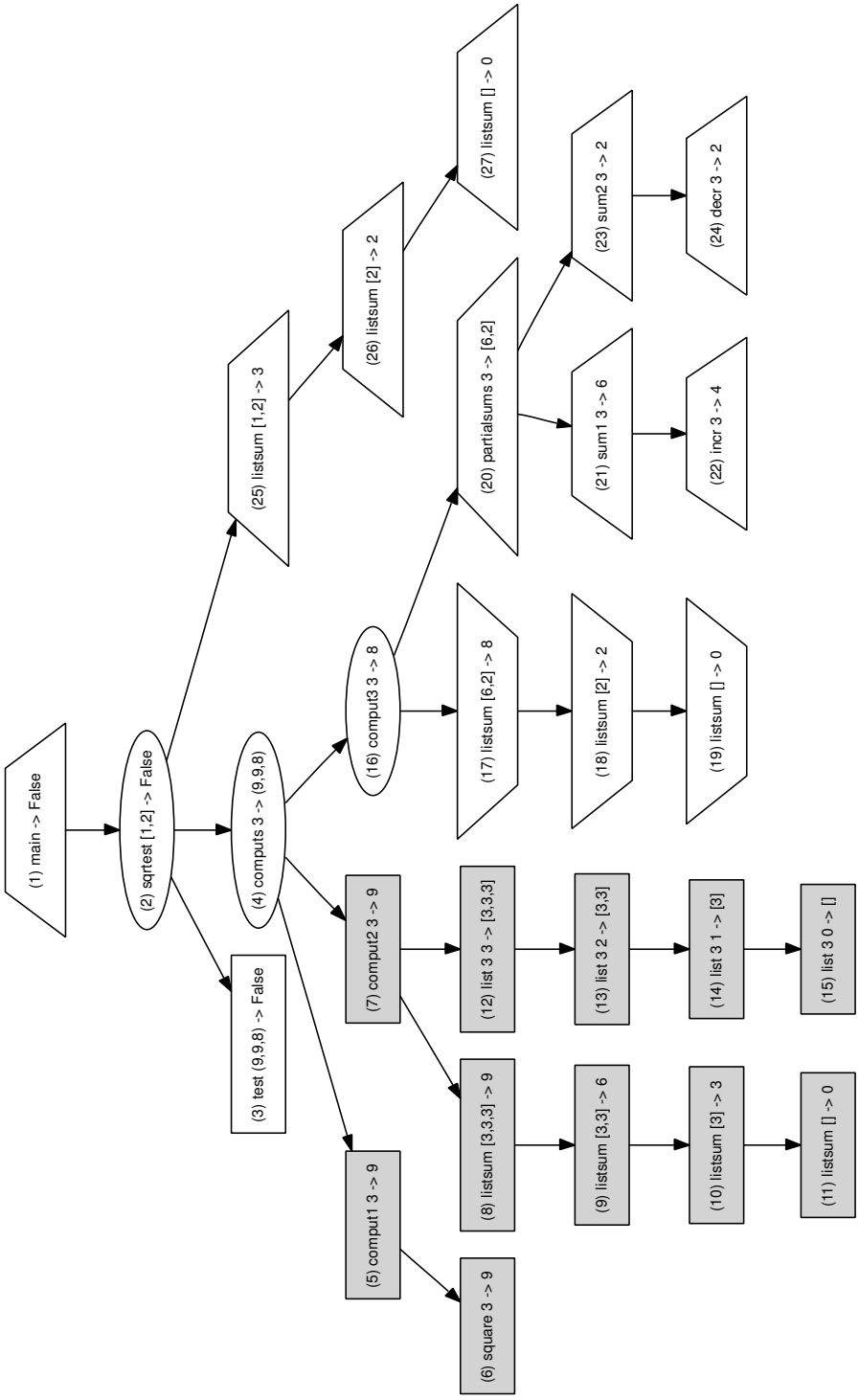


Figure 1.3: Execution tree of the program in Figure 1.1

tree contains a wrong equation. Hence, we know that there is at least one bug in the program. We will also assume that the oracle is able to answer all the questions. Then, all the strategies will find the bug.

Single Stepping *(Shapiro, 1982)*

The first algorithmic debugging strategy to be proposed was *single stepping* [Shapiro, 1982]. In essence, this strategy performs a bottom-up search because it proceeds by doing a post-order traversal of the ET. It asks first about all the children of a given node, and then (if they are correct) about the node itself. If the equation of this node is wrong then this is the buggy node; if it is correct, then the post-order traversal continues. Therefore, the first node answered NO is identified as buggy (because all its children have already been answered YES).

For instance, the sequence of 19 questions asked for the ET in Figure 1.3 is:

```
Starting Debugging Session...
(3)  test [9,9,8] = False? YES        (7)  comput2 3 = 9? YES
(6)  square 3 = 9? YES                (19) listsum [] = 0? YES
(5)  comput1 3 = 9? YES               (18) listsum [2] = 2? YES
(11) listsum [] = 0? YES              (17) listsum [6,2] = 8? YES
(10) listsum [3] = 3? YES             (22) incr 3 = 4? YES
(9)  listsum [3,3] = 6? YES           (21) sum1 3 = 6? YES
(8)  listsum [3,3,3] = 9? YES         (24) decr 3 = 2? YES
(15) list 3 0 = []? YES               (23) sum2 3 = 2? NO
(14) list 3 1 = [3]? YES
(13) list 3 2 = [3,3]? YES             Bug found in rule:
(12) list 3 3 = [3,3,3]? YES           sum2 x = div (x + (decr x)) 2
```

Note that in this strategy questions are semantically unconnected.

Top-Down Search *(Av-Ron, 1984)*

Due to the fact that questions are asked in a logical order, *top-down search* [Av-Ron, 1984] is the strategy that has been traditionally used (see, e.g., [Caballero, 2005; Kokai *et al.*, 1999]) to measure the performance of different debugging tools and methods. It basically consists in a top-down, left-to-right traversal of the ET and, thus, the node asked is always a child or a sibling of the previous question node. When a node is answered NO, one of its children is asked; if it is answered YES, one of its siblings is. Therefore, the idea is to follow the path of wrong equations from the root of the tree to the buggy node.

For instance, the sequence of 12 questions asked for the ET in Figure 1.3 is shown in Figure 1.2.

This strategy significantly improves single stepping because it prunes a part of the ET after every answer. However, it is still very naive, since it does not take into account the structure of the tree (e.g., how balanced it is). For this reason, a number of variants aiming at improving it can be found in the literature:

Top-Down Zooming *(Maeji and Kanamori, 1987)*

During the search of previous strategies, the rule or indeed the function definition may change from one query to the next. If the oracle is human, this continuous change of function definitions slows down the answers of the programmer because he has to switch thinking once and again from one function definition to another. This drawback can be partially overcome by changing the order of the questions: In this strategy [Maeji and Kanamori, 1987], recursive child calls are preferred.

The sequence of questions asked for the ET in Figure 1.3 is exactly the same as with top-down search (Figure 1.2) because no recursive calls are found.

Another variant of this strategy called *exception zooming*, introduced by Ian MacLarty [MacLarty, 2005], selects first those nodes that produced an exception at runtime.

Heaviest First *(Binks, 1995)*

Selecting always the left-most child does not take into account the size of the subtrees that can be explored. Binks proposed in [Binks, 1995] a variant of top-down search in order to consider this information when selecting a child. This variant is called *heaviest first* because it always selects the child with a bigger subtree. The objective is to avoid selecting small subtrees which have a lower probability of containing the bug.

For instance, the sequence of 9 questions asked for the ET in Figure 1.3 would be[3]:

```
Starting Debugging Session...
(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(4)  computs 3 = (9,9,8)? NO
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
```

[3]Here, and in the following, we will break the indeterminism by selecting the left-most node in the figures. For instance, the fourth question could be either (7) or (16) because both have a weight of 9. We selected (7) because it is on the left.

```
(24) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

Divide & Query *(Shapiro, 1982)*

In 1982, together with single stepping, Shapiro proposed another strategy: the so-called divide & query (D&Q) [Shapiro, 1982]. The idea of D&Q is to ask in every step a question which divides the remaining nodes in the ET by two, or, if this is not possible, into two parts with a weight as similar as possible. In particular, the original algorithm by Shapiro always chooses the heaviest node whose weight is less than or equal to $w/2$ where w is the weight of the suspicious area in the ET. This strategy has a worst case query complexity of order $b\ log_2\ n$ where b is the average branching factor of the tree and n its number of nodes.

This strategy works well with a large search space—this is normally the case of realistic programs—because its query complexity is proportional to the logarithm of the number of nodes in the tree. If the ET is big and unbalanced this strategy is better than top-down search [Caballero, 2005]; however, the main drawback of this strategy is that successive questions may have no connection, from a semantic point of view, with each other; requiring the programmer more time for answering the questions.

For instance, the sequence of 6 questions asked for the ET in Figure 1.3 is:

```
Starting Debugging Session...
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(21) sum1 3 = 6? YES
(24) decr 3 = 2? YES
(23) sum2 3 = 2? NO

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

Hirunkitti's Divide & Query *(Hirunkitti and Hogger, 1993)*

In [Hirunkitti and Hogger, 1993], Hirunkitti and Hogger noted that Shapiro's algorithm does not always choose the node closest to the halfway point in the tree and addressed this problem slightly modifying the original divide & query algorithm.

Their version of divide & query is the same as the one of Shapiro except that their version always chooses a node which produces a least difference between:

- $w/2$ and the heaviest node whose weight is less than or equal to $w/2$
- $w/2$ and the lightest node whose weight is greater than or equal to $w/2$

where w is the weight of the suspicious area in the computation tree.

For instance, the sequence of 6 questions asked for the ET in Figure 1.3 is:

```
Starting Debugging Session...
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(21) sum1 3 = 6? YES
(24) decr 3 = 2? YES
(23) sum2 3 = 2? NO

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

Biased Weighting Divide & Query *(MacLarty, 2005)*

MacLarty proposed in his PhD thesis [MacLarty, 2005] that not all the nodes should be considered equally while dividing the tree. His variant of D&Q divides the tree by only considering some kinds of nodes and/or by associating a different weight to every kind of node.

In particular, his algorithmic debugger was implemented for the functional logic language Mercury [Conway *et al.*, 1995] which distinguishes between 13 different node types.

Hat-delta *(Davie and Chitil, 2005)*

Hat [Wallace *et al.*, 2001] is a tracer for Haskell. Davie and Chitil introduced a declarative debugger tool based on the Hat's traces that includes a new strategy called Hat-delta [Davie and Chitil, 2005]. Initially, Hat-delta is identical to top-down search but it becomes different as the number of questions asked increases. The main idea of this strategy is to use previous answers of the oracle in order to compute which node has an associated rule that is more likely to be wrong (e.g., because it has been answered NO more times than the others).

This strategy assumes that a rule answered NO n times out of m is more likely to be wrong than a rule answered NO n' times out of m if $n' < n \leqslant m$. During a debugging session, a sequence of questions, each of them related to a particular

rule, is asked. In general, after every question, it is possible to compute the total number of questions asked for each rule, the total number of answers YES/NO, and the total number of nodes associated with this rule. Moreover, when a node is set correct or wrong, Hat-delta marks all the rules of its descendants as correctly or incorrectly executed respectively. This strategy uses all this information to select the next question. In particular, three different heuristics have been proposed based on this idea [Davie and Chitil, 2006]:

- *Counting the number of YES answers.* If a rule has been executed correctly before, then it will likely execute correctly again. The debugger associates to each rule of the program the number of times it has been executed in correct computations based on previous answers.
- *Counting the number of NO answers.* This is analogous to the previous heuristic but collecting wrong computations.
- *Calculating the proportion of NO answers.* This is derived from the previous two heuristics. For a node with associated rule r we have:

$$\frac{number\ of\ answers\ NO\ for\ r}{number\ of\ answers\ NO/YES\ for\ r}$$

  If r has not been asked before a value of $\frac{1}{2}$ is assigned.

Example 2 *Consider this program:*

```
4|0|0     sort [] = []
8|4|1/3   sort (x:xs) = insert x (sort xs)
4|0|0     insert x [] = [x]
          insert x (y:ys)
4|0|0       | x<y = x:y:ys
0|0|1/2     | otherwise = insert x ys
```

where the left numbers indicate respectively the number of times each rule has been executed correctly, the number of times each rule has failed and the proportion of NO answers for this rule.
With this information, `otherwise = insert x ys` *is more likely to be wrong.*

Subterm Dependency Tracking *(MacLarty et al., 2005)*

In 1986, Pereira [Pereira, 1986] noted that the answers YES, NO and *I don't know* were insufficient; and he pointed out another possible answer of the programmer: *Inadmissible* (see also [Naish, 1997b]). An equation or, more precisely, some of its

arguments, are inadmissible if they violate the preconditions of its function definition. For instance, consider the equation `insert 'b' "cc" = "bcc"`, where function `insert` inserts the first argument in a list of mutually different characters (the second argument). This equation is not wrong but inadmissible, since the argument "`cc`" has repeated characters. Hence, inadmissibility allows us to identify errors in left-hand sides of equations.

However, with only these four possible answers the system fails to get fundamental information from the programmer about *why* the equation is wrong or inadmissible. In particular, the programmer could specify which exact (sub)term in the result or the arguments is wrong or inadmissible respectively. This provides specific information about *why* an equation is wrong (i.e., which part of the result is incorrect? is one particular argument inadmissible?).

Consider again the equation `insert 'b' "cc" = "bcc"`. Here, the programmer could detect that the second argument should not have been computed; he could then mark the second argument ("`cc`") as inadmissible. This information is essential because it allows the debugger to avoid questions related to the correct parts of the equation and concentrate on the wrong parts.

Based on this idea, MacLarty et al. [MacLarty, 2005] proposed a new strategy called subterm dependency tracking. Essentially, once the programmer selects a particular wrong subterm, this strategy searches backwards in the computation for the node that introduced the wrong subterm. All the nodes traversed during the search define a *dependency chain* of nodes between the node that produced the wrong subterm and the node where the programmer identified it. The sequence of questions defined in this strategy follows the dependency chain from the origin of the wrong subterm.

For instance, if the programmer is asked question 3 from the ET in Figure 1.3, his answer would be YES but he could also mark subexpression "8" as inadmissible. Then, the system would compute the chain of nodes which passed this subexpression from the node which computed it until question 3. This chain is formed by nodes 4, 16 and 17. The system would ask first 17, then 16, and finally 4 following the computed chain.

In our example, the sequence of 8 questions asked for the ET in Figure 1.3, combining this strategy with top-down search, is:

```
Starting Debugging Session...
(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(3)  test (9,9,8) = False? YES  (the programmer marks ‘‘8")
(17) listsum [6,2] = 8? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO  (the programmer marks ‘‘2")
```

```
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

1.2.4 Abstract Debugging

Abstract interpretation is a formal method introduced by Cousot and Cousot [1977] to statically determine dynamic properties of programs. This method have also been used for the debugging of programs in a technique known as *abstract debugging*.

Abstract debugging [Bourdoncle, 1993] was first introduced in the context of higher-order imperative programs. In particular, its first implementation was the *Syntox* system which was able to debug Pascal programs.

The technique starts with the programmer inserting a set of assertions in the source code. This assertions can be *invariant assertions*, which are properties that must always hold at the control point where they are placed; and *intermittent assertions*, which are properties that must eventually hold at the control point where they are placed. The assertions introduced by the programmer are interpreted as the intended semantics of the program, thus their violation is considered a bug. Therefore, abstract interpretation is used to find under which conditions does the assertions are violated.

Example 3 *Consider the following Pascal program:*

```
 (1) program AbsDeb;
 (2)    var i,n: integer;
 (3)        T: array [1..100] of integer;
 (4) begin
 (5)    read(n);
 (6)    for i:=0 to n do
*(7)       read(T[i]);
 (8) end.
```

*where the user introduced an assertion in the line marked with * specifying that, at this line,* $1 \leq i \leq 100$ *(i.e., the array's index is not out of bounds). An abstract debugger would propagate this condition backwards, and would determine that it is violated in every execution of the program because the assignment* `i := 0`*. Then, the programmer would correct this bug by replacing this assignment by* `i := 1`*. If the abstract debugger is executed again with the correct program, it would determine that the assertion holds whenever* $1 \leq n \leq 100$ *at line* (5)*. This is not a bug, but a precondition of the program to ensure correctness.*

Abstract debugging has been later applied in the logic paradigm by Comini *et al.* [1994]. In particular, they have investigated the use of this technique in the context of positive logic programs under the left-most selection rule.

1.2.5 Tracing

Tracing is a technique which relies in the construction of a data structure (a trace) representing the execution of a program. This data structure is usually computed during the execution of a program and can be later explored in order to get information about the execution itself. The fact of being able to 'record' the execution of a program in a data structure (the trace) and reproduce it forwards and/or backwards is also known as *execution backtracking* [Agrawal *et al.*, 1991].

A trace liberates from the time arrow of computation, thus it is possible to inspect a trace forwards or backwards looking how redexes were produced during the computation.

A trace can contain information about the proper execution of the program being debugged, but it can also contain other information (e.g., compiler information, source code data, etc) so that many properties can be observed and improved. One of the most appropriate data structures used for tracing in the context of declarative languages are the *redex trails*, originally introduced by Sparud and Runciman [1997], and its subsequent extension, the *Augmented Redex Trails (ART)* [Wallace *et al.*, 2001]. Redex trails will be presented and used in Chapter 5.

Tracing debuggers present to the user a view of the reductions that the compiler did while evaluating a program. However, the subtle pragmatics of lazy evaluation can lead to intricate program structures that produce very complex or illegible traces. One of the solutions adopted is to provide the user with special tools that allow us to browse traces at selected stages in the computation and inspect them by showing the information in a more intuitive manner, for instance by showing the arguments of functions in their most evaluated form.

Typically, two major technical problems that tracing debuggers need to overcome are related to scaling up to large applications. The size of the traces can be huge (indeed gigabytes). For nontrivial problems, either a stream approach is used (never storing the whole trace to disk or memory), or only selected parts of the computation are traced (e.g., by trusting some modules and/or functions). The structures that the user can view and browse are also huge. Sometimes, the user can be prompted with enormous data structures which are hardly understandable and, thus, useless. As an example, the algorithmic debugger of Mercury [MacLarty, 2005] was used to debug the own Mercury compiler—a program with more than 300.000 lines of code— and during the debugging session the user was prompted with a question of more than 1400 lines.

1.2.6 Program Slicing

Program slicing was originally introduced in 1984 by Mark Weiser. Since then, many researchers have extended it in many directions and for all programming paradigms. The huge amount of program slicing-based techniques has lead to the publication of different surveys [Tip, 1995; Binkley and Gallagher, 1996; Harman *et al.*, 1996; Harman and Gallagher, 1998; De Lucia, 2001; Harman and Hierons, 2001; Binkley and Harman, 2004; Xu *et al.*, 2005] trying to clarify the differences between them. However, each survey presents the techniques from a different perspective. For instance, [Tip, 1995; Binkley and Gallagher, 1996; Harman and Gallagher, 1998; De Lucia, 2001; Harman and Hierons, 2001; Xu *et al.*, 2005] mainly focus on the advances and applications of program slicing-based techniques; in contrast, [Binkley and Harman, 2004] focuses on their implementation comparing empirical results; and [Harman *et al.*, 1996] tries to compare and classify them in order to predict future techniques and applications. In this section we describe all the techniques in a way similar to [De Lucia, 2001]. We propose an example program and we extract a slice from it by applying every technique.

Program Slicing *(Weiser, 1984)*

The original ideas of program slicing come from the PhD thesis of Mark Weiser in 1979 [Weiser, 1979] that were presented in the *International Conference on Software Engineering* in 1981 [Weiser, 1981] and finally published in *Transactions on Software Engineering* in 1984 [Weiser, 1984].

Program slicing is a technique to decompose programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those program statements which are (potentially) related to the values computed at some program point and/or variable, referred to as a *slicing criterion*.

As it was originally defined:

> "*A slice is itself an executable program subset of the program whose behavior must be identical to the specified subset of the original program's behavior*"

However, the constraint of being an executable program has been sometimes relaxed. Given a program p, slices are produced w.r.t a given slicing criterion $\langle s, v\rangle$ which specifies a sentence s and a set of variables v in p. Note that the variables in v not necessarily appear in s; consider for instance the slicing criterion $\langle 18, \{lines, chars, subtext\}\rangle$ for the program in Figure 1.4 (a). Observe that, in this case, not all the variables appear in line 18.

As an example of slice, consider again the program in Figure 1.4 (a) (for the time being the reader can ignore the breakpoints marked with $*$) where function

```
*(1)  read(text);                              (1)  read(text);
 (2)  read(n);
 (3)  lines = 1;                               (3)  lines = 1;
 (4)  chars = 1;
 (5)  subtext = "";
 (6)  c = getChar(text);                       (6)  c = getChar(text);
 (7)  while (c != '\eof')                      (7)  while (c != '\eof')
 (8)       if (c == '\n')                      (8)       if (c == '\n')
 (9)       then lines = lines + 1;             (9)       then lines = lines + 1;
*(10)            chars = chars + 1;
 (11)      else chars = chars + 1;
 (12)           if (n != 0)
*(13)           then subtext = subtext ++ c;
 (14)                n = n - 1;
 (15)      c = getChar(text);                  (15)      c = getChar(text);
 (16) write(lines);                            (16) write(lines);
 (17) write(chars);
*(18) write(subtext);

          (a) Example program                              (b) Slice
```

Figure 1.4: Example of slice

`getChar` extracts the first character of a string. This program takes a text (i.e., a string of characters including '\n'—carriage return—and '\eof'—end-of-file—) and a number n, and it returns the number of characters and lines of the text and a subtext composed of the first n characters excluding '\n'. A slice of this program w.r.t. the slicing criterion $\langle 16, lines \rangle$ is shown in Figure 1.4 (b).

Static Slicing *(Weiser, 1984)*

The original definition of program slicing was static [Weiser, 1984], in the sense that it did not consider any particular input for the program being sliced. Particularly, the slice shown in Figure 1.4 (b) is a static slice, because it does not consider any particular execution (i.e., it works for any possible input data).

In order to extract a slice from a program, the dependences between its statements must be computed first. The *Control Flow Graph* (CFG) is a data structure which makes the control dependences for each operation in a program explicit. For instance, the CFG of the program in Figure 1.4 (a) is depicted in Figure 1.5.

However, the CFG does not generally suffice for computing program slices because it only stores control dependences and, for many applications (such as debugging),

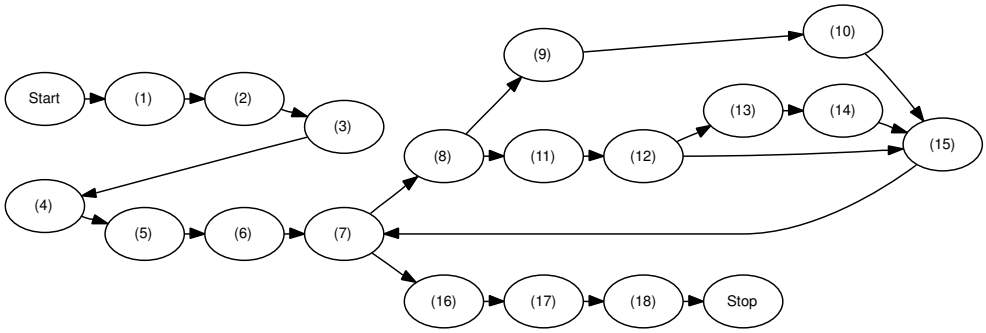


Figure 1.5: Control flow graph of Figure 1.4 (a)

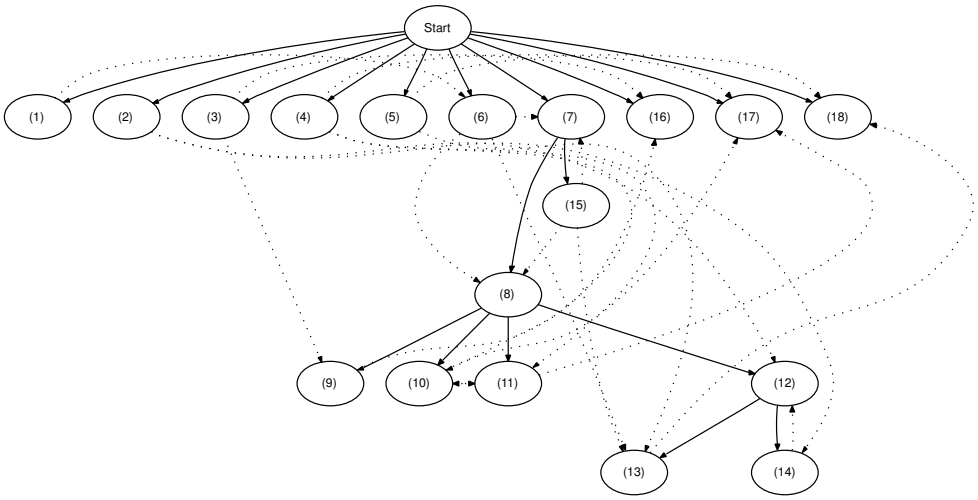


Figure 1.6: Program dependence graph of Figure 1.4 (a)

data dependences are also necessary. Ottenstein and Ottenstein [1984] noted that the *Program Dependence Graph* (PDG) [Kuck *et al.*, 1981; Ferrante *et al.*, 1987] was the "ideal" data structure for program slicing because it makes explicit both the data and control dependences for each operation in a program. In essence, a PDG is an oriented graph where the nodes represent statements in the source code and edges represent control and data dependences between statements in such a way that they induce a partial ordering in the nodes preserving the semantics of the program. As an example, the PDG of the program in Figure 1.4 (a) is depicted in Figure 1.6 where solid arrows represent control dependences and dotted arrows represent data dependences. Here, nodes are labeled with the number of the sentence they represent; except node *Start* which represents the start of the program. The solid arrow between nodes 7 and 8 indicates that the execution of sentence 8 depends on the execution of sentence 7. The

same happens with the solid arrow between sentences 8 and 12; thus, transitively, the execution of sentence 12 also depends on the execution of sentence 7.

The slice shown in Figure 1.4 (b) is a static slice w.r.t. the slicing criterion $\langle 18, lines \rangle$.

Dynamic Slicing *(Korel and Laski, 1988)*

One of the main applications of program slicing is debugging. Often, during debugging, the value of a variable v at some program statement s is observed to be incorrect. A program slice w.r.t. $\langle s, v \rangle$ contains the cause of the error.

However, in such a case, we are interested in producing a slice formed by those statements that could cause this particular error, i.e., we are only interested in one specific execution of the program. Such slices are called dynamic slices [Korel and Laski, 1988], since they use dynamic information during the process. In general, dynamic slices are much smaller than static ones because they contain the statements of the program that affect the slicing criterion for *a particular* execution (in contrast to *any* execution as with static slicing).

During a program execution, the same statement can be executed several times in different contexts (i.e., with different values of the variables); as a consequence, pointing out a statement in the program is not enough in dynamic slicing. A dynamic slicing criterion needs to specify which particular execution of the statement during the computation is of interest; thus it is defined as $\langle s^i, v, \{a_1 \dots a_n\} \rangle$, where i is the number of occurrence of statement s in the execution history; v is the set of variables we are interested in, and the set $\{a_1 \dots a_n\}$ are the initial values of the program's input. As an example, consider the input values $\{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}$ for the program in Figure 1.4 (a). The execution history would be:

$$(1, 2, 3, 4, 5, 6, 7, (8, 11, 12, 13, 14, 15, 7)^{12}, 16, 17, 18)$$

where the superscript 12 indicates that the statements inside the parenthesis are repeated twelve times in the execution history.

A dynamic slice of the program in Figure 1.4 (a) w.r.t. the slicing criterion $\langle 18^{94},\ \{lines\},\ \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\} \rangle$ is shown in Figure 1.7. Note that this slice is much smaller than its static counterpart because here the slicer can compute the specific control and data-flow dependences produced by the provided input data. However, it comes with a cost: the computation of such dependences usually implies the computation of an expensive (measured in time and space) and complex data structure (e.g., a trace [Sparud and Runciman, 1997]).

```
(3)  lines = 1;
(16) write(lines);
```

Figure 1.7: Dynamic slice of Figure 1.4 (a) with respect to the slicing criterion $\langle 18^{94},\ \{lines\},\ \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$

Backward Slicing *(Weiser, 1984)*

A program can be traversed forwards or backwards from the slicing criterion. When we traverse it backwards (the so called backward slicing), we are interested in all those statements that could influence the slicing criterion. In contrast, when we traverse it forwards (the so called forward slicing), we are interested in all those statements that could be influenced by the slicing criterion.

The original method by Weiser—described above—was static backward slicing. The main applications of backward slicing are debugging, program differencing and testing. As an example, the slice shown in Figure 1.4 (b) is a backward slice of the program in Figure 1.4 (a).

Forward Slicing *(Bergeretti and Carré, 1985)*

Forward slicing [Bergeretti and Carré, 1985] allows us to determine how a modification in a part of the program will affect other parts of the program. As a consequence, it has been used for dead code removal and for software maintenance. In this context, Reps and Bricker were the first to use the notion of forward slice [Reps and Bricker, 1989]. However, despite backward slicing is the preferred method for debugging, forward slicing has also been used for this purpose. In particular, forward slicing can detect initialization errors [Gaucher, 2003].

A forward static slice of the program in Figure 1.4 (a) w.r.t. the slicing criterion $\langle 3, \{lines\}\rangle$ is shown in Figure 1.8.

```
(3)  lines = 1;
(9)        lines = lines + 1;
(16) write(lines);
```

Figure 1.8: Forward static slice of Figure 1.4 (a) w.r.t. $\langle 3, \{lines\}\rangle$

Chopping *(Jackson and Rollins, 1994)*

In chopping [Jackson and Rollins, 1994], the slicing criterion selects two sets of variables, *source* and *sink*, and then it computes all the statements in the program that

being affected by source, they affect sink. Therefore, chopping is a generalization of both forward and backward slicing where either source or sink is empty. As noted by Reps and Rosay [Reps and Rosay, 1995], chopping is particularly useful to detect those statements that "transmit effects" from one part of the program (source) to another (sink).

For instance, a chop of the program in Figure 1.4 (a) w.r.t. the slicing criterion $\langle source = \{(3, \{lines\})\},\ sink = \{(9, \{lines\})\}\rangle$ is shown in Figure 1.9.

```
(3)  lines = 1;
(9)        lines = lines + 1;
```

Figure 1.9: Chop of Figure 1.4 (a) with respect to the slicing criterion $\langle source = \{(3, \{lines\})\},\ sink = \{(9, \{lines\})\}\rangle$

Relevant Slicing *(Agrawal, Horgan, Krauser and London, 1993)*

A dynamic program slice only contains the program statements that actually affect the slicing criterion. However, it is sometimes (e.g., in debugging) interesting to include in the slice those statements that *could have affected* the slicing criterion. This is the objective of relevant slicing [Agrawal *et al.*, 1993], which computes all the statements that *potentially* affect the slicing criterion.

A slicing criterion for relevant slicing is exactly the same as for dynamic slicing, but if we compute a relevant slice and a dynamic slice w.r.t. the same slicing criterion, the relevant slice is a superset of the dynamic slice because it contains all the statements of the dynamic slice and it also contains those statements of the program that did not affect the slicing criterion but could have affected it if they would have changed (for instance because they were faulty).

Let us explain it with an example: consider the previously used slicing criterion $\langle 18^{94}, \{lines\}, \{text = \text{"}hello\ world!\backslash eof\text{"}, n = 4\}\rangle$. The relevant slice computed is shown in Figure 1.10. It contains all the statements of the dynamic slice (see Figure 1.7) and also it includes statements `(6)` and `(8)`. Statement `(6)` could have influenced the value of `lines` being redefined to "`(6) c = '\n';`" and statement `(8)` could have influenced the value of `lines` being redefined to "`(8) if (c != '\n')`".

It should be clear that the slices produced in relevant slicing can be non-executable. The reason is that the main application of such a technique is debugging, where the programmer is interested in those parts of the program that can contain the bug. The statements included in the slice are those whose contamination could result in the contamination of the variable of interest. In order to make the slice executable preserving the behavior (including termination) of the original program it is necessary

to augment the slice with those statements that are required for the evaluation of all the expressions included in the slice (even if this evaluation does not influence the variable of interest).

The forward version of relevant slicing [Gyimóthy *et al.*, 1999] can be useful in debugging and in program maintenance. A forward relevant slice contains those statements that *could be affected* by the slicing criterion (if it is redefined). Therefore, it could be used to study module cohesion by determining what could be the impact of a module modification over the rest of modules.

```
(3)  lines = 1;
(6)  c = getChar(text);
(8)       if (c == '\n')
(16) write(lines);
```

Figure 1.10: Relevant slice of Figure 1.4 (a) with respect to the slicing criterion $\langle 18^{94}, \{lines\}, \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$

Hybrid Slicing *(Gupta and Soffa, 1995)*

When debugging, it is usually interesting to work with dynamic slices because they focus on a particular execution (i.e., the one that showed a bug) of the program being debugged; therefore, they are much more precise than static ones. However, computing dynamic slices is very expensive in time and space due to the large data structures (up to gigabytes) that need to be computed. In order to increase the precision of static slicing without incurring in the computation of these large structures needed for dynamic slicing, a new technique called hybrid slicing [Gupta and Soffa, 1995] was proposed. A hybrid slice is more accurate—and consequently smaller—than a static one, and less costly than a dynamic slice.

The key idea of hybrid slicing consists in integrating dynamic information into the static analysis. In particular, the information provided to the slicer is a set of breakpoints inserted into the source code that, when the program is executed, can be activated thus providing information about which parts of the code have been executed. This information allows the slicer to eliminate from the slice all the statements which are in a non-executed possible path of the computation.

For instance, consider again the program in Figure 1.4 (a) which contains 4 breakpoints marked with '$*$'. A particular execution will activate some of the breakpoints; this information can be used by the slicer. For instance, a possible execution could be $\{1, 13, 13, 13, 10, 13, 18\}$ which means that the loop has been entered 5 times, and both branches of the outer if-then-else have been executed.

A hybrid slicing criterion is a triple $\langle s, v, \{b_1 \dots b_n\}\rangle$, where s and v have the same meaning than in static slicing and the set $\{b_1 \dots b_n\}$ are the sequence of breakpoints activated during a particular execution of the program.

Figure 1.11 shows an example of hybrid slice.

```
(1)  read(text);
(4)  chars = 1;
(6)  c = getChar(text);
(7)  while (c != '\eof')
(8)       if (c == '\n')
(11)      then chars = chars + 1;
(15)      c = getChar(text);
(17) write(chars);
```

Figure 1.11: Hybrid slice of Figure 1.4 (a) with respect to the slicing criterion $\langle 18, \{chars\}, \{1, 13, 13, 13, 18\}\rangle$

Intraprocedural Slicing *(Weiser, 1984)*

The original definition of program slicing has been later classified as *intraprocedural slicing* (i.e., the slice in Figure 1.4 (b) is an intraprocedural slice), because the original algorithm was defined for programs consisting of a single monolithic procedure; in other words it did not take into account information related to the fact that slices can cross the boundaries of procedure calls.

It does not mean that the original definition fails to slice multi-procedural programs; it means that it loses precision in such cases. As an example, consider the program in Figure 1.12. A static backward slice of this program w.r.t. the slicing criterion $\langle 17, \{a\}\rangle$ includes all the statements of the program except (12), (13) and (14). However, it is clear that statements (3) and (8) included in the slice cannot affect the slicing criterion. They are included in the slice because procedure `sum` influences the slicing criterion, and statements (3) and (8) can influence procedure `sum`. However, they cannot transitively affect the slicing criterion. This loss of precision has been later solved by another technique called *interprocedural slicing*.

Interprocedural Slicing *(Horwitz, Reps and Binkley, 1988)*

In 1988, Horwitz et al. noted that the program dependence graph was not appropriate for representing multi-procedural programs and they proposed a new dependence graph representation of programs called *system dependence graph* [Horwitz *et al.*, 1988]. This new representation incorporates collections of procedures with procedure

```
(1)  program main
(2)  read(text);
(3)  lines = 1;
(4)  chars = 1;
(5)  c = getChar(text);
(6)  while (c != '\eof')
(7)       if (c == '\n')
(8)       then lines = sum(lines,1);
(9)            chars = sum(chars,1);
(10)      else chars = increment(chars);
(11)      c = getChar(text);
(12) write(lines);
(13) write(chars);
(14) end

(15) procedure increment(a)
(16) sum(a,1);
(17) return

(18) procedure sum(a, b)
(19) a = a + b;
(20) return
```

Figure 1.12: Example of multiprocedural program

calls, and it allows us to produce more precise slices from multi-procedural programs because it has information available about the actual procedures' calling-context.

For instance, consider `Program 1` in Figure 1.18 together with the slicing criterion $\langle 3, \{chars\}\rangle$. An intraprocedural static slice contains exactly the same sentences (`Program 1`). However, it is clear that procedure `increment` cannot affect the slicing criterion (in fact it is dead code). Roughly speaking, this inaccuracy is due to the fact that the call to procedure `sum` makes the slice include this procedure. Then, all the calls to this procedure can influence it and thus, procedure `increment` is also included.

In contrast, an interprocedural static slice (`Program 2`) uses information stored in the system dependence graph about the calling context of procedures. Hence, it would remove procedure `increment` thus increasing the precision w.r.t. the intraprocedural algorithms.

Figure 1.13 shows the interprocedural slice of the program in Figure 1.12 w.r.t.

the slicing criterion $\langle 17, \{a\}\rangle$.

```
(1)  program main
(2)  read(text);
(4)  chars = 1;
(5)  c = getChar(text);
(6)  while (c != '\eof')
(7)       if (c == '\n')
(9)       then chars = sum(chars,1);
(10)      else chars = increment(chars);
(11)      c = getChar(text);

(15) procedure increment(a)
(16) sum(a,1);
(17) return

(18) procedure sum(a, b)
(19) a = a + b;
(20) return
```

Figure 1.13: Interprocedural slice of Figure 1.12 w.r.t. $\langle 17, \{a\}\rangle$

Quasi-Static Slicing *(Venkatesh, 1991)*

While static slicing computes slices w.r.t. any execution, dynamic slicing computes slices w.r.t. a particular execution. However, it is sometimes (e.g., in program understanding) interesting to produce a slice w.r.t. a particular set of executions. Quasi-static slicing [Venkatesh, 1991] can be used in those applications in which a set of the program inputs are fixed, and the rest of the inputs is unknown. This leads to a potentially infinite set of considered executions.

A quasi-static slicing criterion is a tuple $\langle s, v, a\rangle$ where s and v have the same meaning as in static slicing, and a is a mapping from (some of the) input variables to values. For instance, a quasi-static slicing criterion for the program in Figure 1.4 (a) could be $\langle 18, \{subtext\}, \{n = 0\}\rangle$. The slice computed w.r.t. this criterion is shown in Figure 1.14.

Simultaneous Slicing *(Hall, 1995)*

Simultaneous slicing is a generalization of program slicing in which a set of slicing criteria is considered. Hence, a set of points of interest is provided and thus the slice

```
(1)  read(text);
(5)  subtext = "";
(18) write(subtext);
```

Figure 1.14: Quasi-static slice of Figure 1.4 (a) w.r.t. $\langle 18, \{subtext\}, \{n = 0\}\rangle$

can be computed from the slices computed for each point. However, the construction of such a slice does not simply reduces to the union of slices (this is not sound) and it requires the use of more elaborated methods such as the SDS procedure [Hall, 1995].

Similarly to quasi-static slicing, simultaneous dynamic slicing computes a slice w.r.t. a particular set of executions; however, while quasi-static slicing fixes a set of the program inputs being the rest unknown, in simultaneous dynamic slicing [Hall, 1995] a set of "*complete*" inputs is known. Thus, a simultaneous dynamic slicing criterion can be seen as a set of dynamic slicing criterions.

A simultaneous dynamic slicing criterion has the form: $\langle s^i, v, \{I_1 \dots I_n\}\rangle$ where i is the number of occurrence of statement s in the execution history, v is the set of variables of interest and $\{I_1 \dots I_n\}$ is a set of complete inputs for the program. For instance, a slice of Figure 1.4 (a) w.r.t. the slicing criterion $\langle 18^{94}, \{lines\}, \{I_1, I_2\}\rangle$ where $I_1 = (text =$ "$hello\ \ world!\backslash eof$", $n = 4)$ and $I_2 = (text =$ "$hello\backslash nworld!\backslash n\ \backslash eof$", $n = 0)$ is depicted in Figure 1.15.

Note that, in contrast to quasi-static slicing where the set of considered executions can be infinite, in simultaneous dynamic slicing it is always finite.

```
(1)  read(text);
(2)  read(n);
(4)  chars = 1;
(6)  c = getChar(text);
(7)  while (c != '\eof')
(8)       if (c == '\n')
(9)       then lines = lines + 1;
(10)           chars = chars + 1;
(11)      else chars = chars + 1;
(15)      c = getChar(text);
(17) write(chars);
```

Figure 1.15: Simultaneous dynamic slice of Figure 1.4 (a) w.r.t. $\langle 18^{94}, \{lines\}, \{I_1, I_2\}\rangle$, where $I_1 = (text =$ "$hello\ \ world!\backslash eof$", $n = 4)$ and $I_2 = (text =$ "$hello\backslash nworld!\backslash n\backslash eof$", $n = 0)$

In 1993, Lakhotia [1993] introduced a new kind of slicing criterion in order to compute module cohesion. The aim of his work was to collect the module's components which contribute to the final value of the output variables of this module. Therefore, in this scheme, the slicing criterion is formed by a set of variables. It is called "end slicing" because the slicing point is the end of the program. Then, an end slicing criterion is formed by a set of variables of the program, and thus it can be represented as a set of slicing criteria (i.e., it is a particular case of simultaneous static slicing).

Program Dicing *(Lile and Weiser, 1987)*

Program dicing [Lyle and Weiser, 1987] was originally defined as: "*remove those statements of a static slice of a variable that appears to be correctly computed from the static slice of an incorrectly valued variable*". From this definition it is easy to deduce that program dicing was originally defined for debugging. In essence, a dice is the set difference between at least two slices; typically, one of an incorrect value and the other of a correct value. The main idea is that after some tests we can find some correct and incorrect outputs. The statements in a slice of an incorrect output that do not belong to the slice of a correct output are more likely to be wrong. As a consequence, a program dicing criterion specifies n points in the program, one for the correct computation and $n-1$ for the wrong computations. For instance, a backward static program dicing criterion for the program in Figure 1.4 (a) could be $\langle(18,\{chars\}),\{(18,\{lines\})\}\rangle$; its meaning is that variable *chars* at line 18 produced a correct value (its slice contains the statements 1,4,6,7,8,10,11,15 and 17) whereas variable *lines* at line 18 produced an incorrect value (its slice contains the statements 1,3,6,7,8,9,15 and 16). Note that only one point has been specified for the incorrect computation, but a set is possible. The dice computed w.r.t. this criterion is shown in Figure 1.8.

In contrast to program slicing where a dynamic slice is included in its corresponding static slice, a dynamic dice is not necessarily included in its corresponding static dice. Chen and Cheung [1993] investigated under what conditions a dynamic dice is more precise than a static dice.

Conditioned/Constraint Slicing *(Ning, Engberts and Kozaczynski, 1994)*

Although Ning *et al.* [1994] were the first to work with conditioned slices; this technique was formally defined for the first time by Canfora *et al.* [1994].

Similarly to simultaneous dynamic slicing and quasi-static slicing, conditioned slicing [Canfora *et al.*, 1994, 1998] computes slices w.r.t. a set of initial states of the program. The original definition proposed the use of a condition (from the programming language notation) to specify the set of initial states. Posteriorly, Field *et al.* [1995] proposed the same idea (known as parametric program slicing or constraint

slicing) based on constraints over the initial values. Finally, De Lucia *et al.* [1996] proposed the condition to be a universally quantified formula of first-order predicate logic. In this approach, which is a generalization of the previous works, a conditioned slicing criterion is a quadruple $\langle i, F, s, v\rangle$ where i is a subset of the input variables of the program, F is a logic formula on i, s is a statement of the program and v a subset of the variables in the program.

The logic formula F identifies a set of the possible inputs of the program which could be infinite. For instance, consider the conditioned slicing criterion $\langle(text, n), F,$ $18, \{subtext\}\rangle$ where $F = (\forall c \in text, c \neq$ '$\backslash n$', $n > 0)$; the conditioned slice of the program in Figure 1.4 (a) w.r.t. this criterion is shown in Figure 1.16.

It should be clear that conditioned slicing is a generalization of both simultaneous dynamic slicing and quasi-static slicing because their respective slicing criteria are a particular case of a conditioned slicing criterion. As an advantage, conditioned slicing allows us to specify relations between input values. For instance, condition F above specifies that if `text` is a single line, then `n` must be higher than 0.

```
(1)  read(text);
(2)  read(n);
(5)  subtext = "";
(6)  c = getChar(text);
(7)  while (c != ‘\eof’)
(8)       if (c == ‘\n’)
(12)           if (n != 0)
(13)           then subtext = subtext ++ c;
(14)                n = n - 1;
(15)      c = getChar(text);
(18) write(subtext);
```

Figure 1.16: Conditioned slice of Figure 1.4 (a) with respect to the slicing criterion $\langle(text, n), F, 18, \{subtext\}\rangle$ where $F = (\forall c \in text, c \neq$ '$\backslash n$', $n > 0)$

Abstract Slicing *(Hong, Lee and Sokolsky, 2005)*

Static slicing is able to determine for each statement in a program whether it affects or is affected by the slicing criterion; however, it is not able to determine *under which variable values* do the statements affect or are affected by the slicing criterion. This problem is overcome by abstract slicing [Hong *et al.*, 2005].

Essentially, abstract slicing extends static slicing with predicates and constraints that are processed with an abstract interpretation and model checking based technique. Given a predicate of the program, every statement is labeled with the value

of this predicate under the statement affects (or is affected by) the slicing criterion. Similarly, given a constraint on some variables of the program, every statement is labeled indicating if they can or cannot satisfy the constraint. Clearly, static slicing is a particular instance of abstract slicing where neither predicates nor constraints are used. Moreover, in a way similar to conditioned slicing, abstract slicing is able to produce slices w.r.t. a set of executions by restricting the allowed inputs of the program using constraints.

In abstract slicing, a slicing criterion has the form $\langle s, P, C\rangle$ where s is a statement of the program and P and C are respectively a predicate and a constraint for some statements defined over some variables of the program. For instance, C could be $((1), y > x)$, meaning that, at statement (1), the condition $y > x$ holds. Bases on previous slicing techniques, Hong et al. adapted this technique to forward/backward slicing and chopping, giving rise to abstract forward/backward slicing and abstract chopping.

As an example, consider the program in Figure 1.17 (a); the abstract slice of this program w.r.t. the slicing criterion $\langle(6), y > x, True\rangle$ is depicted in Figure 1.17 (b). Here, we are interested in knowing the condition under each statement affects the slicing criterion; we do not impose any condition. In the slice we see that the statement labeled with '$[true]$' will always affect the slicing criterion, '$max = x$' will only affect the slicing criterion if $y \leq x$, and the 'if-then' statements will affect the slicing criterion if $y > x$.

```
(1)  read(x);
(2)  read(y);
(3)  max = x;
(4)  if (y > x)
(5)  then max = y;
(6)  write(max);
```

(a) Example program

```
(1)  read(x);        [true]
(2)  read(y);        [true]
(3)  max = x;        [not(y>x)]
(4)  if (y > x)      [y>x]
(5)  then max = y;   [y>x]
(6)  write(max);     [true]
```

(b) Abstract slice

Figure 1.17: Abstract slicing: (b) is an abstract slice of (a) w.r.t. $\langle(6), y > x, True\rangle$

Amorphous Slicing *(Harman and Danicic, 1997)*

All approaches to slicing discussed so far have been based on two assumptions: the slice preserves (part of) the semantics of the program, and it is "syntax preserving", i.e., the slice is a subset of the original program statements. In contrast, amorphous slices [Harman and Danicic, 1997] preserve the semantics restriction but they drop the

syntactic restriction: amorphous slices are constructed using some program transformation which simplifies the program and which preserves the semantics of the program with respect to the slicing criterion.

This syntactic freedom allows amorphous slicing to perform greater simplifications, thus often being considerably smaller than conventional program slicing. This simplifications are very convenient in the context of program comprehension where the user needs to simplify the program as much as possible in order to understand a part of the semantics of the program having the syntax less importance.

For instance, consider `Program 1` in Figure 1.18 together with the slicing criterion $\langle 3, \{chars\}\rangle$. An intraprocedural static slice of this program would contain exactly the same sentences (`Program 1`). In contrast, an interprocedural static slice would remove sentences 4, 5 and 6 (`Program 2`). Finally, its amorphous slice would be `Program 3`. It should be clear that the three programs preserve the same semantics, but `Program 3` has been further simplified by partially evaluating [Jones, 1996] some expressions (thus changing the syntax) of `Program 1`. Clearly, `Program 3` is much more understandable than `Program 1` and `Program 2`.

```
(1) program main            (1) program main            (1) program main
(2) chars = sum(chars,1);   (2) chars = sum(chars,1);   (2) chars = chars + 1;
(3) end                     (3) end                     (3) end

(4) procedure increment(a)
(5) sum(a,1);
(6) return

(7) procedure sum(a,b)      (7) procedure sum(a,b)
(8) a = a + b;              (8) a = a + b;
(9) return                  (9) return

      Program 1                   Program 2                   Program 3
```

Figure 1.18: Example of amorphous slicing

It is known [Harman and Danicic, 1997] that amorphous static slicing subsumes traditional static slicing, i.e., there will always be an amorphous static slice which is at least as thin as the static slice constructed for the same slicing criterion. In addition, there will usually be an amorphous slice which is thinner than the associated static slice. This leads to the search of the *minimal* amorphous slice. However, as proved by Harman and Danicic [Harman and Danicic, 1997], the computation of the minimal amorphous static slice of an arbitrary program is, in general, undecidable.

Decomposition Slicing *(Gallagher and Lyle, 1991)*

Decomposition slicing [Gallagher and Lyle, 1991] was introduced in the context of software maintenance to capture all computation on a given variable. The objective of this technique is to extract those program statements which are needed to compute the values of a given variable. Therefore, a decomposition slicing criterion is composed of a single variable v. The slice is then built from the union of the static backward slices constructed for the criteria $\{\langle n_1, v\rangle, \ldots, \langle n_m, v\rangle, \langle end, v\rangle\}$ where $\{n_1, \ldots, n_m\}$ is the set of lines in which v is output and *end* is the "End" of the program. It should be clear that decomposition slicing is an instance of simultaneous slicing where the set of slicing criteria is derived from the variable and the program.

A decomposition slice of the program in Example 1.4 (a) w.r.t. the slicing criterion $\langle lines\rangle$ is shown in Figure 1.4 (b).

1.3 Structure of the Thesis

The core of this document has been organized in four parts. The first part presents some foundations. First, some preliminary definitions related to term rewriting systems and several basic concepts that will be used in the rest of the thesis are formally described. Both the syntax and the semantics of the languages that will be later used are introduced in Chapter 2. In particular, the multi-paradigm language Curry is presented together with its internal representation, namely the flat programs and their normalized version.

The second part contains two chapters which present two profiling techniques for functional logic languages. First, in Chapter 3, we introduce a technique to formally reason about program execution costs by computing precise time equations. This technique computes the number of function unfoldings performed during the evaluation of a program. This chapter also includes a discussion about how the costs computed for a particular execution could be used to reason about the general cost of the program.

In Chapter 4 it is proposed a new technique for the development of profiling tools and it presents its application to the Curry language. This new profiling scheme allows us to measure different kinds of symbolic costs and attribute them to different cost centers distributed over the program. The technique is based on the instrumentation of a semantics with costs. However, in order to avoid the big overhead introduced by this approach, a program transformation which is equivalent to the instrumented semantics is also introduced.

The third part introduces a new dynamic program slicing technique based on redex trails. This technique is introduced and discussed in Chapter 5. A redex trail is a data structure which represents a computation. This technique is based on the

identification of control dependences in the redex trail in order to extract slices from programs. The technique is defined for source Curry programs, but the redex trail records the computation of equivalent FlatCurry programs (an internal representation of Curry programs used by the compiler PAKCS. See Section 2.4 for details); therefore, the technique needs a kind of translation between both languages. A formalization of classical program slicing concepts is introduced in order to adapt these concepts to the redex trails formalism. Moreover, it is shown that the same technique can be adapted to perform program specialization. Finally, it is also described the implementation of this technique in Curry.

The fourth part describes algorithmic debugging techniques, and it includes two chapters. In Chapter 6, program slicing is combined with algorithmic debugging in order to produce a more accurate debugging technique. The central idea is to allow the user to provide the algorithmic debugger with more information. To do so, the technique is described for higher-order term rewriting systems. In essence, it consists on the use of program slicing to prune the execution trees used in algorithmic debugging with the information provided by the user.

In Chapter 7, firstly, a classification of all program slicing techniques studied in this chapter is introduced. This classification is then used in order to predict new slicing techniques and to identify relationships between the techniques. Afterwards, three new algorithmic debugging strategies are introduced. Then, they are compared with the rest of strategies revised in this chapter and, as a result, a classification of algorithmic debugging strategies is obtained.

Finally, the conclusions together with the future work are discussed in Chapter 8.

Part I

Foundations

Chapter 2

Preliminaries

We recall in this chapter some basic notions of term rewriting, functional programming, and functional logic programming that will be used in the rest of the thesis. Firstly, we introduce some notations and concepts and provide a precise terminology. Most of the definitions and notations in this chapter are taken from [Dershowitz and Jouannaud, 1990; Klop, 1992; Antoy, 1992; Hanus, 1994; Baader and Nipkow, 1998; Alpuente *et al.*, 1999; Antoy *et al.*, 2000; Albert, 2001]

2.1 Signature and Terms

Throughout the rest of the thesis, we consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. We write $c/n \in \mathcal{C}$ and $f/n \in \mathcal{F}$ for n-ary constructor and operation symbols, respectively. There is at least one sort *Bool* containing the 0-ary Boolean constructors *true* and $false$. The set of *terms* and *constructor terms* (possibly with *variables* from $\mathcal{X}$; e.g., $x, y, \ldots$) are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. The set of variables occurring in a term t is denoted by $Var(t)$. A term t is *ground* if $Var(t) = \varnothing$. A term is *linear* if it does not contain multiple occurrences of any variable. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$.

A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A term is *operation-rooted* (*constructor-rooted*) if it has an operation (constructor) symbol at the root. $root(t)$ denotes the symbol at the root of the term t.

2.2 Substitutions

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. The set

$\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is called the *domain* of σ. The codomain of a substitution σ is the set $\mathcal{R}an(\sigma) = \{\sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. A substitution σ is *(ground) constructor*, if $\sigma(x)$ is (ground) constructor for all $x \in \mathcal{D}om(\sigma)$. The identity substitution is denoted by *id*. Substitutions are extended to morphisms on terms by $\sigma(f(\overline{t_n})) = f(\overline{\sigma(t_n)})$ for every term $f(\overline{t_n})$. Given a substitution θ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta_{|V}$ the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta_{|V} = \sigma_{|V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$.

A term t' is an *instance* of t if there is a substitution σ with $t' = \sigma(t)$. This implies a *subsumption ordering* on terms which is defined by $t \leq t'$ iff t' is an instance of t. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ is the *most general* unifier (*mgu*) if $\sigma \leq \sigma'\ [\mathcal{X}]$ for each other unifier σ'. A *mgu* is unique up to variable renaming.

2.3 Term Rewriting Systems

A set of rewrite rules $l \to r$ such that $l \notin \mathcal{X}$, and $Var(r) \subseteq Var(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* and the *right-hand side* of the rule, respectively. A TRS $\mathcal{R}$ is *left-linear* if l is linear for all $l \to r \in \mathcal{R}$. A TRS is constructor-based (CB) if each left-hand side l is a pattern. Two (possibly renamed) rules $l \to r$ and $l' \to r'$ *overlap*, if there is a non-variable position p in l and a most general unifier σ such that $\sigma(l|_p) = \sigma(l')$. *Extra variables* are those variables in a rule which do not occur in the left-hand side. The pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r)\rangle$ is called a *critical pair* (and is also called an *overlay* if $p = \Lambda$). A critical pair $\langle t, s\rangle$ is trivial if $t = s$ (up to variable renaming). A left-linear TRS without critical pairs is called *orthogonal*. It is called *almost orthogonal* if its critical pairs are trivial overlays. If it only has trivial critical pairs it is called *weakly orthogonal*. Note that, in CB-TRSs, almost orthogonality and weak orthogonality coincide.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = l \to r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$ (p and R will often be omitted in the notation of a computation step). The instantiated left-hand side $\sigma(l)$ is called a *redex* (reducible expression).

A term t is root-stable (often called a *head-normal form*) if it cannot be rewritten to a redex. A *constructor root-stable* term is either a variable or a *constructor-rooted* term, that is, a term rooted by a constructor symbol. By abuse, when it is clear from the context, we use the notion "head-normal form" to refer to constructor root-stable terms. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$. Given a binary relation $\to$, $\to^+$ denotes the transitive closure of $\to$ and $\to^*$ denotes the reflexive and transitive closure of $\to$.

Example 4 *Consider the following TRS that defines the addition on natural numbers represented by terms built from* `Zero` *and* `Succ`*:*[1]

$$\begin{array}{rcll} \texttt{Zero} + \texttt{y} & = & \texttt{y} & (\mathtt{R}_1) \\ (\texttt{Succ x}) + \texttt{y} & = & \texttt{Succ}\ (\texttt{x} + \texttt{y}) & (\mathtt{R}_2) \end{array}$$

Given the term $(\texttt{Succ Zero}) + (\texttt{Succ Zero})$*, we have the following sequence of rewrite steps:*

$$\begin{array}{lll} (\texttt{Succ Zero}) + (\texttt{Succ Zero}) & \to_{\Lambda,\mathtt{R}_2} & \texttt{Succ}\ (\texttt{Zero} + (\texttt{Succ Zero})) \\ & \to_{1,\mathtt{R}_1} & \texttt{Succ}\ (\texttt{Succ Zero}) \end{array}$$

2.3.1 Narrowing

Functional *logic* programs mainly differ from purely functional programs in that function calls may contain *free* variables. In order to evaluate terms containing free variables, *narrowing* non-deterministically instantiates these variables so that a rewrite step is possible. Formally,

Definition 1 (narrowing step) $t \leadsto_{(p,R,\sigma)} t'$ *is a* narrowing step *if* p *is a non-variable position of* t *and* $\sigma(t) \to_{p,R} t'$*.*

We often write $t \leadsto_\sigma t'$ when the position and the rule are clear from the context. We denote by $t_0 \leadsto_\sigma^n t_n$ a sequence of n narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$), usually restricted to the variables of t_0. Due to the presence of free variables, a term may be reduced to different values after instantiating these variables to different terms. Given a narrowing derivation $t_0 \leadsto_\sigma^* t_n$, we say that t_n is a computed *value* and σ is a computed *answer* for t_0.

Example 5 *Consider again the definition of function "*`+`*" in Example 4. Given the term* $\texttt{x} + (\texttt{Succ Zero})$*, narrowing non-deterministically performs the following derivations:*

$$\begin{array}{lll} \texttt{x} + (\texttt{Succ Zero}) & \leadsto_{\Lambda,\mathtt{R}_1,\{\mathtt{x}\mapsto\mathtt{Zero}\}} & \texttt{Succ Zero} \\ \texttt{x} + (\texttt{Succ Zero}) & \leadsto_{\Lambda,\mathtt{R}_2,\{\mathtt{x}\mapsto\mathtt{Succ}\ \mathtt{y}_1\}} & \texttt{Succ}\ (\texttt{y}_1 + (\texttt{Succ Zero})) \\ & \leadsto_{1,\mathtt{R}_1,\{\mathtt{y}_1\mapsto\mathtt{Zero}\}} & \texttt{Succ}\ (\texttt{Succ Zero}) \\ \texttt{x} + (\texttt{Succ Zero}) & \leadsto_{\Lambda,\mathtt{R}_2,\{\mathtt{x}\mapsto\mathtt{Succ}\ \mathtt{y}_1\}} & \texttt{Succ}\ (\texttt{y}_1 + (\texttt{Succ Zero})) \\ & \leadsto_{1,\mathtt{R}_2,\{\mathtt{y}_1\mapsto\mathtt{Succ}\ \mathtt{y}_2\}} & \texttt{Succ}\ (\texttt{Succ}\ (\texttt{y}_2 + (\texttt{Succ Zero}))) \\ & \leadsto_{1.1,\mathtt{R}_1,\{\mathtt{y}_2\mapsto\mathtt{Zero}\}} & \texttt{Succ}\ (\texttt{Succ}\ (\texttt{Succ Zero})) \\ \ldots & & \end{array}$$

Therefore, $\texttt{x} + (\texttt{Succ Zero})$ *non-deterministically computes the values*

[1] In the examples, we write constructor symbols starting with upper case (except for the list constructors, "`[]`" and "`:`", which are a shorthand for `Nil` and `Cons`, respectively).

- `Succ(Zero)` *with answer* $\{\mathtt{x} \mapsto \mathtt{Zero}\}$,
- `Succ(Succ(Zero))` *with answer* $\{\mathtt{x} \mapsto \mathtt{Succ\ Zero}\}$,
- `Succ(Succ(Succ(Zero)))` *with answer* $\{\mathtt{x} \mapsto \mathtt{Succ\ (Succ\ (Succ\ Zero))}\}$, *etc.*

As in logic programming, narrowing derivations can be represented by a (possibly infinite) finitely branching *tree*. Formally, given a program $\mathcal{R}$ and an operation-rooted term t, a *narrowing tree* for t in $\mathcal{R}$ is a tree satisfying the following conditions:

1. each node of the tree is a term,
2. the root node is t,
3. if s is a node of the tree then, for each narrowing step $s \leadsto_{p,R,\sigma} s'$, the node has a child s' and the corresponding arc is labeled with (p, R, σ), and
4. nodes which are constructor terms have no children.

In order to avoid unnecessary computations and to deal with infinite data structures, demand-driven generation of the search space has been advocated by a number of *lazy* narrowing strategies Giovannetti *et al.* [1991]; Loogen *et al.* [1993]; Moreno-Navarro and Rodríguez-Artalejo [1992]. Due to its optimality properties w.r.t. the length of derivations and the number of computed solutions, *needed narrowing* Antoy *et al.* [2000] is currently the best lazy narrowing strategy.

2.3.2 Needed Narrowing

Needed narrowing [Antoy *et al.*, 2000] is defined on *inductively sequential* TRSs [Antoy, 1992], a subclass of left-linear constructor-based TRSs. Essentially, a TRS is *inductively sequential* when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction. Inductive sequentiality is not a limiting condition for programming. In fact, the first-order components of many functional (logic) programs written in, e.g., Haskell, ML or Curry, are inductively sequential.

These systems have the property that the rules defining any operation can be organized in a hierarchical structure called definitional tree [Antoy, 1992], which is used to implement needed narrowing.

Definitional Trees

A definitional tree can be seen as a partially ordered set of patterns with some additional constraints. The symbols *branch* and $leaf$, used in the next definition, are uninterpreted functions used to classify the nodes of the tree.

Definition 2 (partial definitional tree [Antoy, 1992]) *$\mathcal{T}$ is a partial definitional tree, or pdt, with pattern π iff one of the following cases hold:*

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ where π is a pattern, o is the occurrence of a variable of π, the sort of $\pi|_o$ has constructors $c_1, \ldots, c_k$ for some $k > 0$, and for all i in $\{1, \ldots, k\}$, $\mathcal{T}_i$ is a pdt with pattern $\pi[c_i(x_1, \ldots, x_n)]_o$, where n is the arity of c_i and $x_1, \ldots, x_n$ are new distinct variables.

$\mathcal{T} = leaf(\pi)$ where π is a pattern.

We denote by $\mathcal{P}(\Sigma)$ the set of *pdt*s over the signature Σ. Let $\mathcal{R}$ be a rewrite system. $\mathcal{T}$ is a *definitional tree* of an operation f iff $\mathcal{T}$ is a *pdt* whose pattern argument is $f\ x_1 \ldots x_n$, where n is the arity of f and $x_1, \ldots, x_n$ are new distinct variables, and for every rule $l \to r$ of $\mathcal{R}$ with $l = f\ x_1 \ldots x_n$, there exists a leaf $leaf(\pi)$ of $\mathcal{T}$ such that l is a *variant* (i.e., equal modulo variable renaming) of π, and we say that the node $leaf(\pi)$ represents the rule $l \to r$. A definitional tree $\mathcal{T}$ of an operation f is called *minimal* iff below any *branch* node of $\mathcal{T}$ there is a $leaf$ representing a rule defining f.

Definition 3 (inductively sequential operation) *An operation f of a rewrite system $\mathcal{R}$ is called* inductively sequential *iff there exists a definitional tree $\mathcal{T}$ of f such that each $leaf$ node of $\mathcal{T}$ represents at most one rule of $\mathcal{R}$ and all rules are represented.*

Definition 4 (inductively sequential rewrite system) *A rewrite system $\mathcal{R}$ is called* inductively sequential *if all its defined functions are inductively sequential.*

It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules.

Let us note that, even if we consider a first-order language, we use a curried notation in the examples (as is usual in functional languages). Likewise, we use lower case letter for variables and upper case letter for constructor symbols, following the Curry syntax. Analogously, in our examples, rules have the form $l = r$ instead of $l \to r$, as in our formal notation.

Example 6 Consider the rules defining the "$\leqslant$" function:

$$\begin{array}{lcl} \texttt{0} \leqslant \texttt{y} & = & \texttt{True} \\ (\texttt{Succ x}') \leqslant \texttt{0} & = & \texttt{False} \\ (\texttt{Succ x}') \leqslant (\texttt{Succ y}') & = & \texttt{x}' \leqslant \texttt{y}' \end{array}$$

The definitional tree for the function "$\leqslant$" is illustrated in Figure 2.1. In the picture, each branch node contains the pattern of the corresponding node of the definitional

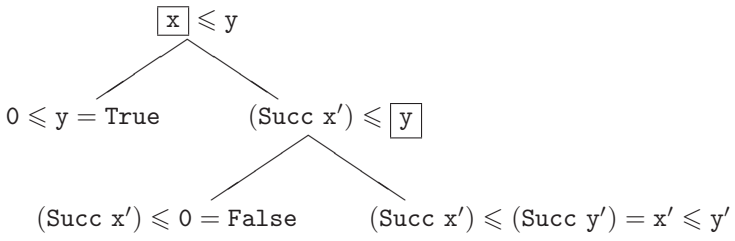


Figure 2.1: Definitional tree for the function "⩽"

tree. Every leaf node represents a rule. The inductive argument of a branch node is pointed by surrounding the corresponding subterm within a box.

The needed narrowing strategy performs steps which are *needed* for computing solutions:

Definition 5 (needed narrowing step/derivation [Antoy *et al.*, 2000])

1. *A narrowing step $t \leadsto_{p,R,\sigma} t'$ is called needed (or outermost-needed) iff, for every $\eta \geq \sigma$, p is the position of a needed (or outermost-needed) redex of $\eta(t)$, respectively.*

2. *A narrowing derivation is called needed (or outermost-needed) iff every step of the derivation is needed (or outermost-needed, respectively).*

In comparison to needed reduction, the definition of needed narrowing adds a new level of difficulty for computing the needed steps. In addition to consider the normal forms, one must take into account any possible instantiation of the input term. In inductively sequential systems, the problem has an efficient solution which is achieved by giving up the requirement that the unifier of a narrowing step be most general.

This strategy is basically equivalent to *lazy narrowing* [Moreno-Navarro and Rodríguez-Artalejo, 1992] where narrowing steps are applied to the outermost function, if possible, and inner functions are only narrowed if their evaluation is *demanded* by a constructor symbol in the left-hand side of some rule (i.e., a typical outermost strategy).

For instance, consider again Example 6. In a term like $t_1 \leqslant t_2$, it is always necessary to evaluate t_1 to some *head normal form* (i.e., a variable or a constructor-rooted term) since all three rules defining "⩽" have a non-variable first argument. On the other hand, the evaluation of t_2 is only *needed* if t_1 is of the form `Succ` t. Thus, if t_1 is a free variable, needed narrowing instantiates it to a constructor, here `Zero` or `Succ x`. Depending on this instantiation, either the first rule is applied or the second argument t_2 is evaluated.

2.4 Declarative Multi-Paradigm Languages

Functional logic languages have evolved to so called declarative *multi-paradigm* languages like, e.g., Curry [Hanus, 2006a], Toy [López-Fraguas and Sánchez-Hernández, 1999] and Escher [Lloyd, 1994]. In order to make things concrete, we consider in this thesis the language Curry. Curry is an international initiative intended to provide a common platform for the research, teaching and application of integrated functional logic languages. It is a programming language which combines the most important operational principles developed from the most relevant declarative programming paradigms, namely functional programming and logic programming.

Curry combines features from functional programming (nested expressions, higher-order functions, lazy evaluation), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). This combination allows Curry to provide additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions).

Curry follows a Haskell-like syntax, i.e., variables and function names start with lowercase letters and data constructors start with an uppercase letter. The application of function f to an argument e is denoted by juxtaposition, i.e., $(f\ e)$.

The basic operational semantics of Curry is based on a combination of needed narrowing and residuation [Hanus, 1997]. The *residuation* principle is based on the idea of delaying function calls until they are ready for a deterministic evaluation. Residuation preserves the deterministic nature of functions and naturally supports concurrent computations. The precise mechanism—narrowing or residuation—for each function is specified by *evaluation annotations*. The annotation of a function as *rigid* forces the delayed evaluation by rewriting, while functions annotated as *flexible* can be evaluated in a non-deterministic manner by narrowing.

In actual implementations, e.g., the PAKCS environment [Hanus *et al.*, 2006] for Curry, programs may also include a number of additional features: calls to external (built-in) functions, concurrent constraints, higher-order functions, overlapping left-hand sides, guarded expressions, etc. In order to ease the compilation of programs as well as to provide a common interface for connecting different tools working on source programs, a *flat representation* for programs has been introduced. This representation is based on the formulation of Hanus and Prehofer [1999] to express pattern-matching by case expressions. The complete flat representation is called FlatCurry [Hanus *et al.*, 2006] and is used as an intermediate language during the compilation of source programs.

Flat Programs

In FlatCurry, all functions are defined at the top level (i.e., local function declarations in source programs are globalized by lambda lifting) and the pattern-matching strategy is made explicit by the use of case expressions. Thus, a FlatCurry program basically consists of a list of data type declarations (defining the data constructors used in the program) and a list of function definitions. To support separate compilation, FlatCurry contains also information about modules (i.e., each module is represented as a FlatCurry program). To support interactive tools (e.g., debuggers, program verifiers, interactive back ends that allow the user to evaluate arbitrary expressions), the FlatCurry representation contains also some information about the syntactic representation of source programs (list of infix operator declarations and a translation table to map source program names to internal names and vice versa). Altogether, a FlatCurry program is a tuple:

$$(module, imports, types, functions, operators, translation)$$

with the following components [Hanus, 2006b]:

module: The name of the module represented by this structure.

imports: The names of the modules imported by this program.

types: The list of data type declarations defined in this module. Each data type declaration consists of a list of type variables as parameters (for polymorphic data structures) and a list of constructor declarations where each constructor declaration contains the name and arity of the data constructor and a list of type expressions (the argument types of this data constructor).

functions: The functions defined in this module. A function declaration consists of the name and arity of the function together with a type expression specifying the function's type and a single rule (where the right-hand side can contain case expressions and disjunctions) specifying the function's meaning. External functions (i.e., primitive functions not defined in this or another module) contain instead of the rule the external name of this function.

operators: The infix operators declared in this module. An operator declaration contains the fixity (or associativity) and the precedence for each operator. Although an operator declaration contains no semantic information, it might be used by interactive tools to print or read expressions.

translation: The translation table describes the mapping of identifiers (type constructors, data constructors, defined functions) used in the source program into their corresponding internal names used in the intermediate FlatCurry program

P	::=	$D_1 \ldots D_m$	(program)	t	::=	x	(variable)
D	::=	$f\ \overline{x_n} = e$	(rule)		|	$c\ \overline{t_n}$	(constructor call)
e	::=	t	(term)		|	$f\ \overline{t_n}$	(function call)
	|	$case\ x\ of\ \{\overline{p_m \to e_m}\}$	(rigid case)	p	::=	$c\ \overline{x_n}$	(flat pattern)
	|	$fcase\ x\ of\ \{\overline{p_m \to e_m}\}$	(flexible case)				

Figure 2.2: Syntax of flat programs

(since it is often necessary to rename identifiers to avoid name clashes, e.g., the internal names are prefixed by the module name). Only the externally visible names (i.e., names exported by this module) are listed here. This information might be used by back ends for representing intermediate (e.g., during debugging) or final results or to allow the user to type in and evaluate arbitrary expressions.

In order to simplify the presentation, in some chapters we will only consider the *core* of the flat representation. Extending the developments to the remaining features is not difficult and, indeed, the implementations reported for the debugging techniques in this thesis cover many of these features. The syntax of flat programs is summarized in Figure 2.2. Extra variables are intended to be instantiated by flexible case expressions. We assume that each extra variable x is explicitly introduced in flat programs by a direct circular let binding of the form "*let* $x = x$ *in* e". We also call such variables which are bound to themselves *logical variables*.

We consider the following domains:

$$e_1, e_2, \ldots \in \mathcal{E}\ (\mathsf{expressions}) \qquad t_1, t_2, \ldots \in \mathcal{T}\ (\mathsf{terms}) \qquad v_1, v_2, \ldots \in \mathcal{V}\ (\mathsf{values})$$

The only difference between *terms* and *expressions* is that the latter may contain case expressions. *Values* are terms in head normal form, i.e., variables or constructor-rooted terms. A program P consists of a sequence of function definitions; each function is defined by a single rule whose left-hand side contains only different variables as parameters. The right-hand side is an expression e composed by variables, constructors, function calls, and case expressions for pattern-matching. In general, a case expression has the form:

$(f)case\ x\ of\ \{c_1\ \overline{x_{n_1}} \to e_1; \ldots; c_k\ \overline{x_{n_k}} \to e_k\}$

where $(f)case$ stands for either *fcase* or *case*, x is a variable, $c_1, \ldots, c_k$ are different constructors, and $e_1, \ldots, e_k$ are expressions (possibly containing nested $(f)case$'s). The *pattern variables* $\overline{x_{n_i}}$ are locally introduced and bind the corresponding vari-

ables of the subexpression e_i. The difference between *case* and *fcase* shows up when the argument x evaluates to a free variable (within a particular computation): *case* suspends—which corresponds to *residuation*, i.e., pure functional reduction—whereas *fcase* non-deterministically binds this variable to the pattern in a branch of the case expression—which corresponds to narrowing [Antoy *et al.*, 2000]. Note that our functional logic language mainly differs from typical (lazy) functional languages in the presence of flexible case expressions.

Example 7 *Consider again the rules defining functions "+" (Example 4) and "$\leqslant$" (Example 6). These functions can be defined in the flat representation as follows*[2]*:*

```
x + y = fcase x of { Zero   → y;
                     Succ n → Succ (n + y) }

x ⩽ y = fcase x of { Zero   → True;
                     Succ n → fcase y of { Zero   → False;
                                           Succ m → n ⩽ m } }
```

An automatic transformation from source (inductively sequential) programs to flat programs has been introduced by Hanus and Prehofer [1999]. Translated programs always fulfill the following restrictions: case expressions in the right-hand sides of program rules appear always in the outermost positions (i.e., there is no case expression inside a function or constructor call) and all case arguments are variables, thus the syntax of Figure 2.2 is general enough for our purposes. We shall assume these restrictions on flat programs in the following.

The operational semantics of flat programs is shown in Figure 2.3. It is based on the LNT—for Lazy Narrowing with definitional Trees—calculus of Hanus and Prehofer [1999]. The one-step transition relation $\Longrightarrow_\sigma$ is labeled with the substitution σ computed in the step. Let us briefly describe the LNT rules:

The select rule selects the appropriate branch of a case expression and continues with the evaluation of this branch. This rule implements *pattern matching*.

The guess rule applies when the argument of a *flexible* case expression is a variable. Then, this rule non-deterministically binds this variable to a pattern in a branch of the case expression. The step is labeled with the computed binding. Observe that there is no rule to evaluate a rigid case expression with a variable argument. This situation produces a *suspension* of the evaluation.

The case eval rule can be applied when the argument of the case construct is not in head normal form (i.e., it is either a function call or another case construct). Then, it tries to evaluate this expression recursively.

[2] Although we consider a first-order representation—the flat language—we use a curried notation in concrete examples (as in Curry).

$$\text{(select)} \quad (f)case\ (c\ \overline{t_n})\ of\ \{\overline{p_m \to e_m}\} \Longrightarrow_{id} \sigma(e_i) \quad \text{if } p_i = c\ \overline{x_n},\ c \in \mathcal{C}, \text{ and } \sigma = \{\overline{x_n \mapsto t_n}\}$$

$$\text{(guess)} \quad f case\ x\ of\ \{\overline{p_m \to e_m}\} \Longrightarrow_{\sigma} \sigma(e_i) \quad \text{if } \sigma = \{x \mapsto p_i\} \text{ and } i \in \{1, \ldots, m\}$$

$$\text{(case eval)} \quad (f)case\ e\ of\ \{\overline{p_m \to e_m}\} \Longrightarrow_{\sigma} \sigma((f)case\ e'\ of\ \{\overline{p_m \to e_m}\}) \quad \text{if } e \text{ is not in head normal form and } e \Longrightarrow_{\sigma} e'$$

$$\text{(fun)} \quad f\ \overline{t_n} \Longrightarrow_{id} \sigma(e) \quad \text{if } f\ \overline{x_n} = e \in P \text{ and } \sigma = \{\overline{x_n \mapsto t_n}\}$$

Figure 2.3: Standard operational semantics (LNT calculus)

Finally, the fun rule performs the unfolding of a function call. As in proof procedures for logic programming, we assume that we take a program rule with fresh variables in each such evaluation step.

Note that there is no rule to evaluate terms in head normal form; in this case, the computation stops *successfully*. An LNT *derivation* is denoted by $e_0 \Longrightarrow_{\sigma}^{*} e_n$, which is a shorthand for the sequence $e_0 \Longrightarrow_{\sigma_1} \ldots \Longrightarrow_{\sigma_n} e_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). An LNT derivation $e \Longrightarrow_{\sigma}^{*} e'$ is *successful* when e' is in head normal form. Then, we say that e *evaluates* to e' with *computed answer* σ.

Example 8 *Consider the function "$\leqslant$" of Example 7 together with the initial call "*`(Succ x) ⩽ y`*". The LNT calculus computes, among others, the following successful derivation:*

```
(Succ x) ⩽ y
 ⟹id   fcase (Succ x) of                                          (fun)
          {Z       → True;
          (Succ n) → fcase y of {Z → False; (Succ m) → n ⩽ m}}
 ⟹id   fcase y of {Z → False; (Succ m) → x ⩽ m}                    (select)
 ⟹{y↦Z} False                                                      (guess)
```

Therefore, `(Succ x) ⩽ y` *evaluates to* `False` *with computed answer* $\{\mathtt{y} \mapsto \mathtt{Z}\}$.

In the remainder of this thesis, unless the contrary is stated, we will assume that computations always start from the distinguished function `main` which has no arguments.

Normalized Flat Programs

In some chapters we will assume that flat programs are *normalized*, i.e., *let* constructs are used to ensure that the arguments of functions and constructors are always vari-

P	::=	$D_1 \ldots D_m$	(program)	t	::=	x	(variable)
D	::=	$f\ \overline{x_n} = e$	(rule)		|	$c\ \overline{x_n}$	(constructor call)
e	::=	t	(term)		|	$f\ \overline{x_n}$	(function call)
	|	$case\ x\ of\ \{\overline{p_m \to e_m}\}$	(rigid case)	p	::=	$c\ \overline{x_n}$	(flat pattern)
	|	$fcase\ x\ of\ \{\overline{p_m \to e_m}\}$	(flexible case)				
	|	$e_1\ or\ e_2$	(disjunction)				
	|	$let\ x = e_1\ in\ e_2$	(let binding)				

Figure 2.4: Syntax of normalized flat programs

ables (not necessarily pairwise different). As in [Launchbury, 1993], this is essential to express *sharing* without the use of complex graph structures.

Definition 6 (normalization [Albert *et al.*, 2005]) *The normalization of an expression e flattens all the arguments of function (or constructor) calls by means of the mapping e^* which is inductively defined as follows:*

$$
\begin{array}{rcl}
x^* & = & x \\
\varphi(\overline{x_n})^* & = & \varphi(\overline{x_n}) \\
\varphi(\overline{x_{i-1}}, \overline{e_m})^* & = & let\ x_i = e_1^*\ in \\
 & & \quad \varphi(\overline{x_i}, e_2, \ldots, e_m)^* \\
 & & where\ e_1 \notin Var,\ x_i\ is\ fresh \\
(let\ \overline{x_k = e_k}\ in\ e)^* & = & let\ \overline{x_k = {e_k}^*}\ in\ e^* \\
(e_1\ or\ e_2)^* & = & {e_1}^*\ or\ {e_2}^* \\
((f)case\ x\ of\ \{\overline{p_k \to e_k}\})^* & = & (f)case\ x\ of\ \{\overline{p_k \mapsto {e_k}^*}\} \\
((f)case\ e\ of\ \{\overline{p_k \to e_k}\})^* & = & let\ x = e^*\ in \\
 & & \quad (f)case\ x\ of\ \{\overline{p_k \mapsto {e_k}^*}\} \\
 & & where\ e \notin Var,\ x\ is\ fresh
\end{array}
$$

Here, φ denotes a constructor or function symbol. The extension of normalization to flat programs is straightforward, i.e., each rule $f\ \overline{x_n} = e$ is transformed into $f\ \overline{x_n} = e^*$.

For simplicity, the normalization mapping above introduces one new let construct for each non-variable argument, e.g., $f\ e$ is transformed into "$let\ x = e\ in\ f\ x$". Trivially, this could be modified in order to produce one single let with the bindings for all non-variable arguments of a function (or constructor) call, which we assume for the subsequent examples.

The syntax of normalized flat programs is shown in Figure 2.4. A program P consists of a sequence of function definitions D such that the left-hand side has pairwise different variable arguments. The right-hand side is an expression $e \in \mathcal{E}$ composed

(VarCons) $\Gamma[x \mapsto c(\overline{x_n})] : x \;\Downarrow\; \Gamma[x \mapsto c(\overline{x_n})] : c(\overline{x_n})$

(VarExp) $$\frac{\Gamma : e \;\Downarrow\; \Delta : v}{\Gamma[x \mapsto e] : x \;\Downarrow\; \Delta[x \mapsto v] : v}$$ (where e is not a value)

(Val) $\Gamma : v \;\Downarrow\; \Gamma : v$ (where v is a value)

(Fun) $$\frac{\Gamma : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : f(\overline{x_n}) \;\Downarrow\; \Delta : v}$$ (where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$)

(Let) $$\frac{\Gamma[y \mapsto \rho(e')] : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : let\ x = e'\ in\ e \;\Downarrow\; \Delta : v}$$ (where $\rho = \{x \mapsto y\}$ and y is fresh)

(Or) $$\frac{\Gamma : e_i \;\Downarrow\; \Delta : v}{\Gamma : e_1\ or\ e_2 \;\Downarrow\; \Delta : v}$$ (where $i \in \{1, 2\}$)

(Select) $$\frac{\Gamma : x \;\Downarrow\; \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : (f)case\ x\ of\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$$ (where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$)

(Guess) $$\frac{\Gamma : x \;\Downarrow\; \Delta : y \qquad \Delta[y \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : fcase\ x\ of\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$$ (where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$ and $\overline{y_n}$ are fresh variables)

Figure 2.5: Rules of the natural semantics

by variables, data constructors, function calls, let bindings where the local variable x is only visible in e_1 and e_2, disjunctions (e.g., to represent set-valued functions), and case expressions for pattern-matching.

The Natural Semantics

In this section, we describe a natural (big-step) semantics for functional logic languages (defined in [Albert *et al.*, 2005]). Figure 2.5 shows the semantics. A *heap*, denoted by Γ, Δ, or Θ, is a partial mapping from variables to expressions (the *empty heap* is denoted by $[\,]$). The value associated to variable x in heap Γ is denoted by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap with $\Gamma[x] = e$. A logical variable x is represented by a circular binding of the form $\Gamma[x] = x$. A *value* v is a constructor-rooted term $c\ \overline{e_n}$ (i.e., a term whose outermost function symbol is a constructor symbol) or a logical variable (w.r.t. the associated heap). We use judgements of the form "$\Gamma : e \;\Downarrow\; \Delta : v$" which are interpreted as "in the context of heap Γ, the expression e evaluates to value v, producing a new heap Δ".

In order to evaluate a variable which is bound to a constructor-rooted term in the

heap, rule VarCons reduces the variable to this term.

Rule VarExp achieves the effect of *sharing*. If the variable to be evaluated is bound to some expression in the heap, then the expression is evaluated and the heap is updated with the computed value; finally, we return this value as the result.

For the evaluation of a value, rule Val returns it without modifying the heap.

Rule Fun corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule (we assume that the considered program P is a global parameter of the calculus).

Rule Let adds its associated binding[3] to the heap and proceeds with the evaluation of its main argument. Note that we give the introduced variable a fresh name in order to avoid variable name clashes.

Rule Or non-deterministically reduces an *or* expression to either the first or the second argument.

Rule Select corresponds to the evaluation of a case expression whose argument reduces to a constructor-rooted term. In this case, we select the appropriate branch and, then, proceed with the evaluation of the expression in this branch by applying the corresponding matching substitution.

Rule Guess applies when the argument of a flexible case expression reduces to a logical variable. It binds this variable to one of the patterns and proceeds by evaluating the corresponding branch. If there is more than one branch, one of them is chosen non-deterministically. Renaming the pattern variables is necessary to avoid name clashes. We also update the heap with the (renamed) logical variables of the pattern.

A proof of a judgement corresponds to a derivation sequence using the rules of Figure 2.5. Given a program P, the *initial configuration* has the form "$[\,] : main$". If the judgement

$$[\,] : main \;\Downarrow\; \Gamma : v$$

holds, we say that *main* evaluates to value v. The computed *answer* can be extracted from the final heap Γ by a simple process of *dereferencing*.

The Operational Semantics

We now present the (small-step) operational semantics of Albert *et al.* [2005]. Each configuration of the small-step semantics is a triple, $\langle \Gamma, e, S \rangle$, where Γ is a heap, e is an expression (often called the *control* of the small-step semantics), and S is a stack. A *stack* (a list of variable names and case alternatives where the empty stack is denoted by $[\,]$) is used to represent the current context.

[3] We consider that Let only has one binding for simplicity.

(VarCons)	$\langle \Gamma[x \mapsto t], x, S\rangle \Longrightarrow \langle \Gamma[x \mapsto t], t, S\rangle$	where t is constructor-rooted
(VarExp)	$\langle \Gamma[x \mapsto e], x, S\rangle \Longrightarrow \langle \Gamma[x \mapsto e], e, x : S\rangle$	where e is not constructor-rooted and $e \neq x$
(Val)	$\langle \Gamma[x \mapsto e], v, x : S\rangle \Longrightarrow \langle \Gamma[x \mapsto v], v, S\rangle$	where v is a value
(Fun)	$\langle \Gamma, f(\overline{x_n}), S\rangle \Longrightarrow \langle \Gamma, \rho(e), S\rangle$	where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$
(Let)	$\langle \Gamma, let\ x = e_1\ in\ e_2, S\rangle \Longrightarrow \langle \Gamma[y \mapsto \rho(e_1)], \rho(e_2), S\rangle$	where $\rho = \{x \mapsto y\}$ and y is a fresh variable
(Or)	$\langle \Gamma, e_1\ or\ e_2, S\rangle \Longrightarrow \langle \Gamma, e_i, S\rangle$	where $i \in \{1, 2\}$
(Case)	$\langle \Gamma, (f) case\ x\ of\ \{\overline{p_k \to e_k}\}, S\rangle \Longrightarrow \langle \Gamma, x, (f)\{\overline{p_k \to e_k}\} : S\rangle$	
(Select)	$\langle \Gamma, c(\overline{y_n}), (f)\{\overline{p_k \to e_k}\} : S\rangle \Longrightarrow \langle \Gamma, \rho(e_i), S\rangle$	where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, and $\rho = \{\overline{x_n \mapsto y_n}\}$
(Guess)	$\langle \Gamma[y \mapsto_{(g,w)} y], y, f\{\overline{p_k \to e_k}\} : S\rangle \Longrightarrow \langle \Gamma[y \mapsto \rho(p_i), \overline{y_n \mapsto y_n}], \rho(e_i), S\rangle$	where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh variables

Figure 2.6: Small-step operational semantics

The transition rules are shown in Figure 2.6. The meaning of the rules is the same as in the natural semantics. Once a value is eventually computed, the computation either terminates—when the stack is empty—or this value is used to update the current heap (rule val).

The stack is used during the evaluation of case expressions. Rule Case initiates the evaluation of a case expression by evaluating the case argument and pushing the alternatives $(f)\{\overline{p_k \to e_k}\}$ on top of the stack. If we reach a constructor-rooted term, then rule Select is applied to select the appropriate branch and continue with the evaluation of this branch. If we reach a logical variable and the case expression on the stack is flexible (i.e., of the form $f\{\overline{p_k \to e_k}\}$), then rule Guess is used to non-deterministically choose one alternative and continue with the evaluation of this branch; moreover, the heap is updated with the binding of the logical variable to the corresponding pattern.

Example 9 *Consider function* `leq` *of Example 11 (where the bug is now corrected), together with the initial call* (`leq Z` (`SZ`)). *The corresponding normalized flat program*

is the following:

```
main = let x3 = Z in
       let x1 = Z in
       let x2 = S x3 in leq x1 x2

leq x y = case x of
            { Z     → True;
              (S n) → case y of { Z     → False;
                                  (S m) → leq n m } }
```

The complete computation with the rules of Figure 2.6 is as follows:

$$
\begin{array}{rlll}
\langle [\,], \texttt{main}, [\,]\rangle & \Longrightarrow_{\mathsf{fun}} & \langle [\,], & \texttt{let x3} = \texttt{Z in} \ldots, \quad [\,]\rangle \\
& \Longrightarrow_{\mathsf{let}} & \langle [\texttt{x3} \mapsto \texttt{Z}], & \texttt{let x1} = \texttt{Z in} \ldots, \quad [\,]\rangle \\
& \Longrightarrow_{\mathsf{let}} & \langle [\texttt{x1} \mapsto \texttt{Z}, \ldots], & \texttt{let x2} = \texttt{S x3 in} \ldots, \quad [\,]\rangle \\
& \Longrightarrow_{\mathsf{let}} & \langle [\texttt{x2} \mapsto \texttt{S x3}, \ldots], & \texttt{leq x1 x2}, \quad [\,]\rangle \\
& \Longrightarrow_{\mathsf{fun}} & \langle [\ldots], & \texttt{case x1 of} \ldots, \quad [\,]\rangle \\
& \Longrightarrow_{\mathsf{case}} & \langle [\texttt{x1} \mapsto \texttt{Z}, \ldots], & \texttt{x1}, \quad [\{\ldots\}]\rangle \\
& \Longrightarrow_{\mathsf{varcons}} & \langle [\texttt{x1} \mapsto \texttt{Z}, \ldots], & \texttt{Z}, \quad [\{\ldots\}]\rangle \\
& \Longrightarrow_{\mathsf{select}} & \langle [\ldots], & \texttt{True}, \quad [\,]\rangle
\end{array}
$$

For clarity, each computation step is labeled with the applied rule. Therefore, `main` *computes the value* `True`*.*

Part II

Profiling

Chapter 3

Time Equations for Lazy Functional (Logic) Languages

There are very few approaches to measure the execution costs of *lazy* functional (logic) programs. The use of a lazy execution mechanism implies that the complexity of an evaluation depends on its *context*, i.e., the evaluation of the same expression may have different costs depending on the degree of evaluation required by the different contexts where it appears. In this chapter, we present a novel approach to complexity analysis of functional (logic) programs. We focus on the construction of equations which compute the time-complexity of expressions. In contrast to previous approaches, it is simple, precise—i.e., it computes exact costs rather than upper/lower bounds—, and fully automatic.

Part of the material of this chapter has been presented in the 2003 *Joint Appia Gulp Prode Conference on Declarative Programming* (AGP 2003), celebrated in Reggio Calabria (Italy) in 2003 [Albert *et al.*, 2003].

3.1 Introduction

There are very few approaches to formally reason about program execution costs in the field of lazy functional (logic) languages. Most of these approaches analyze the cost by first constructing (recursive) equations which describe the time-complexity of a given program and then producing a simpler cost function (ideally as a *closed form*) by some (semi-)automatic manipulation of these equations. In this chapter, we concentrate on the *first* part of this process: the (automatic) construction of equations which compute the time-complexity of a given lazy functional (logic) program.

Laziness introduces a considerable difficulty in the time-complexity analysis of

functional (logic) programs. In eager (call-by-value) languages, the cost of nested applications, e.g., $f(g(v))$, can be expressed as the sum of the costs of evaluating $g(v)$ and $f(v')$, where v' is the result returned by the call $g(v)$. In a lazy (call-by-name) language, $g(v)$ is only evaluated as much as needed by the outermost function f (in the extreme, it could be no evaluated at all). Therefore, one cannot determine (statically) the cost associated to the evaluation of the arguments of function calls. There exist several methods which simplify this problem by transforming the original program into an equivalent one under a call-by-value semantics and, then, by analyzing the transformed program (e.g., [Métayer, 1988]). The translation, however, makes the program significantly more complex and the number of evaluation steps is not usually preserved through the transformation. A different approach is taken by [Sands, 1990; Wadler, 1988], where the generated time-equations give upper (resp. lower) bounds by using a strictness (resp. neededness) analysis. In particular, Wadler [1988] may produce equations which are non-terminating even if the original program terminates; in contrast, Sands [1990] guarantees that the time-equations terminate at least as often as the original program. As an alternative approach, there are techniques (e.g., [Bjerner and Holmström, 1989]) which produce time equations which are *precise*, i.e., they give an *exact* time analysis. In this case, the drawback is that the time equations cannot be solved mechanically.

In this chapter, we introduce a novel program transformation which produces (automatically) time equations which are *precise*—i.e., give an exact time—and, moreover, they can be executed in the same language of the source program—i.e., we define a source-to-source transformation.

To be more precise, our aim is to transform a program so that it computes not only values but their associated time-complexities. More formally, given a lazy (call-by-name) semantics Sem and its cost-augmented version Sem_{cost}, we want to define a program transformation $trans(P) = P_{cost}$ such that the execution of program P with the cost-augmented semantics gives the same result as the execution of the transformed program P_{cost} with the standard semantics, i.e., $Sem_{cost}(P) = Sem(P_{cost})$. Let us note that the considered problem is decidable. First, it is not difficult to construct a cost-augmented semantics which precisely computes the cost of a program's execution (see next section). Then the Futamura projections [Futamura, 1999] ensure the existence of the desired transformed program P_{cost}: the *partial evaluation* [Jones *et al.*, 1993] of Sem_{cost} w.r.t. P gives a program P' such that P' produces the same outputs as $Sem_{cost}(P)$, i.e.,

$$Sem(P, input) = output$$
$$Sem_{cost}(P, input) = (output, cost)$$
$$mix(Sem_{cost}, P) = P' \quad \text{with} \quad P'(input) = (output, cost)$$

where mix denotes a partial evaluator. Therefore, P_{cost} (equivalently, P' in the above

equations) exists and can be effectively constructed. However, by using a partial evaluator for the considered language (e.g., [Albert *et al.*, 2002; Albert and Vidal, 2001]), P_{cost} will be probably a cost-augmented interpreter slightly extended to only consider the rules of a particular program P. Therefore, similarly to the cost-augmented interpreter, the program P_{cost} generated in this way will still use a sort of "ground representation" to evaluate expressions. This makes the complexity analysis of the program P_{cost} as difficult as the analysis of the original program (or even harder). Furthermore, its execution will be almost as slow as running the cost-augmented interpreter with the original program P (which makes it infeasible for profiling).

Our main concern in this chapter is to provide a more practical transformation. In particular, our new approach produces simple equations, it gives exact time-complexities, and it is fully automatic. For the sake of generality, our developments are formalized in the context of a lazy functional *logic* language and, in particular, we consider normalized flat programs in our formalizations. The use of *logical variables* may be useful to analyze the complexity of the time-equations (as we will discuss in Section 3.4.1). Nevertheless, our ideas can be directly applied to a pure (first-order) lazy functional language.

The chapter is organized as follows. Section 3.2 introduces the cost-augmented semantics of the considered lazy functional logic language. In Sect. 3.3, we present our program transformation and state its correctness. The usefulness of the transformation is illustrated in Sect. 3.4 and several applications are outlined. Finally, Section 3.5 reviews some related works.

3.2 Cost Augmented Semantics

In this section, we introduce a precise characterization of a cost semantics for our lazy functional logic language.

We base our developments on the big-step semantics of Section 2.4. The only difference is that our cost-augmented semantics does not model *sharing*, since then the generated time-equations would be much harder to analyze (see Section 3.4.2). In the following, we present an extension of the big-step semantics of Section 2.4 in order to count the number of *function unfoldings*. The step-counting (state transition) semantics for lazy functional logic programs is defined in Figure 3.1. We use judgments of the form "$\Gamma : e \Downarrow_k \Delta : v$" which are interpreted as "the expression e in the context of the heap Γ evaluates to the value v with the (possibly modified) heap Δ by unfolding k function calls". We briefly explain the rules of our semantics:

(VarCons) In order to evaluate a variable which is bound to a constructor-rooted term in the heap, we simply reduce the variable to this term. The heap remains unchanged and the associated cost is trivially zero.

(VarCons) $\Gamma[x \mapsto t] : x \ \Downarrow_0 \ \Gamma[x \mapsto t] : t$ where t is constructor-rooted

(VarExp) $$\frac{\Gamma[x \mapsto e] : e \ \Downarrow_k \ \Delta : v}{\Gamma[x \mapsto e] : x \ \Downarrow_k \ \Delta[x \mapsto e] : v}$$ where e is not constructor-rooted and $e \neq x$

(Val) $\Gamma : v \ \Downarrow_0 \ \Gamma : v$ where v is constructor-rooted or a variable with $\Gamma[v] = v$

(Fun) $$\frac{\Gamma : \rho(e) \ \Downarrow_k \ \Delta : v}{\Gamma : f(\overline{x_n}) \ \Downarrow_{k+1} \ \Delta : v}$$ where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$

(Let) $$\frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \ \Downarrow_k \ \Delta : v}{\Gamma : let \ \{\overline{x_k = e_k}\} \ in \ e \ \Downarrow_k \ \Delta : v}$$ where $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh variables

(Select) $$\frac{\Gamma : e \ \Downarrow_{k_1} \ \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \ \Downarrow_{k_2} \ \Theta : v}{\Gamma : (f)case \ e \ of \ \{\overline{p_k \to e_k}\} \ \Downarrow_{k_1+k_2} \ \Theta : v}$$ where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$

(Guess) $$\frac{\Gamma : e \ \Downarrow_{k_1} \ \Delta : x \qquad \Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \ \Downarrow_{k_2} \ \Theta : v}{\Gamma : fcase \ e \ of \ \{\overline{p_k \to e_k}\} \ \Downarrow_{k_1+k_2} \ \Theta : v}$$ where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh variables

Figure 3.1: Step-counting semantics for lazy functional logic programs

(VarExp) If the variable to be evaluated is bound to some expression in the heap, then this expression is evaluated and, then, this value is returned as the result. The cost is not affected by the application of this rule.

It is important to notice that this rule is different from the one presented in Section 2.4. In particular, in the original big-step semantics, it is defined as follows:

$$\frac{\Gamma[x \mapsto e] : e \ \Downarrow_k \ \Delta : v}{\Gamma[x \mapsto e] : x \ \Downarrow_k \ \Delta[x \mapsto v] : v}$$

which means that the heap is updated with the computed value v for e. This is essential to correctly model sharing (trivially, our modification does not affect to the computed results, only to the number of function unfoldings). The generation of time-equations that correctly model sharing is a difficult topic which is out of the scope of this thesis (see Section 3.4.2).

(Val) For the evaluation of a value, we return it without modifying the heap, with associated cost zero.

(Fun) This rule corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule. We assume that the considered program P is a global parameter of the calculus. Trivially, if the evaluation of the unfolded right-hand side requires k function unfoldings, the original function call will require $k + 1$ function calls.

(Let) In order to reduce a let construct, we add the bindings to the heap and proceed with the evaluation of the main argument of *let*. Note that we rename the variables introduced by the let construct with fresh names in order to avoid variable name clashes. The number of function unfoldings is not affected by the application of this rule.

(Select) This rule corresponds to the evaluation of a case expression whose argument reduces to a constructor-rooted term. In this case, we select the appropriate branch and, then, proceed with the evaluation of the expression in this branch by applying the corresponding matching substitution. The number of function unfoldings is obtained by adding the number of function unfoldings required to evaluate the case argument with those required to evaluate the expression in the selected branch.

(Guess) This rule considers the evaluation of a flexible case expression whose argument reduces to a logical variable. It non-deterministically binds this variable to one of the patterns and proceeds with the evaluation of the corresponding branch. Renaming of pattern variables is also necessary to avoid variable name clashes. Additionally, we update the heap with the (renamed) logical variables of the pattern. Similarly to the previous rule, the number of function unfoldings is the addition of the function unfoldings in the evaluation of the case argument and the (non-deterministically) selected branch.

In the following, we denote by a judgment of the form $\Gamma : e \Downarrow \Delta : v$ a derivation with the standard semantics (without cost), i.e., the semantics of Figure 3.1 by ignoring the cost annotations.

3.3 The Program Transformation

In this section, we present our program transformation to compute the time complexity of lazy functional logic programs. First, let us informally describe the approach of Wadler [1988] in order to motivate our work and its advantages (the approach of Sands [1990] is basically equivalent except for the use of a neededness analysis rather than a strictness analysis). In this method, for each function definition $f(x_1, \ldots, x_n) = e$, a new rule of the form

$$cf(x_1, \ldots, x_n, \alpha) \;=\; \alpha \;\hookrightarrow\; 1 + \mathcal{T}[\![e]\!]\alpha$$

is produced, where cf is a *time function* and α denotes an evaluation *context* which indicates whether the evaluation of a given expression is required. In this way, transformed functions are parameterized by a given context. The right-hand side is "guarded" by the context so that the corresponding cost is not added when its

evaluation is not required. A key element of this approach is the *propagation* of contexts through expressions. For instance, the definition of function $\mathcal{T}$ for function calls is as follows:

$$\mathcal{T}[\![f(e_1,\ldots,e_n)]\!]\alpha = \mathcal{T}[\![e_1]\!](f^{\sharp 1}\alpha) + \ldots + \mathcal{T}[\![e_n]\!](f^{\sharp n}\alpha) + cf(e_1,\ldots,e_n,\alpha)$$

The basic idea is that the cost of evaluating a function application can be obtained from the cost of evaluating the *strict* arguments of the function plus the cost of the outermost function call. This information is provided by a strictness analysis. Recall that a function is *strict* in some argument i iff $f(\ldots,\bot_i,\ldots)=\bot$, where $\bot$ denotes an undefined expression (i.e., the result of function f cannot be computed when the argument i cannot be evaluated to a value).

Functions $f^{\sharp i}$ are used to propagate strictness information through function arguments. To be more precise, they are useful to determine whether argument i of function f is strict in the context α. Let us illustrate this approach with an example. Consider the following function definitions:[1]

$$\begin{array}{lcl} \texttt{head xs} & = & \texttt{case xs of } \{(\texttt{y} : \texttt{ys}) \rightarrow \texttt{y}\} \\ \texttt{foo xs} & = & \texttt{let ys} = \texttt{head xs in head ys} \end{array}$$

According to Wadler [1988], these functions are transformed into

$$\begin{array}{lcl} \texttt{chead xs } \alpha & = & \alpha \hookrightarrow \texttt{1} \\ \texttt{cfoo xs } \alpha & = & \alpha \hookrightarrow \texttt{1} + \texttt{chead xs } (\texttt{head}^{\sharp 1}\alpha) \\ & & \quad + \texttt{let ys} = \texttt{head xs in chead } (\texttt{ys } \alpha) \end{array}$$

If the strictness analysis has a good precision, then whenever this function is called within a strict context, $(\texttt{head}^{\sharp 1}\alpha)$ should indicate that the head normal form of the argument of `head` is also needed.

Unlike the previous transformation, we do not use the information of a strictness (or neededness) analysis. A novel idea of our method is that we *delay* the computation of the cost of a given function up to the point where its evaluation is actually required. This contrasts to [Sands, 1990; Wadler, 1988] where the costs of the arguments of a function call are computed *a priori*, hence the need of the strictness information. The following definition formalizes our program transformation:

Definition 7 (time-equations) *Let P be a program. For each program rule of the form $f(x_1,\ldots,x_n)=e \in P$, we produce a transformed rule:*

$$\begin{array}{l} f_c(x_1,\ldots,x_n) = \mathit{let}\ y = 1 \\ \qquad\qquad\qquad \mathit{in}\ \mathcal{C}[\![e]\!] \pm y \end{array}$$

[1] Although we consider a first-order language, we use a curried notation in examples, as it is common practice in functional programming.

$$
\begin{array}{rcl}
\mathcal{C}[\![x]\!] & = & x \\
\mathcal{C}[\![c(x_1,\ldots,x_n)]\!] & = & \mathit{let}\ y = c(x_1,\ldots,x_n) \\
 & & \mathit{in}\ \mathit{let}\ z = 0 \\
 & & \quad \mathit{in}\ (y,z) \\
\mathcal{C}[\![f(x_1,\ldots,x_n)]\!] & = & f_c(x_1,\ldots,x_n) \\
\mathcal{C}[\![(f)\mathit{case}\ x\ \mathit{of}\ \{\overline{p_n \to e_n}\}]\!] & = & (f)\mathit{case}\ x\ \mathit{of}\ \{\overline{(p_n,k_n) \to \mathcal{C}[\![e_n]\!] \mathbin{\underline{+}} k_n}\} \\
\mathcal{C}[\![\mathit{let}\ \overline{y_n = e_n}\ \mathit{in}\ e]\!] & = & \mathit{let}\ \overline{y_n = \mathcal{C}[\![e_n]\!]}\ \mathit{in}\ \mathcal{C}[\![e]\!]
\end{array}
$$

Figure 3.2: Cost function $\mathcal{C}$

where the definition of the cost function $\mathcal{C}$ is shown in Figure 3.2. Additionally, we add the following function "$\underline{+}$" to the transformed program:

$$
\begin{array}{rcll}
(x,\ y) \mathbin{\underline{+}} z & = & \mathit{case}\ x\ \mathit{of} & \\
 & & \{\quad c \to \mathit{let}\ v = y + z & \mathit{for\ all\ constructor}\ c/0\ \mathit{in}\ P \\
 & & \quad\ \ \mathit{in}\ (x,v) & \\
 & & c(\overline{x_n}) \to \mathit{let}\ v = y + z & \mathit{for\ all\ constructor}\ c/n\ \mathit{in}\ P \\
 & & \quad\ \ \mathit{in}\ (x,v)\} &
\end{array}
$$

For clarity, we will use in the examples the non-normalized version of this function:

$$
\begin{array}{rcll}
(c,\ k_1) \mathbin{\underline{+}} k_2 & = & (c,\ k_1 + k_2) & \text{for all constructor } c/0 \text{ in } P \\
(c(\overline{x_n}),\ k_1) \mathbin{\underline{+}} k_2 & = & (c(\overline{x_n}),\ k_1 + k_2) & \text{for all constructor } c/n \text{ in } P
\end{array}
$$

Basically, for each constructor term $c(\overline{x_n})$ computed in the original program, a pair of the form $(c(\overline{x'_n}), k)$ is computed in the transformed program, where k is the number of function unfoldings which are needed to produce the constructor "c". For instance, a constructor term—a list—like $(\mathtt{x} : \mathtt{y} : [])$ will have the form $(\mathtt{x} : (\mathtt{y} : ([], \mathtt{k_3}), \mathtt{k_2}), \mathtt{k_1})$ in the transformed program, where $\mathtt{k_1}$ is the cost of producing the outermost constructor ":", $\mathtt{k_2}$ the cost of producing the innermost constructor ":" and $\mathtt{k_3}$ the cost of producing the empty list. The auxiliary function "$\underline{+}$" is introduced to correctly increment the current cost.

The definition of the cost function $\mathcal{C}$ distinguishes the following cases depending on the structure of the expression in the right-hand side of the program rule, as it is depicted in Figure 3.2:

- Variables are not modified. Nevertheless, observe that now their domain is different, i.e., they denote pairs of the form $(c(\overline{x_n}), k)$.
- For constructor-rooted terms, their associated cost is always zero, since no function unfolding is needed to produce them.
- Each function call $f(x_1,\ldots,x_k)$ is replaced by its counterpart in the transformed

program, i.e., a call of the form $f_c(x_1, \ldots, x_n)$.

- Case expressions are transformed by leaving the case structure and then replacing each pattern p_i by the pair (p_i, k_i), where k_i denotes the cost of producing the pattern p_i (i.e., the cost of evaluating the case argument to some head normal form). Then, this cost is added to the transformed expression in the selected branch.

- Finally, let bindings are transformed by applying recursively function $\mathcal{C}$ to all the expressions.

Theorem 8 states the correctness of our approach. In this result, we only consider *ground* expressions—without free variables—as initial calls, i.e., we only consider "functional" computations. The use of non-ground calls will be discussed in Section 3.4.1. Below, we denote by Δ^c the heap obtained from Δ by replacing every binding $x \mapsto e$ in Δ by $x \mapsto \mathcal{C}[\![e]\!]$.

Theorem 8 *Let P be a program and let P_{cost} be its transformed version obtained by applying Definition 7. Then, given a ground expression e, we have $[\,] : e \Downarrow_k \Delta : v$ w.r.t. P iff $[\,] : \mathcal{C}[\![e]\!] \Downarrow \Delta^c : (\mathcal{C}[\![v]\!], k)$ w.r.t. P_{cost}.*

Proof. (sketch)
We prove a more general claim: for each cost-augmented derivation

$$\Gamma : e \Downarrow_k \Delta : v$$

w.r.t. P, there exists a standard derivation

$$\Gamma^c : e' \Downarrow \Delta^c : (\mathcal{C}[\![v]\!], k)$$

w.r.t. P_{cost} and vice versa, where e' is either a logical variable (according to Γ_c) or $e' = \mathcal{C}[\![e]\!]$.

We prove the claim by induction on the depth m of the derivation.
(Base case $m = 0$) In this case, only two rules can be applied:

- If e is a logical variable (i.e., a variable x such that $\Gamma[x] = x$) then, by rule Val of the cost-augmented semantics, it is $(\Gamma : x \Downarrow_0 \Gamma : x)$ in the original program P. Since e is a logical variable, $e' = (e, 0)$ and, by rule Val of the standard semantics, we also have $(\Gamma^c : (x, 0) \Downarrow \Gamma^c : (x, 0))$ in P_{cost}.

  Similarly, if $e = c(\overline{x_n})$, only rule Val can be applied: $(\Gamma : c(\overline{x_n}) \Downarrow_0 \Gamma : c(\overline{x_n}))$. Thus, $e' = (c(\overline{x_n}), 0)$ and, by rule Val of the standard semantics, we also have $(\Gamma^c : (c(\overline{x_n}), 0) \Downarrow \Gamma^c : (c(\overline{x_n})), 0)$.

- Otherwise, $e = x$ is a variable with $\Gamma = \Gamma[x \mapsto c(\overline{x_n}))]$ and rule VarCons of the cost-augmented semantics is applied in program P:

  $$\Gamma[x \mapsto (c(\overline{x_n})] : x \ \Downarrow_0 \ \Gamma[x \mapsto c(\overline{x_n})] : c(\overline{x_n})$$

  Therefore, in the instrumented program, P_{cost}, it is $\Gamma_c[x \mapsto (c(\overline{x_n}), k)] : x = \Gamma_c[x \mapsto (c(\overline{x_n}), k)] : (c(\overline{x_n}), k)$ and, thus, the cost k is not incremented.

(Inductive case $m > 1$) In this case, rules VarExp, Fun, Let, Or, Select, and Guess are applicable. We consider each case separately.

(VarExp) Then, e is a variable with $\Gamma[x] \neq x$ and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{\Gamma : e_1 \ \Downarrow_k \ \Delta : v}{\Gamma[x \mapsto e_1] : x \ \Downarrow_k \ \Delta[x \mapsto v] : v}$$

In the instrumented program, P_{cost}, it is $e' = \mathcal{C}[\![x]\!] = x$. Therefore, the following derivation holds in P_{cost}:

$$\frac{\Gamma^c : e_1 \ \Downarrow \ \Delta^c : (v, k)}{\Gamma^c[x \mapsto e_1] : x \ \Downarrow \ \Delta^c[x \mapsto v] : (v, k)}$$

Finally, the topmost judgement follows by the inductive hypothesis.

(Fun) Then, e is a function call and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{\Gamma : \rho(e_1) \ \Downarrow_k \ \Delta : v}{\Gamma : f(\overline{x_n}) \ \Downarrow_{k+1} \ \Delta : v}$$

where $f(\overline{y_n}) = e_1 \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$. Therefore, $e' = \mathcal{C}[\![f(\overline{x_n})]\!] = f_c(\overline{x_n})$ and $f_c(\overline{x_n}) = e \pm 1 \in P_{cost}$ iff $f(\overline{x_n}) = e \in P$. Hence, the following subderivation holds in P_{cost} by applying rules Let and Fun of the standard semantics:

$$\cfrac{\cfrac{\cfrac{\Gamma^c : \rho(e_1) \ \Downarrow \ \Delta^c : (v, k)}{\Gamma^c[y \mapsto 1] : \rho(e_1) \pm y \ \Downarrow \ \Delta^c : (v, k+1)}}{\Gamma^c : let \ y = 1 \ in \ \rho(e_1) \pm y \ \Downarrow \ \Delta^c : (v, k+1)}}{\Gamma^c : f_c(\overline{x_n}) \ \Downarrow \ \Delta^c : (v, k+1)}$$

Finally, the claim follows by applying the inductive hypothesis to the topmost judgement.

(Let) Then, $e = (let \ x = e_1 \ in \ e_2)$ is a let expression and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{\Gamma[y \mapsto \rho(e_1)] : \rho(e_2) \ \Downarrow_k \ \Delta : v}{\Gamma : let \ x = e_1 \ in \ e_2 \ \Downarrow_k \ \Delta : v}$$

where $\rho = \{x \mapsto y\}$. In the instrumented program, P_{cost}, it is $e' = \mathcal{C}[\![let\ x = e_1\ in\ e_2]\!] = let\ x = \mathcal{C}[\![e_1]\!]\ in\ \mathcal{C}[\![e_2]\!]$. Therefore, the following derivation holds in P_{cost}:

$$\frac{\Gamma^c[x \mapsto \rho(\mathcal{C}[\![e_1]\!])] : \mathcal{C}[\![e_2]\!] \ \Downarrow\ \Delta^c : (v,k)}{\Gamma^c : let\ x = \mathcal{C}[\![e_1]\!]\ in\ \mathcal{C}[\![e_2]\!] \ \Downarrow\ \Delta^c : (v,k)}$$

Finally, the topmost judgement follows by the inductive hypothesis.

(Select) Then, $e = (f)case\ x\ of\ \{\overline{p_k \to e_k}\}$ is a case expression and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{\Gamma : x \ \Downarrow_{k1}\ \Delta : c(\overline{y_n}) \qquad \Delta : \rho(e_i) \ \Downarrow_{k2}\ \Theta : v}{\Gamma : (f)case\ x\ of\ \{\overline{p_m \to e_m}\} \ \Downarrow_{k1+k2}\ \Theta : v}$$

where $p_i = c(\overline{x_n})$, $i \in \{1,\ldots,n\}$, and $\rho = \{\overline{x_n \mapsto y_n}\}$. Therefore, since x evaluates to a non-variable expression, $e' = \mathcal{C}[\![(f)case\ x\ of\ \{\overline{p_m \to e_m}\}]\!] = (f)case\ x\ of\ \{\overline{(p_m,k_m) \to \mathcal{C}[\![e_m]\!] \pm k_m}\}$. Then, the following subderivation holds in P_{cost} by applying rules Select of the standard semantics:

$$\frac{\Gamma^c : x \ \Downarrow\ \Delta^c : (c(\overline{y_n}),k1) \qquad \Delta^c : \rho(\mathcal{C}[\![e_i]\!]) \ \Downarrow\ \Theta : (v,k2)}{\Gamma^c : (f)case\ x\ of\ \{\overline{(p_m,k_m) \to \mathcal{C}[\![e_m]\!] \pm k_m}\} \ \Downarrow\ \Theta^c : (v,k1+k2)}$$

and the claim follows by applying the inductive hypothesis to the topmost judgements.

(Guess) Then, $e = fcase\ x\ of\ \{\overline{p_k \to e_k}\}$ is a flexible case expression and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{\Gamma : x \ \Downarrow_{k1}\ \Delta : y \qquad \Delta[y \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \ \Downarrow_{k2}\ \Theta : v}{\Gamma : fcase\ x\ of\ \{\overline{p_k \to e_k}\} \ \Downarrow_{k1+k2}\ \Theta : v}$$

where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, and $\overline{y_n}$ are fresh variables. Therefore, since x evaluates to a logical variable, it is $e' = \mathcal{C}[\![(f)case\ x\ of\ \{\overline{p_m \to e_m}\}]\!] = (f)case\ x\ of\ \{\overline{(p_m,k_m) \to \mathcal{C}[\![e_m]\!] \pm k_m}\}$. Then, the following subderivation holds in P_{cost} by applying rules Guess and Fun of the standard semantics:

$$\frac{\Gamma^c : x \ \Downarrow\ \Delta^c : (c(\overline{y_n}),k1) \qquad \Delta^c[y \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(\mathcal{C}[\![e_i]\!]) \ \Downarrow\ \Theta : (v,k2)}{\Gamma^c : (f)case\ x\ of\ \{\overline{(p_m,k_m) \to \mathcal{C}[\![e_m]\!] \pm k_m}\} \ \Downarrow\ \Theta^c : (v,k1+k2)}$$

and the claim follows by applying the inductive hypothesis to the topmost judgements.

□

Roughly speaking, we obtain the same cost by evaluating an expression w.r.t. the original program and the cost-augmented semantics defined in Figure 3.1 and by evaluating it w.r.t. the transformed program and the standard semantics.

3.4 The Transformation in Practice

In this section, we illustrate the usefulness of our approach by means of some selected examples and point out possible applications of the new technique.

3.4.1 Complexity Analysis

The generated time-equations are simple and, thus, they are appropriate to reason about the cost of evaluating expressions in our lazy language. We consider an example which shows the behavior of lazy evaluation:

$$
\begin{array}{lcl}
\texttt{hd xs} & = & \texttt{case xs of } \{(\texttt{y} : \texttt{ys}) \rightarrow \texttt{y}\} \\
\texttt{from x} & = & \texttt{let z} = \texttt{S x}, \\
 & & \quad \texttt{y} = \texttt{from z} \\
 & & \texttt{in x} : \texttt{y} \\
\texttt{nth n xs} & = & \texttt{case n of} \\
 & & \quad \{ \quad Z \rightarrow hd\ xs; \\
 & & \quad (S\ m) \rightarrow case\ xs\ of\ \{(y : ys) \rightarrow nth\ m\ ys\}\}
\end{array}
$$

where natural numbers are built from `Z` and `S`. Function `hd` returns the first element of a list, function `from` computes an infinite list of consecutive numbers (starting at its argument), and function `nth` returns the element in the n-th position of a given list. The transformed program is as follows:

$$
\begin{array}{lcl}
\texttt{hd}_\texttt{c}\ \texttt{xs} & = & \texttt{case xs of } \{(\texttt{y} : \texttt{ys}, \texttt{k}_\texttt{xs}) \rightarrow \texttt{y} \pm \texttt{k}_\texttt{xs}\} \pm \texttt{1} \\
\texttt{from}_\texttt{c}\ \texttt{x} & = & \texttt{let z} = (\texttt{S x}, \texttt{0}), \\
 & & \quad \texttt{y} = \texttt{from}_\texttt{c}\ \texttt{z} \\
 & & \texttt{in } (\texttt{x} : \texttt{y}, \texttt{0}) \pm \texttt{1} \\
\texttt{nth}_\texttt{c}\ \texttt{n xs} & = & \texttt{case n of} \\
 & & \quad \{ \ (\texttt{Z}, \texttt{k}_\texttt{n}) \rightarrow \texttt{hd}_\texttt{c}\ \texttt{xs} \pm \texttt{k}_\texttt{n}; \\
 & & \quad (\texttt{S m}, \texttt{k}_\texttt{n}) \rightarrow \texttt{case xs of} \\
 & & \qquad \{(\texttt{y} : \texttt{ys}, \texttt{k}_\texttt{xs}) \rightarrow \texttt{nth}_\texttt{c}\ \texttt{m ys } \pm \texttt{k}_\texttt{xs}\} \pm \texttt{k}_\texttt{n}\} \pm \texttt{1}
\end{array}
$$

Consider, for instance, the function call[2] "`nth (S (S (S Z))) (from Z)`", which returns the value `(S (S (S Z)))` with 9 function unfoldings. Consequently, the transformed call

$$\texttt{nth}_\texttt{c}\ (\texttt{S}\ (\texttt{S}\ (\texttt{S}\ (\texttt{Z}, \texttt{0}), \texttt{0}), \texttt{0}), \texttt{0})\ (\texttt{from}_\texttt{c}\ (\texttt{Z}, \texttt{0}))$$

returns the pair $((\texttt{S}\ (\texttt{S}\ (\texttt{S}\ (\texttt{Z}, \texttt{0}), \texttt{0}), \texttt{0}), \texttt{0}), \texttt{9})$. Let us now reason about the complexity of the original program. First, the equations of the transformed program can easily be transformed as follows:[3]

[2] Here, we *inline* the let bindings to ease the reading of the function calls.

[3] Since we only consider ground initial calls—and program rules have no extra-variables—we know that case expressions will never suspend.

$$
\begin{array}{lcl}
\mathtt{hd_c\ (y : ys,\ k_{xs})} & = & \mathtt{y} \pm (\mathtt{k_{xs}} + 1) \\
\mathtt{from_c\ x} & = & (\mathtt{x : from_c\ (S\ x, 0),\ 1}) \\
\mathtt{nth_c\ (Z,\ k_n)\ xs} & = & (\mathtt{hd_c\ xs}) \pm (\mathtt{k_n} + 1) \\
\mathtt{nth_c\ (S\ m,\ k_n)\ (y : ys,\ k_{xs})} & = & (\mathtt{nth_c\ m\ ys}) \pm (\mathtt{k_{xs}} + \mathtt{k_n} + 1)
\end{array}
$$

Here, we have unnormalized and unflattened the program by using inlining (for let expressions) and by moving the arguments of the case expressions to the left-hand sides of the rules. This transformation is correct as proved by Hanus and Prehofer (see Theorem 3.3 in [Hanus and Prehofer, 1999]. Now, we consider a generic call of the form $\mathtt{nth_c\ v_1\ (from\ v_2)}$, where $\mathtt{v_1}$ and $\mathtt{v_2}$ are constructor terms (hence all the associated costs are zero). By unfolding the call to $\mathtt{from_c}$ and replacing $\mathtt{k_n}$ by 0 (since $\mathtt{v_1}$ is a constructor term), we get:

$$
\begin{array}{lcl}
\mathtt{nth_c\ (Z,\ 0)\ (x : from_c\ (S\ x,\ 0),\ 1)} & = & (\mathtt{hd_c\ (x : from_c\ (S\ x,\ 0),\ 1)}) \pm 1 \\
\mathtt{nth_c\ (S\ m,\ 0)\ (x : from_c\ (S\ x,\ 0),\ 1)} & = & (\mathtt{nth_c\ m\ (from_c\ (S\ x,\ 0))}) \pm 2
\end{array}
$$

By unfolding the call to $\mathtt{hd_c}$ in the first equation, we have:

$$
\begin{array}{lcl}
\mathtt{nth_c\ (Z,\ 0)\ (x : from_c\ (S\ x,\ 0),\ 1)} & = & \mathtt{x} \pm 3 \\
\mathtt{nth_c\ (S\ m,\ 0)\ (x : from_c\ (S\ x,\ 0),\ 1)} & = & (\mathtt{nth_c\ m\ (from_c\ (S\ x,\ 0))}) \pm 2
\end{array}
$$

Therefore, each recursive call to $\mathtt{nth_c}$ requires 2 steps, while the base case requires 3 steps, i.e., the total cost of a call $\mathtt{nth\ v_1\ (from\ v_2)}$ is $n * 2 + 3$, where n is the natural number represented by $\mathtt{v_1}$.

A theorem prover may help to perform this task; and we think that *logical variables* may also help to perform this task. For instance, if one executes the function call $\mathtt{nth_c\ x\ (from_c\ (Z,\ 0))}$, where $\mathtt{x}$ is a logical variable, we get the following (infinite) sequence (computed bindings for $\mathtt{x}$ are applied to the initial call):

$$
\begin{array}{lcl}
\mathtt{nth_c\ (Z,\ n_1)\ (from_c\ (Z,\ 0))} & \Longrightarrow & (\mathtt{Z,\ 3 + n_1}) \\
\mathtt{nth_c\ (S\ (Z, n_1),\ n_2)\ (from_c\ (Z,\ 0))} & \Longrightarrow & (\mathtt{S\ (Z,\ 0),\ 5 + n_1 + n_2}) \\
\mathtt{nth_c\ (S\ (S\ (Z, n_1),\ n_2),\ n_3)\ (from_c\ (Z,\ 0))} & \Longrightarrow & (\mathtt{S\ (S\ (Z,\ 0), 0),\ 7 + n_1 + n_2 + n_3}) \\
\ldots &&
\end{array}
$$

If one considers that $\mathtt{n_1} = \mathtt{n_2} = \mathtt{n_3} = 0$ (i.e., that the first input argument is a constructor term), then it is fairly easy to check that 2 steps are needed for each constructor "S" of the first input argument of $\mathtt{nth_c}$, and 3 more steps for the final constructor "Z". Therefore, we have $2 * n + 3$, where n is the natural number represented by the first input argument of $\mathtt{nth_c}$ (i.e., we obtain the same result as with the previous analysis).

On the other hand, there are already some practical systems, like `ciaopp` [Hermenegildo *et al.*, 2003], which are able to solve certain classes of equations. They basically try to match the produced recurrence with respect to some predefined "patterns of equations" which represent classes of recurrences whose solution is known. These ideas could be also adapted to our context.

3.4.2 Other Applications and Extensions

Here we discuss some other practical applications which could be done by extending the developments in this chapter.

There are several approaches in the literature to *symbolic profiling* (e.g., [Albert and Vidal, 2002; Sansom and Peyton-Jones, 1997]). Here, the idea is to extend the standard semantics with the computation of cost information (similarly to our step-counting semantics, but including different cost criteria like case evaluations, function applications, etc). A drawback of this approach is that one should modify the language interpreter or compiler in order to implement it. Unfortunately, this possibility usually implies a significant amount of work. An alternative could be the development of a meta-interpreter extended with the computation of symbolic costs, but the execution of such a meta-interpreter would involve a significant overhead at runtime. Therefore, we have followed the first alternative. It is described in Chapter 4.

In this context, our program transformation could be seen as a preprocessing stage (during compilation), which produces an *instrumented* program whose execution (in a standard environment) returns not only the computed values but also the costs of computing such values. For this approach to be feasible, an important property is the *executability* of the transformed program. The following result states such a property.

Lemma 9 *Let P be a normalized flat program (respecting the syntax of Figure 2.4) and P_{cost} be the set of rules obtained according to Definition 7. Then P_{cost} is also a valid normalized flat program.*

Proof. The claim follows easily from the following facts:

1. the original program is not modified; Only new expressions, and a new function definition ($\pm$)—which is flat and normalized—are added;
2. variables in left-hand sides are not modified; hence, the new time equations have only different variables as parameters in the left-hand side;
3. cost function $\mathcal{C}$ do not introduce *extra-variables* in right-hand sides (apart from those variables in let bindings, which are allowed); and
4. case structures are not modified.

□

There are two possible extensions that we will develop in the next chapter. On the one hand, we could adapt the use of *cost centers* in order to attribute costs to concrete program functions or expressions, similarly to [Albert and Vidal, 2002; Sansom and Peyton-Jones, 1997]. For instance, rather than using pairs (v, k) in the

transformed program, we could use triples of the form (v, k, cc) which indicate the cost center cc to which the cost k should be attributed. On the other hand, counting the number of steps is useful in general, but sometimes it is not very informative. The transformation could also be extended to cover other different cost criteria, e.g., the number of case evaluations, the number of function applications, etc. (similarly to symbolic profilers). In principle, this extension seems difficult since some of these cost criteria are *dynamic*, i.e., they can only be measured at runtime. Therefore, it is not easy to account for them statically in the transformed program. In the next chapter, we first define an instrumented semantics able to compute different symbolic costs; and then, we also define an equivalent program transformation which makes practical the profiling of realistic programs.

Another possible application of our technique is the use of the computed cost equations to produce standard recurrence equations which can be solved by existing tools. For this purpose, a method to estimate the size of the terms would be needed. For instance, one could consider the use of symbolic norms [Lindenstrauss and Sagiv, 1997]:

Definition 10 (symbolic norm) *A symbolic norm is a function $\|\cdot\| : \mathcal{T}(\Sigma, \mathcal{V}) \to I\!N$ such that*

$\|t\| = m + \sum_{i=0}^{n} k_i \|t_i\|$

where $t = f(t_1, ..., t_n), n \geq 0$, and m and $k_1, ..., k_n$ are non-negative integer constants depending only on f/n.

By using symbolic norms, we can associate sizes to terms and use them to produce recurrence systems as it is shown in the next example.

Example 10 *Consider again the unnormalized and unflattened versions of the functions* $\texttt{hd}_\texttt{c}$ *and* $\texttt{nth}_\texttt{c}$ *presented in Section 3.4.1 together with the symbolic norms:*

$$||t||_{ts} = \begin{cases} \sum_{i=1}^{n} ||t_i||_{ts} & \text{if } t = f(t_1, \ldots, t_n),\ n \geqslant 0 \\ n & \text{if } t \text{ is a variable} \end{cases}$$

$$||t||_{ll} = \begin{cases} 1 + ||xs|| & \text{if } t = (x : xs) \\ n & \text{if } t \text{ is a variable} \\ 0 & \text{otherwise} \end{cases}$$

which respectively correspond to the term-size and list-length symbolic norms. Cost variables (i.e., those variables used in the rules to pass the cumulated cost from one call to another) are initially bound to zero; and they take values during a computation. As we want to statically reason about the cost of the program, we can replace that costs associated to arguments by zero. Hence, we have the program:

$$\begin{array}{lcl}
\mathtt{hd_c\ (y : ys,\ 0)} & = & \mathtt{y} \mathbin{\underline{+}} (\mathtt{0} + \mathtt{1}) \\
\mathtt{nth_c\ (Z,\ 0)\ xs} & = & (\mathtt{hd_c\ xs}) \mathbin{\underline{+}} (\mathtt{0} + \mathtt{1}) \\
\mathtt{nth_c\ (S\ m,\ 0)\ (y : ys,\ 0)} & = & (\mathtt{nth_c\ m\ ys}) \mathbin{\underline{+}} (\mathtt{0} + \mathtt{0} + \mathtt{1})
\end{array}$$

After eliminating costs from the left-hand side, replacing $\underline{+}$ *by* $+$*, and simplifying the right-hand side, we can apply the norms to the arguments of function calls and function definitions:*

$$\begin{array}{lcl}
\mathtt{hd_c}\ ||\mathtt{y : ys}||_{\mathtt{ll}} & = & \mathtt{y} + \mathtt{1} \\
\mathtt{nth_c}\ ||\mathtt{Z}||_{\mathtt{ts}}\ ||\mathtt{xs}||_{\mathtt{ll}} & = & \mathtt{hd_c}\ ||\mathtt{xs}||_{\mathtt{ll}} + \mathtt{1} \\
\mathtt{nth_c}\ ||\mathtt{S\ m}||_{\mathtt{ts}}\ ||(\mathtt{y : ys})||_{\mathtt{ll}} & = & \mathtt{nth_c}\ ||\mathtt{m}||_{\mathtt{ts}}\ ||\mathtt{ys}||_{\mathtt{ll}} + \mathtt{1}
\end{array}$$

Then, we delete all expressions in the right-hand side except costs and function calls, because they do not contribute to the cost. The result is the following set of recurrence rules:

$$\begin{array}{lcl}
\mathtt{hd_c\ (n + 1)} & = & \mathtt{1} \\
\mathtt{nth_c\ 0\ \ n} & = & (\mathtt{hd_c\ n}) + \mathtt{1} \\
\mathtt{nth_c\ (n + 1)\ (m + 1)} & = & (\mathtt{nth_c\ n\ m}) + \mathtt{1}
\end{array}$$

Therefore,

$$\begin{array}{lcll}
\mathtt{hd\ n} & = & \mathtt{1} & \textit{where } n > 0 \\
\mathtt{nth\ n\ m} & = & \mathtt{n} + \mathtt{2} & \textit{where } n, m \geq 0
\end{array}$$

which is interpreted as: "computing the head of a list of n elements implies one function unfolding" *and* "computing the n^{th} element of a list of m elements implies $n + 2$ function unfoldings"

Finally, the proposed transformation could also be useful in the context of *program debugging*. In particular, some abstract interpretation based systems, like `ciaopp` [Hermenegildo *et al.*, 2003], allows the user to include annotations in the program with the predicted cost. This cost will be then compared with the one automatically generated by the compiler (here, our program transformation can be useful to compute this cost). If both cost functions are not equivalent, this might indicate a programmer error; for instance, the program may contain undesired loops which directly affect the time complexity.

3.5 Related Work

There exist several approaches to automatic complexity analysis in the context of *eager* functional languages (e.g., [Liu and Gómez, 1998]). Here, the construction

of cost equations is rather straightforward since the order of evaluation is statically determined, i.e., given a nested function call $f(g(t))$, we first compute the cost of evaluating $g(t)$ to some value (say v) and, then, the cost of evaluating $f(v)$. This contrasts with lazy languages, where the order of evaluation depends dynamically on the *context* of the given expression. This means that the above rule gives only a rough upper bound. In general, if f only *demands* g to produce a certain number of outermost constructors (to be consumed by f), function g will not be further evaluated. Consider, for instance, the following function definitions:

$$\begin{array}{lcl} \texttt{from x} & = & \texttt{x} : \texttt{from}\ (\texttt{x} + \texttt{1}) \\ \texttt{head}\ (\texttt{x} : \texttt{xs}) & = & \texttt{x} \end{array}$$

where function `from` computes an infinite list of consecutive numbers starting at `x` and function `head` returns the first element of a list. Given a call of the form `head (from 1)`, only two unfolding steps are necessary. In an eager language, this evaluation will not terminate since function `from` is infinite. Note that determining whether the evaluation of some subexpression is *needed* is in general an undecidable problem.

Let us review some previous approaches to measuring the time-complexity of lazy functional programs. There are some works which transform the original program into an equivalent one under a call-by-value semantics and then analyze the transformed program (e.g., [Métayer, 1988]). A drawback of this approach is that the translation generally makes the program significantly more complex. Furthermore, the transformation does not ensure that the number of evaluation steps is preserved through the transformation. An interesting approach is taken by Sands [1990] (which is based on previous work by Wadler [1988]). His transformation produces time-equations which *approximate* the time complexity. The approximation consists in giving lower or upper bounds which are based on the results of a previous strictness analysis. An important difference with our transformation is that we do not need such a strictness analysis. Moreover, these approaches may produce equations which are non-terminating even if the original program terminates. Our correctness results ensure that this cannot happen in our transformation. There is also an interesting line of research aimed at achieving *exact* time analyses. For instance, Bjerner and Holmström [1989] produce time equations which are *precise*, i.e., they give the *exact* time complexity. The main problem here is that the equations cannot be solved mechanically. An advantage of our approach is that time equations are obtained as a source-to-source program transformation and they are amenable to be further manipulated and even executed in the same language. Rosendahl [1989] defined a program transformation for first-order functional programs whose transformed programs compute the number of call-by-value reduction steps performed by the original program. Posteriorly, Sands [1996] focussed on call-by-name reduction steps. His goal was to prove the correctness of

fold/unfold transformations, and his technique does it by annotating programs with "ticks", an identity function used to simulate a single tick of computation time. Later, Voigtländer [2002] defined a program transformation to annotate left-linear rewrite systems with a special identity function similar to the Sands' ticks. In this case, the function was used to measure the number of call-by-need steps in a computation. The annotations were posteriorly used to prove syntactic conditions under which a transformed program is more efficient than the original program. This annotating scheme used is similar to ours except that it does not consider the use of logical variables.

Our work is implicitly related to the *profiling* of lazy functional (logic) languages. Practical approaches to profiling are based on cost-augmented semantics. The well-known work of Sansom and Peyton-Jones [1997] formalizes a profiling scheme by means of a small-step semantics enhanced with cost information. This semantics is proved to be equivalent to the natural semantics of Launchbury [1993] and the cost criteria are experimentally related to execution times. In the field of functional logic programming, a similar approach has been taken in [Albert and Vidal, 2002]. The previous section discussed how our program transformation could be used for the same purpose.

Finally, in the field of partial evaluation, time equations have been used to assess the effectiveness of the transformation process (see, e.g., [Albert *et al.*, 2001; Vidal, 2002]). An essential difference w.r.t. our approach is the structure of the generated cost equations. In [Albert *et al.*, 2001; Vidal, 2002], they are generated throughout the partial evaluation process in order to mimic the structure of the partially evaluated program. Also, they are sometimes not executable (i.e., when the generated equations are not "closed"). Our approach is thus more general, produces executable equations and is not tied to a particular program transformation (as the aforementioned works).

Chapter 4

Runtime Profiling of Functional Logic Programs

In this chapter, we introduce a profiling scheme for modern functional logic languages covering notions like laziness, sharing, and non-determinism. It extends the profiling scheme presented in the former chapter by computing different kinds of symbolic costs and not only the number of function unfoldings.

Firstly, we instrument a natural (big-step) semantics in order to associate a symbolic cost to each basic operation (e.g., variable updates, function unfoldings, case evaluations). While this *cost semantics* provides a formal basis to analyze the cost of a computation, the implementation of a cost-augmented interpreter based on it would introduce a huge overhead. Therefore, we also introduce a sound transformation that instruments a program such that its execution—under the standard semantics—yields not only the corresponding results but also the associated costs. Finally, we describe a prototype implementation of a profiler based on the developments in this chapter.

Part of the material of this chapter has been presented in the 14th *International Symposium on Logic-based Program Synthesis and Transformation* (LOPSTR'04) celebrated in Verona (Italy) in 2004 [Braßel *et al.*, 2004a].

4.1 Introduction

Comparing always involves measuring. The only way to determine whether a version of a program has improved the previous version, or whether two programs are equivalently efficient, is measuring their performance. The importance of profiling in improving the performance of programs is widely recognized. Profiling tools are essential for the programmer to analyze the effects of different source-to-source program ma-

nipulations (e.g., partial evaluation, specialization, optimization, etc). Despite this, one can find very few profiling tools for modern declarative languages. This situation is mainly explained by the difficulty to correctly map execution costs to source code, which is much less obvious than for imperative languages. In this chapter, we tackle the definition of a profiling scheme for modern functional logic languages covering notions like laziness, sharing, and non-determinism (like Curry [Hanus, 2006a] and Toy [López-Fraguas and Sánchez-Hernández, 1999]); currently, there is no profiling tool practically applicable for such languages.

When profiling the runtime of a given program, the results highly depend on the considered language implementation. Indeed, computing actual runtimes is not always the most useful information for the programmer. Runtimes may help to detect that some function is expensive but they do not explain *why* it is expensive (e.g., is it called many times? Is it heavily non-deterministic?).

In order to overcome these drawbacks, we introduce a *symbolic* profiler which outputs the number of basic operations performed in a computation. For this purpose, we start from the natural semantics for functional logic programs of Figure 2.5 and instrument it with the computation of symbolic costs associated to the basic operations of the semantics: variable lookups, function unfoldings, case evaluations, etc. These operations are performed, in one form or another, by likely implementations of modern functional logic languages. Our *cost semantics* constitutes a formal model of the attribution of costs in our setting. Therefore, it is useful not only as a basis to develop profiling tools but also to analyze the costs of a program computation (e.g., to formally prove the effectiveness of some program transformation).

Trivially, one can develop a profiler by implementing an instrumented interpreter which follows the previous cost semantics. However, this approach is not practicable as it demands a huge overhead, making the profiling of realistic programs impossible. Thus, in a second step, we design a source-to-source transformation that instruments a program such that its execution—under the standard semantics—outputs not only the corresponding results but also the associated costs. We formally prove the correctness of our transformation (i.e., the costs computed in a source program w.r.t. the cost semantics are equivalent to the costs computed in the transformed program w.r.t. the standard semantics) and, finally, we discuss a prototype implementation of a profiler based on the developments presented in this chapter.

The chapter is organized as follows. In the next section, we informally introduce our model for profiling functional logic computations. Section 4.3 formalizes an instrumented semantics which also computes cost information. Section 4.4 introduces a transformation instrumenting programs to compute symbolic costs. Section 4.5 describes an implementation of a profiler for Curry programs. Finally, Section 4.6 includes a comparison to related work and concludes.

4.2 A Runtime Profiling Scheme

Traditionally, profiling tools attribute execution costs to the functions or procedures of the considered program. Following [Albert and Vidal, 2002; Sansom and Peyton-Jones, 1997], in this chapter we take a more flexible approach which allows us to associate a *cost center* with any expression of interest. This allows the programmer to choose an appropriate granularity for profiling, ranging from whole program phases to single subexpressions in a function. Nevertheless, our approach can easily be adapted to work with automatically instrumented cost centers; for instance, if one wants to use the traditional approach in which all functions are profiled, each function can be automatically annotated by introducing a cost center for the entire right-hand side. Cost centers are marked with the (built-in) function `scc` (for `s`et `c`ost `c`enter).

Intuitively speaking, given an expression "`scc`(cc, e)", the costs attributed to cost center cc are the entire costs of evaluating e as far as the enclosing context demands it, including the cost of

- evaluating any function called by the evaluation of the expression e,

but excluding the cost of

- evaluating the *free* variables of e (i.e., those variables which are not introduced by a let binding in e) and
- evaluating any `scc`-expressions within e or within any function which is called from e.

In this chapter we consider again the class of normalized flat programs to formalize our profiling scheme. The following program contains two versions of a function to compute the length of a list (for readability, we show the non-normalized version of function `main`):

```
len(x) = fcase x of { []     → 0;
                    (y:ys) → let z = 1,
                                 w = len(ys)
                             in z + w }
len2s(x) = fcase x of { []     → 0;
                      (y:ys) → fcase ys of
                                 { []     → 1;
                                   (z:zs) → let w = 2,
                                                v = len2s(zs)
                                            in w + v          } }
main = let list = scc("list",[1..5000])
       in scc("len",len(list)) + scc("len2s",len2s(list))
```

Cost criteria	Symbol
Function unfolding	F
Binding a logical variable	B
Variable update	U
Variable lookup	V

Cost criteria	Symbol
Allocating a heap object	H
Case evaluation	C
Non-deterministic choice	N
Entering an `scc`	E

Table 4.1: Basic costs

Here, `main` computes twice the length of the list [`1..5000`], which is a standard predefined way to define the list $[1, 2, 3, \ldots, 4999, 5000]$. Each computation of the length uses a different function, `len` and `len2s`. Intuitively, `len2s` is more efficient than `len` because it performs half the number of function calls (indeed, `len2s` has been obtained by unfolding function `len`). This is difficult to check with traditional profilers because the overhead introduced to build the list hides the differences between `len` and `len2s`. For instance, the computed runtimes in the PAKCS environment [Hanus *et al.*, 2006] for Curry are 9980 ms and 9990 ms for `len`([`1..5000`]) and `len2s`([`1..5000`]), respectively.[1]

Should one conclude that `len` and `len2s` are equally efficient? In order to answer this question, a profiler based on *cost centers* can be very useful. In particular, by including the three cost centers shown in the program above, the costs of `len`, `len2s`, and the construction of the input list can be clearly distinguished. With our execution profiler which distributes the execution time to different cost centers (its implementation is discussed in Section 4.5.1), we have experimentally checked that both functions are not equally efficient:

cost center	`main`	`list`	`len`	`len2s`
runtimes	17710	7668966	1110	790

Here, runtimes are expressed in a number of "ticks" (a basic time unit). Thanks to the use of cost centers, we can easily check that `len2s` is an improvement of `len`. However, what is the reason for such an improvement? We introduce *symbolic* costs—associated with the basic operations of the language semantics—so that a deeper analysis can be made. The considered kinds of costs are shown in Table 4.1. For the example above, our symbolic profiler returns the following results (only the most relevant costs for this example are shown):

[1] The slow execution is due to the fact that experiments were performed with a version of the above program that uses numbers built from `Z` and `Succ` (to avoid the use of built-in functions).

	`main`	`list`	`len`	`len2s`
H	5000	61700	5100	5100
V	5100	280400	5100	5100
C	5100	280400	5100	5100
F	5300	168100	5100	2600

From this information, we observe that only function unfoldings (F) are halved, while the remaining costs are equal for both `len` and `len2s`. Therefore, we can conclude that, in this example, unfolding function `len` with no input data only improves cost F (which has a small impact on current compilers, as it has been shown before).

4.3 Cost Semantics

In this section, we instrument a natural (big-step) semantics for functional logic languages (defined in Figure 2.5) with the computation of symbolic costs. Figure 4.1 shows the cost-augmented semantics. Here, $\Gamma[x \overset{cc}{\mapsto} e]$ denotes a heap with $\Gamma[x] = e$ and associated cost center cc, i.e., we use this notation either as a condition on a heap Γ or as a modification of Γ. We use judgements of the form "$cc, \Gamma : e \Downarrow_\theta \Delta : v, cc_v$" which are interpreted as "in the context of heap Γ and cost center cc, the expression e evaluates to value v with associated cost θ, producing a new heap Δ and cost center cc_v".

In order to evaluate a variable which is bound to a constructor-rooted term in the heap, rule VarCons reduces the variable to this term. Here, cost V is attributed to the current cost center cc to account for the variable lookup (this attribution is denoted by $\{cc \leftarrow V\}$ and similarly for the other cost symbols).

Rule VarExp achieves the effect of *sharing*. If the variable to be evaluated is bound to some expression in the heap, then the expression is evaluated and the heap is updated with the computed value; finally, we return this value as the result. In addition to counting the cost θ of evaluating expression e, both V and U are attributed to cost centers cc and cc_v, respectively.

For the evaluation of a value, rule Val returns it without modifying the heap. No costs are attributed in this rule since actual implementations have no counterpart for this action.

Rule Fun corresponds to the unfolding of a function call. The result is obtained by reducing the right-hand side of the corresponding rule (we assume that the considered program P is a global parameter of the calculus). Cost F is attributed to the current cost center cc to account for the function unfolding.

Rule Let adds its associated binding to the heap and proceeds with the evaluation of its main argument. Note that we give the introduced variable a fresh name in order

(VarCons) $cc, \Gamma[x \overset{cc_c}{\longmapsto} c(\overline{x_n})] : x \ \Downarrow_{\{cc \leftarrow V\}} \ \Gamma[x \overset{cc_c}{\longmapsto} c(\overline{x_n})] : c(\overline{x_n}), cc_c$

(VarExp) $$\frac{cc_e, \Gamma : e \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma[x \overset{cc_e}{\longmapsto} e] : x \ \Downarrow_{\{cc \leftarrow V\}+\theta+\{cc_v \leftarrow U\}} \ \Delta[x \overset{cc_v}{\longmapsto} v] : v, cc_v}$$ (where e is not a value)

(Val) $cc, \Gamma : v \ \Downarrow_{\{\,\}} \ \Gamma : v, cc$ (where v is a value)

(Fun) $$\frac{cc, \Gamma : \rho(e) \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : f(\overline{x_n}) \ \Downarrow_{\{cc \leftarrow F\}+\theta} \ \Delta : v, cc_v}$$ (where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$)

(Let) $$\frac{cc, \Gamma[y \overset{cc}{\longmapsto} \rho(e')] : \rho(e) \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : let \ x = e' \ in \ e \ \Downarrow_{\{cc \leftarrow H\}+\theta} \ \Delta : v, cc_v}$$ (where $\rho = \{x \mapsto y\}$ and y is fresh)

(Or) $$\frac{cc, \Gamma : e_i \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : e_1 \ or \ e_2 \ \Downarrow_{\{cc \leftarrow N\}+\theta} \ \Delta : v, cc_v}$$ (where $i \in \{1, 2\}$)

(Select) $$\frac{cc, \Gamma : x \ \Downarrow_{\theta_1} \ \Delta : c(\overline{y_n}), cc_c \qquad cc, \Delta : \rho(e_i) \ \Downarrow_{\theta_2} \ \Theta : v, cc_v}{cc, \Gamma : (f)case \ x \ of \ \{\overline{p_k \to e_k}\} \ \Downarrow_{\theta_1+\{cc \leftarrow C\}+\theta_2} \ \Theta : v, cc_v}$$ (where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$)

(Guess) $$\frac{cc, \Gamma : x \ \Downarrow_{\theta_1} \ \Delta : y, cc_y \qquad cc, \Delta[y \overset{cc}{\mapsto} \rho(p_i), \overline{y_n \overset{cc}{\mapsto} y_n}] : \rho(e_i) \ \Downarrow_{\theta_2} \ \Theta : v, cc_v}{cc, \Gamma : fcase \ x \ of \ \{\overline{p_k \to e_k}\} \ \Downarrow_{\theta_1+\{cc \leftarrow V, cc \leftarrow U, cc \leftarrow B, cc \leftarrow n*H\}+\theta_N+\theta_2} \ \Theta : v, cc_v}$$ (where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, $\overline{y_n}$ are fresh variables, and $\theta_N = \{cc \leftarrow N\}$ if $k > 1$ and $\theta_N = \{\,\}$ if $k = 1$)

(SCC) $$\frac{cc', \Gamma : e \ \Downarrow_\theta \ \Delta : v, cc_v}{cc, \Gamma : scc(cc', e) \ \Downarrow_{\theta+\{cc' \leftarrow E\}} \ \Delta : v, cc_v}$$

Figure 4.1: Rules of the cost semantics

to avoid variable name clashes. In this case, cost H is added to the current cost center cc.

Rule Or non-deterministically reduces an *or* expression to either the first or the second argument. N is attributed to the current cost center to account for a non-deterministic step.

Rule Select corresponds to the evaluation of a case expression whose argument reduces to a constructor-rooted term. In this case, we select the appropriate branch and, then, proceed with the evaluation of the expression in this branch by applying the corresponding matching substitution. In addition to the costs of evaluating the case argument, θ_1, and the selected branch, θ_2, we add cost C to the current cost center cc to account for the pattern matching.

Rule Guess applies when the argument of a flexible case expression reduces to

a logical variable. It binds this variable to one of the patterns and proceeds by evaluating the corresponding branch. If there is more than one branch, one of them is chosen non-deterministically. Renaming the pattern variables is necessary to avoid name clashes. We also update the heap with the (renamed) logical variables of the pattern. In addition to counting the costs of evaluating the case argument, θ_1, and the selected branch, θ_2, we attribute to the current cost center cc costs V (for determining that y is a logical variable), U (for updating the heap from $y \mapsto y$ to $y \mapsto \rho(p_i)$), B (for binding a logical variable), $n * H$ (for adding n new bindings into the heap) and, if there is more than one branch, N (for performing a non-deterministic step). Note that no cost C is attributed to cost center cc (indeed, cost B is alternative to cost C).

Finally, rule SCC evaluates an scc-expression by reducing the expression e in the context of the new cost center cc'. Accordingly, cost E is added to cost center cc'.

A proof of a judgement corresponds to a derivation sequence using the rules of Figure 4.1. Given a program P, the *initial configuration* has the form "$cc_{main}, [\,] : main$", where cc_{main} is a distinguished cost center. If the judgement

$$cc_{main}, [\,] : main \;\Downarrow_\theta\; \Gamma : v, cc_v$$

holds, we say that $main$ evaluates to value v with associated cost θ. The computed *answer* can be extracted from the final heap Γ by a simple process of *dereferencing*.

Obviously, the cost semantics is a conservative extension of the original big-step semantics of Figure 2.5, since the computation of cost information imposes no restriction on the application of the rules of the semantics.

4.4 Cost Instrumentation

As mentioned before, implementing an interpreter for the cost semantics of Figure 4.1 is impracticable. It would involve too much overhead to profile any realistic program. Thus, we introduce a transformation to instrument programs in order to compute the symbolic costs:

Definition 11 (cost transformation) *Given a program P, its cost instrumented version P_{cost} is obtained as follows: for each program rule*

$$f(x_1, \ldots, x_n) = e$$

P_{cost} includes, for each cost center cc in P, one rule of the form

$$f_{cc}(x_1, \ldots, x_n) = F_{cc}([\![e]\!]_{cc})$$

where $F_{cc}(e)$ is the identity function on e. Counting the calls to F_{cc} in the proof tree corresponds to the number of F's accumulated in cost center cc. Function $[\![\,]\!]$ (shown in Figure 4.2) is used to instrument program expressions.

$$
\begin{array}{rcl}
[\![x]\!]_{cc} & = & V_{cc}(x) \\
[\![c(x_1,\ldots,x_n)]\!]_{cc} & = & c(cc,x_1,\ldots,x_n) \\
[\![f(x_1,\ldots,x_n)]\!]_{cc} & = & f_{cc}(x_1,\ldots,x_n) \\
[\![let\ x = e'\ in\ e]\!]_{cc} & = & H_{cc}\left(\begin{array}{ll} let\ x = x\ in\ [\![e]\!]_{cc} & \text{if } e' = x \\ let\ x = [\![e']\!]_{cc}\ in\ [\![e]\!]_{cc} & \text{if } e' = c(\overline{y_n}) \\ let\ x = update([\![e']\!]_{cc})\ in\ [\![e]\!]_{cc} & \text{otherwise} \end{array}\right) \\
[\![e_1\ or\ e_2]\!]_{cc} & = & N_{cc}([\![e_1]\!]_{cc}\ or\ [\![e_2]\!]_{cc}) \\
[\![case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]_{cc} & = & case\ [\![x]\!]_{cc}\ of\ \{\overline{p'_k \to C_{cc}([\![e_k]\!]_{cc})}\} \\
 & & \text{where } p'_i = c(cc',\overline{y_n}) \text{ for all } p_i = c(\overline{y_n})
\end{array}
$$

$$
\begin{array}{l}
[\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!]_{cc} \\
\quad = \quad if\ isVar(x) \\
\qquad then\ V_{cc}(U_{cc}(B_{cc}(\theta_N(fcase\ [\![x]\!]_{cc}\ of\ \{\overline{p'_k \to |p_k| * H_{cc}([\![e_k]\!]_{cc})}\})))) \\
\qquad else\ fcase\ [\![x]\!]_{cc}\ of\ \{\overline{p'_k \to C_{cc}([\![e_k]\!]_{cc})}\} \\
\text{where } p'_i = c(cc,\overline{y_n}) \text{ for all } p_i = c(\overline{y_n}) \text{ and } \theta_N(e) = \left\{\begin{array}{ll} e & \text{if } k = 1 \\ N_{cc}(e) & \text{if } k > 1 \end{array}\right.
\end{array}
$$

$$
[\![scc(cc',e)]\!]_{cc} \quad = \quad E_{cc'}([\![e]\!]_{cc'})
$$

Here, $|p|$ denotes the arity of pattern p, i.e., $|p| = n$ if $p = c(\overline{x_n})$, and the auxiliary function *update* is used to attribute cost U to the cost center of the computed value:

$$update(x) = case\ x\ of\ \{\overline{c_k(cc_k,\overline{x_{n_k}}) \to U_{cc_k}(c_k(cc_k,\overline{x_{n_k}}))}\}$$

where $c_1,\ldots,c_k$ are the program constructors.

Figure 4.2: Cost transformation $[\![\ ]\!]_{cc}$ for instrumenting expressions

Observe that the transformed program contains as many variants of each function of the original program as the number of different cost centers. Semantically, all these variants are equivalent; the only difference is that we obtain the costs of the computation by counting the calls to the different cost center identity functions (like F_{cc}).

Program instrumentation is mainly performed by function $[\![\ ]\!]_{cc}$, where cc denotes the current cost center. Let us informally explain how the transformation proceeds. We distinguish the following cases depending on the expression, e, in a call of the form $[\![e]\!]_{cc}$:

- If e is a variable, a call to function V_{cc} is added to attribute cost V to cost center cc.
- If $e = c(\overline{x_n})$ is a constructor-rooted term, we add a new argument to store the current cost center. This is necessary to attribute cost U to the appropriate

cost center (i.e., to the cost center of the computed value, see Figure 4.1).

- A call to a function $f(\overline{x_n})$ is translated to a call to the function variant corresponding to cost center cc.

- If $e = (let\ x = e_1\ in\ e_2)$ is a let expression, a call to function H_{cc} is always added to attribute cost H to cost center cc. Additionally, if the binding is neither a logical variable nor a constructor-rooted term, the cost center cc_i, $1 \leq i \leq k$, of the computed value is determined (by means of an auxiliary function *update*, see Figure 4.2) and a call to U_{cc_i} is added to attribute cost U to that cost center.

  The auxiliary function *update* is needed when transforming `Let` expressions. When a `Let` expression introduces a new binding into the heap that binds a variable to an expression that can be further evaluated, it is unknown when this binding will be updated. Therefore, the transformation introduces function *update* into the binding. This ensures that a cost U will be attributed to the cost center of the computed value only if (and when) it is computed.

- If $e = (e_1\ or\ e_2)$ is a disjunction, a call to N_{cc} is added to attribute N to cost center cc.

- If $e = case\ x\ of\ \{\overline{p_k \to e_k}\}$ is a rigid case expression, we recursively transform both the case argument and the expression of each branch, where a call to C_{cc} is added to attribute cost C to cost center cc. Observe that the cost center, cc', of the patterns is not used (it is only needed in the auxiliary function *update*).

- If $e = fcase\ x\ of\ \{\overline{p_k \to e_k}\}$ is a flexible case expression, a runtime test (function *isVar*) is needed to determine whether the argument evaluates to a logical variable or not. This function can be found, e.g., in the library `Unsafe` of PAKCS. If it does not evaluate to a logical variable, we proceed as in the previous case. Otherwise, we add calls to functions V_{cc}, U_{cc}, B_{cc}, and N_{cc} (if $k > 1$). Also, in each case branch, calls to H_{cc} are added to attribute the size of the pattern to cost center cc. Here, we use $n * H_{cc}$ as a shorthand for writing n nested calls to H_{cc} (in particular, $0 * H_{cc}$ means that no call to H_{cc} is written).

- Finally, if $e = scc(cc', e')$ is an scc-expression, a call to function E_{cc} is added. More importantly, we update the current cost center to cc' in the recursive transformation of e'.

Derivations with the standard semantics (i.e., without cost centers) are denoted by $([\,] : main \Downarrow \Gamma_c : v)$. Given a heap Γ, we denote by Γ_c the set of bindings $x \mapsto e'$ such that $x \overset{cc}{\longmapsto} e$ belongs to Γ, where $e' = e$ if e is a logical variable, $e' = [\![e]\!]_{cc}$ if $e = c(\overline{x_n})$, or $e' = update([\![e]\!]_{cc})$ otherwise. Also, in order to make explicit the output of the

instrumented program with the standard semantics, we write $([\,] : main \Downarrow^{\theta} \Gamma_c : v)$, where θ records the set of calls to cost functions (e.g., H_{cc}, F_{cc}).

The correctness of our program instrumentation is stated as follows:

Theorem 12 (correctness) *Let P be a program and P_{cost} be its cost instrumented version. Then,*

$(cc_{main}, [\,] : main \Downarrow_{\theta} \Gamma : v, cc)$ *in P iff* $([\,] : main_{cc_{main}} \Downarrow^{\theta} \Gamma_c : v')$ *in P_{cost}*

where $v = v'$ (if they are variables) or $v = c(\overline{x_n})$ and $v' = c(cc, \overline{x_n})$.

Proof. (sketch)

We prove a more general claim: for each cost-augmented derivation

$$cc_e, \Gamma : e \Downarrow_{\theta} \Delta : v, cc_v$$

in P, there exists a standard derivation

$$\Gamma_c : e' \Downarrow^{\theta} \Delta_c : [\![v']\!]_{cc_v}$$

in P_{cost} and vice versa, where e' is either a logical variable (according to Γ_c) or $e' = [\![e]\!]_{cc_e}$, and $v = v'$ (if they are variables) or $v = c(\overline{x_n})$ and $v' = c(cc, \overline{x_n})$.

We prove the claim by induction on the depth m of the derivation.
(Base case $m = 0$) In this case, only two rules can be applied:

- If e is a logical variable (i.e., a variable x such that $\Gamma[x] = x$) then, by rule Val of the cost-augmented semantics, it is $(cc_e, \Gamma : x \Downarrow_{\{\}} \Gamma : x, cc_e)$ in the original program P. Since e is a logical variable, $e' = e$ and, by rule Val of the standard semantics, we also have $(\Gamma_c : x \Downarrow^{\{\}} \Gamma_c : x)$ in P_{cost}.

  Similarly, if $e = c(\overline{x_n})$, only rule Val can be applied: $(cc_e, \Gamma : c(\overline{x_n}) \Downarrow_{\{\}} \Gamma : c(\overline{x_n}), cc_e)$. Thus, $e' = [\![c(\overline{x_n})]\!]_{cc_e} = c(cc_e, \overline{x_n})$ and, by rule Val of the standard semantics, we also have $(\Gamma_c : c(cc_e, \overline{x_n}) \Downarrow^{\{\}} \Gamma_c : c(cc_e, \overline{x_n}))$.

- Otherwise, $e = x$ is a variable with $\Gamma = \Gamma[x \overset{cc_c}{\longmapsto} c(\overline{x_n})]$ and rule VarCons of the cost-augmented semantics is applied in program P:

  $$cc_e, \Gamma[x \overset{cc_c}{\longmapsto} c(\overline{x_n})] : x \Downarrow_{\{cc_e \leftarrow V\}} \Gamma[x \overset{cc_c}{\longmapsto} c(\overline{x_n})] : c(\overline{x_n}), cc_c$$

  Therefore, in the instrumented program, P_{cost}, it is $\Gamma_c[x \mapsto [\![c(\overline{x_n})]\!]_{cc_c}] : [\![x]\!]_{cc_e} = \Gamma_c[x \mapsto c(cc_c, \overline{x_n})] : V_{cc_e}(x)$ and, thus, the following derivation holds by applying rules Fun (to unfold V_{cc_e}) and VarCons:

  $$\frac{\Gamma_c[x \mapsto c(cc_c, \overline{x_n})] : x \Downarrow^{\{\}} \Gamma_c[x \mapsto c(cc_c, \overline{x_n})] : c(cc_c, \overline{x_n})}{\Gamma_c[x \mapsto c(cc_c, \overline{x_n})] : V_{cc_e}(x) \Downarrow^{\{cc_e \leftarrow V\}} \Gamma_c[x \mapsto c(cc_c, \overline{x_n})] : c(cc_c, \overline{x_n})}$$

(Inductive case $m > 1$) In this case, rules VarExp, Fun, Let, Or, Select, Guess, and SCC are applicable. We consider each case separately.

(VarExp) Then, e is a variable with $\Gamma[x] \neq x$ and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{cc_{e_1}, \Gamma : e_1 \ \Downarrow_\theta \ \Delta : v, cc_v}{cc_e, \Gamma[x \overset{cc_{e_1}}{\longmapsto} e_1] : x \ \Downarrow_{\{cc_e \leftarrow V\}+\theta+\{cc_v \leftarrow U\}} \ \Delta[x \overset{cc_v}{\longmapsto} v] : v, cc_v}$$

In the instrumented program, P_{cost}, it is $e' = [\![x]\!]_{cc_e} = V_{cc_e}(x)$ (since x is not a logical variable) and $[\![e_1]\!]_{cc_{e_1}} = \mathit{update}([\![e_1]\!]_{cc_{e_1}})$ (since e_1 is neither x nor a constructor-rooted term). Therefore, the following derivation holds in P_{cost} by applying rules Fun and VarExp:

$$\dfrac{\dfrac{\dfrac{\Gamma_c[x \mapsto \mathit{update}([\![e_1]\!]_{cc_{e_1}})] : [\![e_1]\!]_{cc_{e_1}} \ \Downarrow^\theta \ \Delta_c[x \mapsto [\![v]\!]_{cc_v}] : [\![v]\!]_{cc_v}}{\Gamma_c[x \mapsto \mathit{update}([\![e_1]\!]_{cc_{e_1}})] : \mathit{update}([\![e_1]\!]_{cc_{e_1}}) \ \Downarrow^{\theta+\{cc_v \leftarrow U\}} \ \Delta_c[x \mapsto [\![v]\!]_{cc_v}] : [\![v]\!]_{cc_v}}}{\Gamma_c[x \mapsto \mathit{update}([\![e_1]\!]_{cc_{e_1}})] : x \ \Downarrow^{\theta+\{cc_v \leftarrow U\}} \ \Delta_c[x \mapsto [\![v]\!]_{cc_v}] : [\![v]\!]_{cc_v}}}{\Gamma_c[x \mapsto \mathit{update}([\![e_1]\!]_{cc_{e_1}})] : V_{cc_e}(x) \ \Downarrow^{\{cc_e \leftarrow V\}+\theta+\{cc_v \leftarrow U\}} \ \Delta_c[x \mapsto [\![v]\!]_{cc_v}] : [\![v]\!]_{cc_v}}$$

Finally, the topmost judgement follows by the inductive hypothesis.

(Fun) Then, e is a function call and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{cc_e, \Gamma : \rho(e_1) \ \Downarrow_\theta \ \Delta : v, cc_v}{cc_e, \Gamma : f(\overline{x_n}) \ \Downarrow_{\{cc_e \leftarrow F\}+\theta} \ \Delta : v, cc_v}$$

where $f(\overline{y_n}) = e_1 \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$. Therefore, $e' = [\![f(\overline{x_n})]\!]_{cc_e} = f_{cc_e}(\overline{x_n})$ and the following subderivation holds in P_{cost} by applying twice rule Fun of the standard semantics:

$$\frac{\dfrac{\Gamma_c : [\![\rho(e_1)]\!]_{cc_e} \ \Downarrow^\theta \ \Delta_c : [\![v]\!]_{cc_v}}{\Gamma_c : F_{cc_e}(\rho([\![e_1]\!]_{cc_e})) \ \Downarrow^{\{cc_e \leftarrow F\}+\theta} \ \Delta_c : [\![v]\!]_{cc_v}}}{\Gamma_c : f_{cc_e}(\overline{x_n}) \ \Downarrow^{\{cc_e \leftarrow F\}+\theta} \ \Delta_c : [\![v]\!]_{cc_v}}$$

since, by definition of cost transformation, $f_{cc}(\overline{y_n}) = F_{cc}([\![e_1]\!]_{cc}) \in P_{cost}$. Finally, the claim follows by applying the inductive hypothesis to the topmost judgement.

(Let) Then, $e = (\mathit{let}\ x = e_1\ \mathit{in}\ e_2)$ is a let expression and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{cc_e, \Gamma[y \overset{cc_e}{\longmapsto} \rho(e_1)] : \rho(e_2) \ \Downarrow_\theta \ \Delta : v, cc_v}{cc_e, \Gamma : \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \ \Downarrow_{\{cc_e \leftarrow H\}+\theta} \ \Delta : v, cc_v}$$

where $\rho = \{x \mapsto y\}$. Now, we consider three cases, depending on whether $e_1 = x$ (a logical variable), $e_1 = c(\overline{x_n})$, or e_1 is any other expression:

- If $e_1 = x$, then $e' = [\![let\ x = e_1\ in\ e_2]\!]_{cc_e} = H_{cc_e}(let\ x = x\ in\ [\![e_2]\!]_{cc_e})$ and the following subderivation holds in P_{cost} by applying rules Fun and Let of the standard semantics:

$$\frac{\dfrac{\Gamma_c[y \mapsto y] : [\![\rho(e_2)]\!]_{cc_e} \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}{\Gamma_c : let\ x = x\ in\ [\![e_2]\!]_{cc_e}) \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}}{\Gamma_c : H_{cc_e}(let\ x = x\ in\ [\![e_2]\!]_{cc_e}) \ \Downarrow^{\{cc_e \leftarrow H\}+\theta}\ \Delta_c : [\![v]\!]_{cc_v}}$$

  and the claim follows by applying the inductive hypothesis to the topmost judgement.

- If $e_1 = c(\overline{x_n})$, then

$$e' = [\![let\ x = e_1\ in\ e_2]\!]_{cc_e} = H_{cc_e}(let\ x = [\![e_1]\!]_{cc_e}\ in\ [\![e_2]\!]_{cc_e})$$

  and the following subderivation holds in P_{cost} by applying rules Fun and Let of the standard semantics:

$$\frac{\dfrac{\Gamma_c[y \mapsto \rho([\![e_1]\!]_{cc_e})] : [\![\rho(e_2)]\!]_{cc_e} \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}{\Gamma_c : let\ x = [\![e_1]\!]_{cc_e}\ in\ [\![e_2]\!]_{cc_e}) \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}}{\Gamma_c : H_{cc_e}(let\ x = [\![e_1]\!]_{cc_e}\ in\ [\![e_2]\!]_{cc_e}) \ \Downarrow^{\{cc_e \leftarrow H\}+\theta}\ \Delta_c : [\![v]\!]_{cc_v}}$$

  and the claim follows by applying the inductive hypothesis to the topmost judgement.

- Finally, if e_1 is any other expression, then $e' = [\![let\ x = e_1\ in\ e_2]\!]_{cc_e} = H_{cc_e}(let\ x = update([\![e_1]\!]_{cc_e})\ in\ [\![e_2]\!]_{cc_e})$ and the following subderivation holds in P_{cost} by applying rules Fun and Let of the standard semantics:

$$\frac{\dfrac{\Gamma_c[y \mapsto \rho(update([\![e_1]\!]_{cc_e}))] : [\![\rho(e_2)]\!]_{cc_e} \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}{\Gamma_c : let\ x = update([\![e_1]\!]_{cc_e})\ in\ [\![e_2]\!]_{cc_e}) \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}}{\Gamma_c : H_{cc_e}(let\ x = update([\![e_1]\!]_{cc_e})\ in\ [\![e_2]\!]_{cc_e}) \ \Downarrow^{\{cc_e \leftarrow H\}+\theta}\ \Delta_c : [\![v]\!]_{cc_v}}$$

  and the claim follows by applying the inductive hypothesis to the topmost judgement.

(Or) Then, e is a disjunction and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{cc_e, \Gamma : e_i \ \Downarrow_{\theta}\ \Delta : v, cc_v}{cc_e, \Gamma : e_1\ or\ e_2 \ \Downarrow_{\{cc_e \leftarrow N\}+\theta}\ \Delta : v, cc_v}$$

where $i \in \{1, 2\}$. Therefore, $e' = [\![e_1\ or\ e_2]\!]_{cc_e} = N_{cc_e}([\![e_1]\!]_{cc_e}\ or\ [\![e_2]\!]_{cc_e})$ and the following subderivation holds in P_{cost} by applying rules Fun and Or of the standard semantics:

$$\frac{\dfrac{\Gamma_c : [\![e_i]\!]_{cc_e} \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}{\Gamma_c : [\![e_1]\!]_{cc_e}\ or\ [\![e_2]\!]_{cc_e} \ \Downarrow^{\theta}\ \Delta_c : [\![v]\!]_{cc_v}}}{\Gamma_c : N_{cc_e}([\![e_1]\!]_{cc_e}\ or\ [\![e_2]\!]_{cc_e}) \ \Downarrow^{\{cc_e \leftarrow N\}+\theta}\ \Delta_c : [\![v]\!]_{cc_v}}$$

and the claim follows by applying the inductive hypothesis to the topmost judgement.

(Select) Then, $e = (f)\,case\ x\ of\ \{\overline{p_k \to e_k}\}$ is a case expression and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{cc_e, \Gamma : x \ \Downarrow_{\theta_1}\ \Delta : c(\overline{y_n}), cc_c \qquad cc_e, \Delta : \rho(e_i) \ \Downarrow_{\theta_2}\ \Theta : v, cc_v}{cc_e, \Gamma : (f)\,case\ x\ of\ \{\overline{p_k \to e_k}\} \ \Downarrow_{\theta_1 + \{cc_e \leftarrow C\} + \theta_2}\ \Theta : v, cc_v}$$

where $p_i = c(\overline{x_n})$, $i \in \{1, \ldots, n\}$, and $\rho = \{\overline{x_n \mapsto y_n}\}$. Therefore, since x evaluates to a non-variable expression, $e' = [\![(f)\,case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]_{cc_e} = (f)\,case\ [\![x]\!]_{cc_e}\ of\ \{\overline{p'_k \to C_{cc_e}([\![e_k]\!]_{cc_e})}\}$ (we omit the evaluation of $isVar(x)$ for clarity), where $p'_i = c(cc, \overline{z_l})$ for all $p_i = c(\overline{z_l})$, $i = 1, \ldots, k$. Then, the following subderivation holds in P_{cost} by applying rules Select and Fun of the standard semantics:

$$\frac{\dfrac{}{\Gamma_c : [\![x]\!]_{cc_e} \ \Downarrow^{\theta_1}\ \Delta_c : [\![c(\overline{y_n})]\!]_{cc_c}} \quad \dfrac{\Delta_c : [\![\rho(e_i)]\!]_{cc_e} \ \Downarrow^{\theta_2}\ \Theta : [\![v]\!]_{cc_v}}{\Delta_c : C_{cc_e}([\![\rho(e_i)]\!]_{cc_e}) \ \Downarrow^{\{cc_e \leftarrow C\} + \theta_2}\ \Theta : [\![v]\!]_{cc_v}}}{\Gamma_c : (f)\,case\ [\![x]\!]_{cc_e}\ of\ \{\overline{p'_k \to C_{cc_e}(e_k)}\} \ \Downarrow^{\theta_1 + \{cc_e \leftarrow C\} + \theta_2}\ \Theta_c : [\![v]\!]_{cc_v}}$$

and the claim follows by applying the inductive hypothesis to the topmost judgements.

(Guess) Then, $e = fcase\ x\ of\ \{\overline{p_k \to e_k}\}$ is a flexible case expression and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{cc_e, \Gamma : x \ \Downarrow_{\theta_1}\ \Delta : y, cc_y \qquad cc_e, \Delta[y \overset{cc_e}{\mapsto} \rho(p_i), \overline{y_n \overset{cc_e}{\mapsto} y_n}] : \rho(e_i) \ \Downarrow_{\theta_2}\ \Theta : v, cc_v}{cc_e, \Gamma : fcase\ x\ of\ \{\overline{p_k \to e_k}\} \ \Downarrow_{\theta_1 + \{cc_e \leftarrow V, cc_e \leftarrow U, cc_e \leftarrow B, cc_e \leftarrow n*H\} + \theta_N + \theta_2}\ \Theta : v, cc_v}$$

where $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$, $\overline{y_n}$ are fresh variables, and $\theta_N = \{cc_e \leftarrow N\}$ if $k > 1$ and $\theta_N = \{\,\}$ if $k = 1$. Therefore, since x evaluates to a logical variable, it is $e' = [\![fcase\ x\ of\ \{\overline{p_k \to e_k}\}]\!]_{cc_e} =$

$V_{cc_e}(U_{cc_e}(B_{cc_e}(\theta_{N_{cc_e}}((f)\,case\ [\![x]\!]_{cc_e}\ of\ \{\overline{p'_k \to |p_k| * H_{cc_e}([\![e_k]\!]_{cc_e})}\}))))$

(we omit the evaluation of $isVar(x)$ for clarity), where $p'_i = c(cc, \overline{z_l})$ for all $p_i = c(\overline{z_l})$, $i = 1, \ldots, k$. Then, the following subderivation holds in P_{cost} by applying rules Guess and Fun of the standard semantics:

$$\frac{\Gamma_c : [\![x]\!]_{cc_e} \ \Downarrow^{\{cc_e \leftarrow V\} + \theta_1}\ \Delta_c : [\![y]\!]_{cc_y} \qquad \begin{array}{c} \Delta_c : [\![\rho(e_i)]\!]_{cc_e} \ \Downarrow^{\theta_2}\ \Theta : [\![v]\!]_{cc_v} \\ \hline \Delta_c[y \mapsto [\![\rho(p_i)]\!]_{cc_e}, \overline{y_n \mapsto [\![y_n]\!]_{cc_e}}] : \\ n * H_{cc_e}([\![\rho(e_i)]\!]_{cc_e}) \\ \Downarrow^{\{cc_e \leftarrow C, cc_e \leftarrow n*H\} + \theta_2}\ \Theta_c : [\![v]\!]_{cc_v} \end{array}}{\begin{array}{l} \Gamma_c : U_{cc_e}(B_{cc_e}(\theta_{N_{cc_e}}((f)\,case\ [\![x]\!]_{cc_e}\ of\ \{\overline{p'_k \to |p_k| * H_{cc_e}([\![e_k]\!]_{cc_e})}\}))) \\ \Downarrow^{\theta_1 + \{cc_e \leftarrow V, cc_e \leftarrow U, cc_e \leftarrow B, cc_e \leftarrow |p_k| * H\} + \theta_N + \theta_2}\ \Theta_c : [\![v]\!]_{cc_v} \end{array}}$$

and the claim follows by applying the inductive hypothesis to the topmost judgements.

(SCC) Then, $e = scc(cc, e_1)$ and we performed the following derivation step in P (with the cost-augmented semantics):

$$\frac{cc, \Gamma : e_1 \;\Downarrow_\theta\; \Delta : v, cc_v}{cc_e, \Gamma : scc(cc, e_1) \;\Downarrow_{\{cc \leftarrow E\}+\theta}\; \Delta : v, cc_v}$$

Therefore, $e' = [\![scc(cc, e_1)]\!]_{cc_e} = E_{cc}([\![e_1]\!]_{cc}$ and the following subderivation holds in P_{cost} by applying rules Fun and SCC of the standard semantics:

$$\frac{\Gamma_c : [\![e_1]\!]_{cc} \;\Downarrow^\theta\; \Delta_c : [\![v]\!]_{cc_v}}{\Gamma_c : E_{cc}([\![e_1]\!]_{cc}) \;\Downarrow^{\{cc \leftarrow E\}+\theta}\; \Delta_c : [\![v]\!]_{cc_v}}$$

The claim follows by applying the inductive hypothesis to the topmost judgement.

□

As an alternative to the transformation presented in this section, we could also instrument programs by *partial evaluation* [Jones *et al.*, 1993], i.e., the partial evaluation of the cost semantics of Section 4.3 w.r.t. a source program P should return an instrumented program which is equivalent to P_{cost}. However, this approach requires both a implementation of the instrumented semantics as well as an optimal partial evaluator for the considered language (in order to obtain a reasonable instrumented program, rather than a slight specialization of the cost semantics). Thus, we preferred to introduce a direct transformation.

Now, we illustrate our cost transformation by means of a simple example. Consider, for instance, the following definition for function `len`:

```
len(x) = fcase x of { Nil → Z;
                      Cons(y,ys) → let w = scc("b",len(ys))
                                   in S(w) }
```

Then, the corresponding instrumented definition for the cost center `"a"` is the following:

```
len_a(x) =
      F_a(if isVar(x)
         then V_a(U_a(B_a(N_a(fcase V_a(x) of
            { Nil(cc) → Z(a);
              Cons(cc,y,ys) → H_a(H_a(H_a(let w = update(len_b(ys))
                                          in S(a,w))))} ))))
          else fcase V_a(x) of
```

```
{ Nil(cc) → Z(a);
  Cons(cc,y,ys) → C_a(H_a(let w = update(len_b(ys))
                          in S(a,w)))})
```

where the auxiliary function *update* is defined as follows:

```
update(x) = case x of { Z(cc) → U_cc(Z(cc));
                        S(cc,x) → U_cc(S(cc,x)) }
```

It is important to note that the transformation introduced in this chapter is different from the one presented in the previous chapter. Although, from a operational point of view, the transformation of this chapter can be seen as a generalization of the transformation in the previous chapter (i.e., The former extends the kind of symbolic costs computed, and introduces cost centers); the goal of the transformations is essentially different. On the one hand, the objective of the transformation in the previous chapter is to reason about costs of programs from a static point of view; that is, to generalize the costs of programs for any input data. On the other hand, the objective of the transformation in this chapter is to compute actual costs of programs for a particular execution.

4.5 Implementation

The main purpose of profiling programs is to increase runtime efficiency. However, in practice, it is important to obtain symbolic profiling information as well as measuring runtimes. As discussed before, we want to provide cost centers for both kinds of profiling in order to be able to analyze arbitrary sub-computations independently of the defined functions. For the formal introduction of costs and correctness proofs, symbolic costs are the appropriate means. Therefore, we introduced a program transformation dealing with symbolic costs. However, the presented program transformation can easily be extended for measuring runtimes and distribute them through cost centers. In this section, we first present our approach to measure runtimes and function calls (Section 4.5.1) and, then, describe the extensions to obtain symbolic profiling (Section 4.5.2).

4.5.1 Measuring Runtimes

When trying to measure actual runtimes, the crucial point is to alter the runtime behavior of the examined program *as little as possible*. If the program instrumented for profiling runs 50% slower or worse, one profiles the process of profiling rather than the program execution. Because of this, measuring actual runtimes is a matter of low-level programming and, thus, highly depending on the actual language implementation.

Our approach is specific to the Curry implementation PAKCS [Hanus *et al.*, 2006]. In this programming environment, Curry programs are compiled by transforming flat programs (cf. Figure 2.2) to SICStus Prolog (see [Antoy and Hanus, 2000] for details about this transformation). In order to provide low-level profiling for PAKCS, we instrument the program with the profiling mechanisms offered by SICStus Prolog. Fortunately, SICStus Prolog features low-level profiling instruments which create an overhead of approximately 22%. The Prolog tools provide precise measuring of the number of predicate and clause calls. For measuring runtime, a number of synthetic units is given which is computed according to Gorlick and Kesselman [1987].

The main challenge was to introduce the cost centers into Prolog profiling. Luckily, we found a way to do this *without* further slowing down the execution of the program being profiled. The only overhead we introduce is code duplication, since we introduce a different version of each function for each cost center, as in the program transformation described above. Thus, for the program

```
main = SCC "len" (length (SCC "list" (enumFromTo 1 10)))
```

function `main` does not call a function `length`. Instead, it calls a variant with the name "`length{len}`" and also a function named "`enumFromTo{list}`". Gathering all runtimes for functions with the attachment $\{cc\}$, one gets the runtime belonging to that cost center. An obvious optimization is to eliminate unreachable functions like `length{list}` in the example.

4.5.2 Extension for Symbolic Profiling

Our approach to symbolic profiling exactly represents the idea described in Section 4.4 above. For each cost, we introduce a new function, e.g., `var_lookup` for cost V. There are variations of these functions for the different cost centers, e.g., `var_lookup{list}` like in Section 4.5.1. After the execution of the transformed program, we simply count each call to `var_lookup{list}` to get the sum of costs V attributed to the cost center `list`.

The advantage of this method is its simplicity. The demands to use our transformation for profiling with any implementation of Curry are not very high. The runtime system must only be able to count the number of calls to a certain function which is easy to implement. The disadvantage is the considerable (but still practicable) slowdown as we are not only introducing new functions but also new function *calls*. Nevertheless, this overhead does not influence the computation of symbolic costs.

It is worthwhile to note that, although the program transformation of Figure 4.2 is equivalent to the cost semantics of Figure 4.1 for *particular computations* (as stated in Theorem 12), there is one important difference:

While the cost semantics is don't-care non-deterministic, the instrumented programs accumulate all costs according to the search strategy.

For instance, the cost for a failing derivation is also accumulated in the cost of the results computed afterwards. Furthermore, completely failing computations also have an associated cost while no proof tree (and thus no costs) can be constructed in the big-step semantics. From a practical point of view, this is an advantage of the program transformation over the cost semantics, since the cost of failing derivations is relevant in the presence of non-deterministic functions.

4.6 Related Work and Conclusions

The closest approaches to our work are [Sansom and Peyton-Jones, 1997] and [Albert and Vidal, 2002]. On the one hand, [Sansom and Peyton-Jones, 1997] presents a formal specification of the attribution of execution costs to cost centers by means of an appropriate cost-augmented semantics in the context of lazy functional programs. A significant difference from our work is that our flat representation of programs allows logical features (like non-determinism) and that we also present a formal transformation to instrument source programs. On the other hand, [Albert and Vidal, 2002] introduces a symbolic profiling scheme for functional logic languages. However, the approach of [Albert and Vidal, 2002] does not consider *sharing* (an essential component of lazy languages) and, thus, it is not an appropriate basis for the development of profiling tools for current implementations of lazy functional logic languages. Also, we introduced a program transformation that allows us to compute symbolic costs with a reasonable overhead. Finally, in the context of the PAKCS environment for Curry, we showed how actual runtimes can also be computed by reusing the SICStus Prolog profiler.

Part III

Program Slicing

Chapter 5

Dynamic Slicing Based on Redex Trails

Tracing computations is a widely used methodology for program debugging. Lazy languages, however, pose new demands on tracing techniques because following the *actual* trace of a computation is generally useless. Typically, tracers for lazy languages rely on the construction of a *redex trail*, a graph that stores the reductions performed in a computation. While tracing provides a significant help for locating bugs, the task still remains complex. A well-known debugging technique for imperative programs is based on *dynamic slicing*, a method to find the program statements that influence the computation of a value for a specific program input.

In this chapter, we introduce a novel technique for dynamic slicing in first-order lazy functional logic languages. Rather than starting from scratch, our technique relies on (a slight extension of) redex trails. We provide a notion of *minimal* dynamic slice and introduce a method to compute a complete dynamic slice from the redex trail of a computation. A clear advantage of our proposal is that one can enhance existing tracers with slicing capabilities with a modest implementation effort, since the same data structure (the redex trail) can be used for both tracing and slicing.

In addition, we propose a lightweight approach to program specialization in the context of lazy functional logic languages which is based on the dynamic slicing technique previously defined.

Part of the material of this chapter has been presented in the *ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation* (PEPM'04) celebrated in Verona (Italy) in 2004 [Ochoa *et al.*, 2004a]; in the *First Workshop on Software Analysis and Development for Pervasive Systems* (SONDA'04) celebrated in Verona (Italy) in 2004 [Ochoa *et al.*, 2004b]; in the *Workshop on Curry and Functional Logic Programming* (WCFLP 2005) celebrated in Tallinn (Estonia) in 2005 [Ochoa

et al., 2005]; and in the *10th European Conference on Logics in Artificial Intelligence* (JELIA'06) celebrated in Liverpool (England) in 2006 [Ochoa *et al.*, 2006].

5.1 Introduction

In lazy functional programming languages, following the *actual* trace of a computation is often useless due to the *on demand* style of evaluation. In the language Haskell [Peyton-Jones, 2003], this drawback has been overcome by introducing higher-level models that hide the details of lazy evaluation. This is the case of the following systems: Hood [Gill, 2000], Freja [Nilsson and Sparud, 1997], and Hat [Sparud and Runciman, 1997; Wallace *et al.*, 2001]. Some of these approaches are based on the construction of a *redex trail* [Sparud and Runciman, 1997], a directed graph which records copies of all values and redexes (*red*ucible *ex*pressions) of a computation. Wallace *et al.* [2001] introduced an extended trail, the *augmented redex trail* (ART), in order to cover all previous three approaches (i.e., those followed by Hood, Freja and Hat). Basically, redex trails allow us to present different—more intuitive—views of a given computation (e.g., by showing function calls with fully evaluated arguments).

Braßel *et al.* [2004b] have introduced an instrumented version of an operational semantics for the kernel of a lazy functional *logic* language like Curry [Hanus, 2006a] that returns, not only the computed values and bindings, but also a trail of the computation. Similarly to the ART approach, this trail can also be used to perform tracing and algorithmic debugging, as well as to observe data structures through a computation. In contrast to previous approaches, the correctness of the computed trail is formally proved, which amounts to say that it contains all (and only) the reductions performed in a computation.

While tracing-based debugging provides a powerful tool for inspecting erroneous computations, it should be clear that the task still remains complex. In this chapter we use a well-known debugging technique for imperative programs known as *program slicing* (see Chapter 1). When debugging functional programs, we usually start from a particular computation that outputs an incorrect value. In this situation, dynamic slicing may help the programmer in finding the location of the bug by extracting a program slice which only contains the sentences whose execution influenced the incorrect value. Therefore, we are mainly interested in dynamic slicing [Korel and Laski, 1988], which only reflects the actual dependences of the erroneous execution, thus producing smaller—more precise—slices than static slicing [Tip, 1995].

In this chapter, we present the first dynamic backward slicing technique for a lazy functional logic language like Curry (though it could also be applied to a purely functional language like Haskell). Rather than starting from scratch, our technique relies on a slight extension of redex trails. To be precise, we extend the redex trail

model of Braßel *et al.* [2004b] in order to also store the *location*, in the program, of each reduced expression. A dynamic slice is then computed by

1. first gathering the set of nodes—in the redex trail—which are *reachable* from the slicing criterion and, then,

2. by deleting (or hiding) those expressions of the original program whose location does not appear in the set of collected nodes (see the next section for an informal overview of our debugging technique).

From our point of view, dynamic slicing may provide a complementary—rather than alternative—approach to tracing-based debugging. A clear advantage of our approach is that one can enhance existing tracers with slicing capabilities with a modest implementation effort, since the same data structure—the redex trail—is used for both tracing and slicing.

The main contributions of this chapter can be summarized as follows:

- We introduce a flexible notion of *slicing criterion* for a first-order lazy functional (logic) language. Previous definitions, e.g., [Biswas, 1997a; Reps and Turnidge, 1996], only considered a projection of the entire program as a slicing criterion.

- We extend the instrumented semantics of Braßel *et al.* [2004b] in order to have an extended redex trail that also stores the positions, in the source program, of each reduced expression, which is essential for slicing.

- We provide a notion of minimal dynamic slice and introduce a method to compute a complete dynamic slice from the extended redex trail of a computation.

- We show how tracing—based on redex trails—and slicing can be combined into a single framework. Moreover, the viability of our approach is supported by a prototype implementation of an integrated debugging tool.

This chapter is organized as follows. In the next section, we present an informal overview of our approach to combine tracing and slicing into a single framework. Section 5.3 formalizes an instrumented semantics that builds an extended redex trail of a computation which also includes the program positions of expressions. Section 5.4 defines the main concepts involved in dynamic slicing and introduces a method to compute dynamic slices from the extended redex trail. Section 5.6 presents some implementation issues of a prototypical implementation which integrates both tracing and slicing. Finally, Section 5.7 includes a comparison to related work.

5.2 Combining Tracing and Slicing

In this section, we present an overview of a debugging tool for lazy functional languages that combines both tracing and dynamic slicing based on redex trails.

A *redex trail*—which was originally introduced by Sparud and Runciman [1997] in the context of lazy functional programs—is defined as a directed graph which records copies of all values and redexes (*red*ucible *ex*pressions) of a computation, with a backward link from each reduct (and its proper subexpressions) to the parent redex that created it. The ART (*Augmented Redex Trail*) model of the Haskell tracer Hat [Wallace *et al.*, 2001] mainly extends redex trails by also including forward links from every redex to its reduct. Braßel *et al.* [2004b] have introduced a data structure which shares many similarities with the ART model but also includes a special treatment to cope with non-determinism (i.e., disjunctions and flexible case structures) so that it can be used for functional *logic* languages. Furthermore, in contrast to previous approaches, the redex trails of Braßel *et al.* [2004b] are defined by instrumenting an operational semantics for the language (rather than by a program transformation) and their correctness is formally proved. Let us present this tracing technique by means of an example. The technique presented in this chapter is based on normalized flat programs. However, in this section, we consider that programs are not normalized for clarity.

Example 11 *Consider the program shown in Figure 5.1—inspired by a similar example by Liu and Stoller [2003]—where data structures are built from*

```
data Nat     = Z | S Nat
data Pairs   = Pair Nat Nat
data ListNat = Nil | Cons Nat ListNat
```

The execution of the program in Figure 5.1 should compute the maximum of the list [`Z`, `S Z`], i.e., `S Z`, and thus return `1`; however, it returns `0`.

In order to trace the execution of this program, the tracing tool of Braßel *et al.* [2004b] builds the redex trail shown in Figure 5.2. In this redex trail, we can distinguish two kinds of arrows:[1]

Successor arrows: There is a successor arrow, denoted by a solid arrow, from each redex to its reduct (e.g., from the node labeled with `main` to the node labeled with `printMax` in Figure 5.2).

Argument arrows: Arguments of both function and constructor calls are denoted by a pointer to the corresponding expression, which is denoted by a dashed arrow.

[1] The redex trails of Braßel *et al.* [2004b] also contain a third kind of arrows—the *parent* arrows—that we ignore in this chapter because they are not necessary for computing program slices.

```
main = printMax (minmax [Z, S Z])

printMin t = case t of { Pair x y → printNat x }
printMax t = case t of { Pair x y → printNat y }

printNat n = case n of { Z → 0; (S m) → 1 + printNat m }

fst t = case t of { Pair x y → x }
snd t = case t of { Pair x y → y }

minmax xs = case xs of
            { y:ys → case ys of
                       { []   → Pair y y;
                         z:zs → let m = minmax (z:zs)
                                in Pair (min y (fst m))
                                        (max y (snd m)) } }
min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

ite x y z = case x of { True → y; False → z }

leq x y = case x of
          { Z    → False;
            (S n) → case y of { Z → False; (S m) → leq n m } }
```

Figure 5.1: Example program `minmax`

When the evaluation of an argument is not required in a computation, we have a null pointer for that argument (e.g., the first argument of the node labeled with `Pair` in Figure 5.2).

From the computed trail, the user can inspect the trace of a computation. In our example, the tracing tool initially shows the following top-level computation for `main`:

```
0 = main
0 = printMax (Pair _ Z)
0 = printNat Z
0 = 0
```

where "`_`" denotes an argument whose evaluation is not needed (i.e., a null pointer in the redex trail). Each row shows a pair "*val* = *exp*" where *exp* is an expression and *val* is its (possibly partial) value. Note that this top-level computation corresponds to the six topmost nodes in Figure 5.2, where the two nodes labeled with a `case`

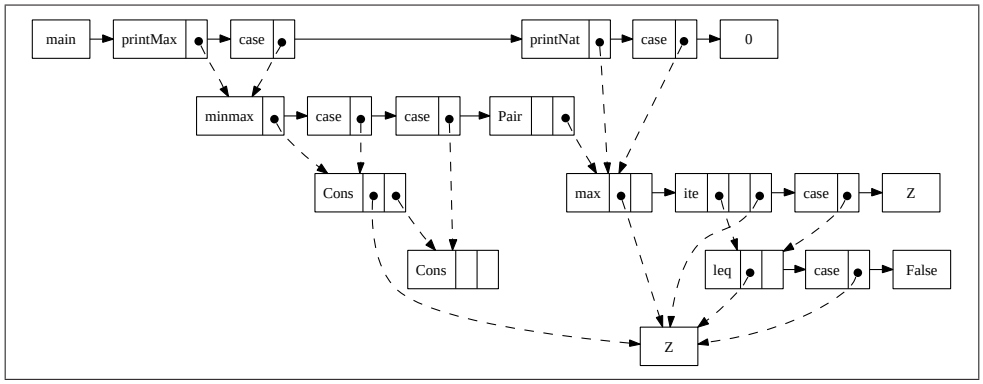


Figure 5.2: Redex trail for Example 11

structure are ignored to simplify the presentation. Note also that *function arguments appear as much evaluated as needed in the complete computation* in order to ease the understanding of the trace.

From the trace above, the user can easily see that the argument of `printMax` is incorrect because the second argument of constructor `Pair`, i.e., the maximum of the input list, should be (`S Z`) rather than `Z`. If the user selects the argument of `printMax`, the expression (`Pair _ Z`), the following subtrace is shown:

```
0         = printMax (Pair _ Z)
Pair _ Z = minmax   (Z : _ : _)
Pair _ Z = Pair     _           Z
```

Here, the user can conclude that the *evaluation* of `minmax` (rather than its *definition*!) contains a bug because it returns `Z` (the second element of `Pair`) as the maximum of a list headed by `Z`, ignoring the remaining elements of the list (which were not evaluated in the computation).

Now, the user can either try to figure out the location of the bug by computing further subtraces (which ones?) or by isolating the code fragment which is responsible of the wrong result by *dynamic slicing* and, then, inspecting the computed slice.

We propose the second approach and will show that incorporating dynamic slicing into existing tracers (based on redex-trails) is simple and powerful. Essentially, our approach allows the user to easily obtain the program slice associated with any entry $val = exp$ in the trace of a computation. Optionally, the user may provide a *pattern* π that indicates which part of val is of interest and which part is not. Then, a dynamic slice including all program constructs that are (potentially) needed to compute the relevant part of val—according to π—from exp is shown. Therefore, in our context, the slicing criterion is given by a tuple of the form $\langle exp, \pi \rangle$.

We consider that patterns for slicing are built from data constructor symbols, the

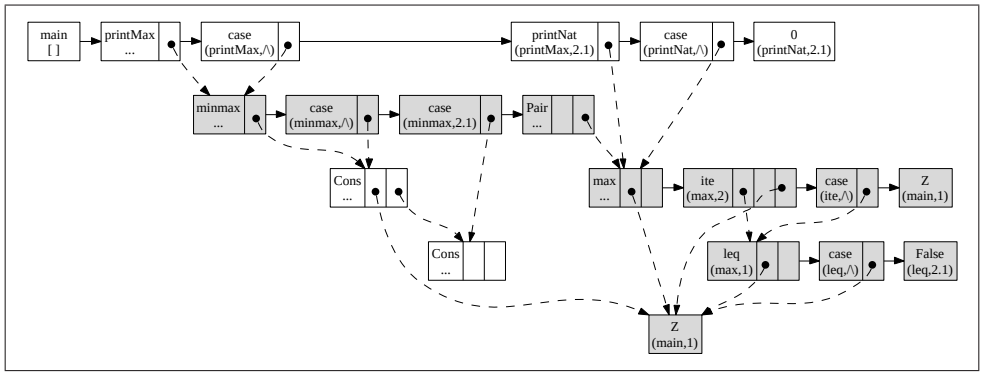


Figure 5.3: Extended redex trail for Example 11

special symbol $\bot$ (which denotes a subexpression of the value whose computation is not relevant), and $\top$ (a subexpression which is relevant). Consider, e.g., the following slicing criterion:

$$\langle \texttt{minmax}\ (\texttt{Z} : \_ : \_),\ \texttt{Pair}\ \bot\ \top \rangle$$

associated with the erroneous entry of the previous trace for `minmax`. Intuitively, it determines the extraction of a slice with the expressions that are involved in the computation of the second argument of `Pair` (starting from the call to `minmax`).

Similarly to Reps and Turnidge [1996], we consider the computation of *backward* slices where the information given by the slicing pattern π should be propagated backwards through the computation to determine the program constructs that are necessary to compute the relevant part of *exp* (according to π).

In order to compute the associated program slice, we extend the redex trail model in order to also store the position, in the source program, of every expression in the redex trail. Program positions are denoted by a pair (fun, pos) where *fun* is a function symbol and *pos* is a sequence of natural numbers that precisely identify a subexpression in the definition of *fun* (see Section 5.3.2 for a detailed definition).

For the example above, the extended redex trail including *some* of the program positions—more details on this example can be found in Section 5.4—is shown in Figure 5.3. The leftmost shadowed node stores the expression associated with the slicing criterion, while the remaining shadowed nodes store the expressions which are reachable from the slicing criterion (according to the slicing pattern).

The computed slice is thus the set of program positions in the shadowed nodes of the extended redex trail. The corresponding program slice is shown in Figure 5.4, where expressions that do not belong to the slice appear in gray. Some of these expressions (e.g., `let`) have been deleted after a process of pretty printing discussed in Section 5.5). By inspecting the slice, we can check that `minmax` only calls to

```
main = printMax (minmax [Z, S Z])

printMin t = case t of { Pair x y → printNat x }
printMax t = case t of { Pair x y → printNat y }

printNat n = case n of { Z → 0; (S m) → 1 + printNat m }

fst t = case t of { Pair x y → x }
snd t = case t of { Pair x y → y }

minmax xs = case xs of
            { y:ys → case ys of
                     { []   → Pair y y;
                       z:zs → let m = minmax (z:zs)
                              in Pair (min y (fst m))
                                      (max y (snd m)) } }
min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

ite x y z = case x of { True → y; False → z }

leq x y = case x of
          { Z    → False;
            (S n) → case y of { Z → False; (S m) → leq n m } }
```

Figure 5.4: Slice of program `minmax`

function `max` which, in turn, calls to functions `ite`, a standard conditional, and `leq` (for "less than or equal to"). However, the only case which is not deleted from the definition of `leq` can be read as "`Z` is *not* less than or equal to any natural number", which is clearly wrong.

Observe that the slice is particularly useful to understand the subtleties of lazy evaluation. For instance, function `minmax` is only called once, with no recursive call to inspect the tail of the input list (as a beginner to a lazy language would expect despite the error). This motivates the need for powerful tools for debugging and program understanding in lazy functional (logic) languages.

5.3 Building the Extended Trail

In this section, we extend the instrumented semantics of Braßel *et al.* [2004b] so that the computed redex trail also stores *program positions*; this additional information will become useful to identify precisely the location, in the source program, of each expression in the trail.

In order to keep the thesis self-contained and to ease the understanding, we introduce our instrumented semantics in a stepwise manner. Furthermore, we consider neither disjunctions nor flexible case structures in Sections 5.3.1, and 5.3.2; the treatment of these language features is delayed to Section 5.3.3.

5.3.1 A Semantics for Tracing

First, we present the instrumented semantics of Braßel *et al.* [2004b] which extends the operational semantics presented in Figure 2.6 to construct a redex trail of the computation. However, in contrast to [Braßel *et al.*, 2004b], we do not consider the "parent" relation because it is not useful for computing program slices. Basically, the trail is a directed graph with nodes identified by references[2] that are labeled with expressions. In the construction of this trail, the following conventions are adopted:

- $r \mapsto e$ denotes that the node with reference r is labeled with expression e. Successor arrows are denoted by $r \underset{q}{\mapsto}$ which means that node q is the successor of node r. The notation $r \underset{q}{\mapsto} e$ is used to denote both $r \mapsto e$ and $r \underset{q}{\mapsto}$.
- Argument arrows are denoted by $x \rightsquigarrow r$ which means that variable x points to node r. This relation suffices to store function arguments in our context because only variable arguments are allowed in normalized programs. These arrows are also called *variable pointers*.

Configurations in the instrumented semantics are tuples of the form $\langle \Gamma, e, S, G, r \rangle$; here, r denotes the *current* reference—where e will be eventually stored—and G is the trail built so far. Analogously to the heap, $G[r \mapsto e]$ is used to extend the graph G with a new node r labeled with the expression e.

The rules of the instrumented semantics are shown in Figure 5.5. Both rules, VarCons and VarExp, add a new variable pointer for x to the current graph if it does not yet contain such a pointer. This is used to account for the fact that the value of x is *needed* in the computation. For this purpose, the auxiliary function "$\bowtie$" is introduced:

$$G \bowtie (x \rightsquigarrow r) = \begin{cases} G[x \rightsquigarrow r] & \text{if there is no } r' \text{ such that } (x \rightsquigarrow r') \in G \\ G & \text{otherwise} \end{cases}$$

[2]The domain for references is not fixed. One can use, e.g., natural numbers but more complex domains are also possible.

(VarCons)	$\langle \Gamma[x \mapsto t], x, S, G, r\rangle$ $\Longrightarrow \langle \Gamma[x \mapsto t], t, S, G \bowtie (x \rightsquigarrow r), r\rangle$ where t is constructor-rooted
(VarExp)	$\langle \Gamma[x \mapsto e], x, S, G, r\rangle$ $\Longrightarrow \langle \Gamma[x \mapsto e], e, x : S, G \bowtie (x \rightsquigarrow r), r\rangle$ where e is not constructor-rooted and $e \neq x$
(Val)	$\langle \Gamma[x \mapsto e], v, x : S, G, r\rangle$ $\Longrightarrow \langle \Gamma[x \mapsto v], v, S, G, r\rangle$ where v is a value
(Fun)	$\langle \Gamma, f(\overline{x_n}), S, G, r\rangle$ $\Longrightarrow \langle \Gamma, \rho(e), S, G[r \underset{q}{\mapsto} f(\overline{x_n})], q\rangle$ where $f(\overline{y_n}) = e \in \mathcal{R}$, $\rho = \{\overline{y_n \mapsto x_n}\}$ and q is a fresh reference
(Let)	$\langle \Gamma, let\ x = e_1\ in\ e_2, S, G, r\rangle$ $\Longrightarrow \langle \Gamma[y \mapsto \rho(e_1)], \rho(e_2), S, G[r \underset{q}{\mapsto} \rho(let\ x = e_1\ in\ e_2)], q\rangle$ where $\rho = \{x \mapsto y\}$, y is a fresh variable and q is a fresh ref.
(Case)	$\langle \Gamma, case\ x\ of\ \{\overline{p_k \rightarrow e_k}\}, S, G, r\rangle$ $\Longrightarrow \langle \Gamma, x, (\{\overline{p_k \rightarrow e_k}\}, r) : S, G[r \mapsto case\ x\ of\ \{\overline{p_k \rightarrow e_k}\}], q\rangle$ where q is a fresh reference
(Select)	$\langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \rightarrow e_k}\}, r') : S, G, r\rangle$ $\Longrightarrow \langle \Gamma, \rho(e_i), S, G[r \mapsto c(\overline{y_n}), r' \underset{q}{\mapsto}], q\rangle$ where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$ and q is a fresh ref.

Figure 5.5: Instrumented semantics for tracing computations

Observe that this function is necessary to take care of variable sharing: if the value of a variable was already demanded in the computation, no new variable pointer should be added to the graph.

In rule Val, the current graph is not modified because computed values are only stored when they are *used* (e.g., in rule Select).

In rules Fun and Let a new node r, labeled with the current expression in the control, is added to the graph. The successor is set to q, a fresh reference, which is now the current reference of the derived configuration.

(Select-f)	$\langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \to e_k}\}, r') : S, G, r\rangle \Longrightarrow \langle \Gamma, Fail, S, G[r \mapsto c(\overline{y_n}), r' \underset{q}{\mapsto}], q\rangle$ where $\nexists i \in \{1, \ldots k\}$ such that $p_i = c(\overline{x_n})$
(Success)	$\langle \Gamma, c(\overline{x_n}), [\,], G, r\rangle \Longrightarrow \langle \Gamma, \diamond, [\,], G[r \mapsto c(\overline{x_n})], \Box\rangle$

Figure 5.6: Tracing semantics: failing and success rules

In rule Case we now push on the stack, not only the case alternatives, but also the current reference. This reference will become useful later if rule Select is eventually applied in order to set the right successor for the case expression (which is not yet known when rule Case is applied). Note that no successor for node r is set in rule Select because *values* are fully evaluated and, thus, have no successor.

When tracing computations, however, we are also interested in *failing* derivations. For this purpose, rule Select-f of Figure 5.6 is added to the calculus. This rule applies when there is no matching branch in a case expression. Furthermore, since expressions are added to the graph *a posteriori*, it does not suffice to reach a "final" configuration (i.e., a configuration with a value in the control and an empty stack). An additional rule Success (shown in Figure 5.6) is needed to store the result of the computation. In the derived configuration, we put the special symbol "$\diamond$" to denote that no further steps are possible.

In the instrumented semantics, a computation starts with a configuration of the form $\langle [\,], \texttt{main}, [\,], G_\varnothing, r\rangle$, where $G_\varnothing$ denotes an empty graph and r an initial reference.

Example 12 *Consider again the program of Example 9. By using the instrumented semantics of Figures 5.5 and 5.6, the derivation shown in Figure 5.7 is computed (here, natural numbers are used as references, where* `0` *is the initial reference). For clarity, in each configuration,* `G` *denotes the graph of the previous configuration.*

5.3.2 Including Program Positions

In this section we extend the previous semantics so that the computed redex trail also includes *program positions*, which are used to uniquely determine the location—in the source program—of each stored expression.

Definition 13 (position, program position) *A* position *is denoted by a sequence of natural numbers, where* Λ *is the empty sequence (i.e., the root position). They are*

$\langle [\,], \texttt{main}, [\,], G_\varnothing, \texttt{0}\rangle$
$\Longrightarrow_{\mathsf{Fun}}$
$\langle [\,],$ let x3 = Z in ..., $[\,],$ $\texttt{G}[\texttt{0} \underset{1}{\mapsto} \texttt{main}], 1\rangle$
$\Longrightarrow_{\mathsf{Let}}$
$\langle [\texttt{x3} \mapsto \texttt{Z}],$ let x1 = Z in ..., $[\,],$ $\texttt{G}[\texttt{1} \underset{2}{\mapsto}$ let x3 = Z in ...$], 2\rangle$
$\Longrightarrow_{\mathsf{Let}}$
$\langle [\texttt{x1} \mapsto \texttt{Z}, \ldots],$ let x2 = S x3 in ..., $[\,],$ $\texttt{G}[\texttt{2} \underset{3}{\mapsto}$ let x1 = Z in ...$], 3\rangle$
$\Longrightarrow_{\mathsf{Let}}$
$\langle [\texttt{x2} \mapsto \texttt{S x3}, \ldots],$ leq x1 x2, $[\,],$ $\texttt{G}[\texttt{3} \underset{4}{\mapsto}$ let x2 = S x3 in leq x1 x2$], 4\rangle$
$\Longrightarrow_{\mathsf{Fun}}$
$\langle [\ldots],$ case x1 of ..., $[\,],$ $\texttt{G}[\texttt{4} \underset{5}{\mapsto}$ leq x1 x2$], 5\rangle$
$\Longrightarrow_{\mathsf{Case}}$
$\langle [\texttt{x1} \mapsto \texttt{Z}, \ldots],$ x1, $[(\{\ldots\}, \texttt{5})],$ $\texttt{G}[\texttt{5} \mapsto$ case x1 of $\{\ldots\}], 6\rangle$
$\Longrightarrow_{\mathsf{VarCons}}$
$\langle [\texttt{x1} \mapsto \texttt{Z}, \ldots],$ Z, $[(\{\ldots\}, \texttt{5})],$ $\texttt{G}[\texttt{x1} \rightsquigarrow \texttt{6}], 6\rangle$
$\Longrightarrow_{\mathsf{Select}}$
$\langle [\ldots],$ True, $[\,],$ $\texttt{G}[\texttt{6} \mapsto \texttt{Z}, \texttt{5} \underset{7}{\mapsto}], 7\rangle$
$\Longrightarrow_{\mathsf{Success}}$
$\langle [\ldots],$ $\diamond$, $[\,],$ $\texttt{G}[\texttt{7} \mapsto$ True$], \Box\rangle$

Figure 5.7: Derivation of Example 12

used to address subexpressions of an expression viewed as a tree, as follows:[3]

$e\|_\Lambda = e$	*for all expression* e
$c(\overline{e_n})\|_{i.w} = e_i\|_w$	*if* $i \in \{1, \ldots, n\}$
$f(\overline{e_n})\|_{i.w} = e_i\|_w$	*if* $i \in \{1, \ldots, n\}$
$\mathit{let}\ x = e\ \mathit{in}\ e'\|_{1.w} = e\|_w$	
$\mathit{let}\ x = e\ \mathit{in}\ e'\|_{2.w} = e'\|_w$	
$\mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\}\|_{1.w} = e\|_w$	
$\mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\}\|_{2.i.w} = e_i\|_w$	*if* $i \in \{1, \ldots, n\}$

A program position *is a pair* (g, w) *that addresses the expression* $e|_w$ *in the right-hand side of the definition,* $g(\overline{x_n}) = e$*, of function* g*. Given a program* P*,* $\mathcal{P}os(P)$ *denotes the set of all program positions in* P*.*

Note that variables in a let construct or patterns in a case expression are not addressed by program positions because they will not be considered in the slicing process. While

[3] We consider arbitrary (not necessarily normalized) expressions for generality.

$\langle [\,], \texttt{main}, [\,], G_\varnothing, 0\rangle$

$\Longrightarrow_{\mathsf{Fun}}$	$\langle [\,],$	let x = Z in ...,	$[\,], \texttt{G}[0 \underset{1}{\mapsto} \texttt{main}],$	$1\rangle$
$\Longrightarrow_{\mathsf{Let}}$	$\langle [\texttt{x} \mapsto \texttt{Z}],$	let y = g x in f y,	$[\,], \texttt{G}[1 \underset{2}{\mapsto}$ let x = Z in ...$],$	$2\rangle$
$\Longrightarrow_{\mathsf{Let}}$	$\langle [\texttt{y} \mapsto \texttt{g x},\ \texttt{x} \mapsto \texttt{Z}],$	f y,	$[\,], \texttt{G}[2 \underset{3}{\mapsto}$ let y = g x in f y$],$	$3\rangle$
$\Longrightarrow_{\mathsf{Fun}}$	$\langle [\texttt{y} \mapsto \texttt{g x},\ \texttt{x} \mapsto \texttt{Z}],$	y,	$[\,], \texttt{G}[3 \underset{4}{\mapsto}$ f y$],$	$4\rangle$
$\Longrightarrow_{\mathsf{VarExp}}$	$\langle [\texttt{y} \mapsto \texttt{g x},\ \texttt{x} \mapsto \texttt{Z}],$	g x,	$[\texttt{y}], \texttt{G}[\texttt{y} \rightsquigarrow 4],$	$4\rangle$
$\Longrightarrow_{\mathsf{Fun}}$	$\langle [\texttt{y} \mapsto \texttt{g x},\ \texttt{x} \mapsto \texttt{Z}],$	x,	$[\texttt{y}], \texttt{G}[4 \underset{5}{\mapsto}$ g x$],$	$5\rangle$
$\Longrightarrow_{\mathsf{VarCons}}$	$\langle [\texttt{y} \mapsto \texttt{g x},\ \texttt{x} \mapsto \texttt{Z}],$	Z,	$[\texttt{y}], \texttt{G}[\texttt{x} \rightsquigarrow 5],$	$5\rangle$
$\Longrightarrow_{\mathsf{Val}}$	$\langle [\texttt{y} \mapsto \texttt{Z},\ \texttt{x} \mapsto \texttt{Z}],$	Z,	$[\,], \texttt{G},$	$5\rangle$
$\Longrightarrow_{\mathsf{Success}}$	$\langle [\texttt{y} \mapsto \texttt{Z},\ \texttt{x} \mapsto \texttt{Z}],$	$\diamond$,	$[\,], \texttt{G}[5 \mapsto \texttt{Z}],$	$\square\rangle$

Figure 5.8: Derivation of Example 13

other approaches to dynamic slicing, e.g., [Biswas, 1997b], consider some form of *explicit* labeling for each program expression, our program positions can be seen as a form of *implicit* labeling.

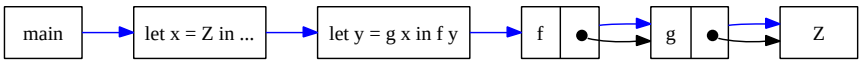
In order to add this additional information to the computed graph, two extensions are necessary. The first extension is straightforward: we label each mapping $x \mapsto e$ in the heap with the program position of expression e. Therefore, mappings in the heap have now the form $x \mapsto_{(g,w)} e$.

The second extension regards the addition of program positions to the nodes of the graph. In principle, we could extend the redex trail model so that $r \mapsto_{(g,w)}$ denotes that the expression which labels node r is located at program position (g, w). Unfortunately, this simple solution is not appropriate because *variables* are not stored in redex trails, but we still need to collect their program positions.

Example 13 *Consider the following simple program:*

```
main  = let x = Z in
          let y = g x in f y
f x = x
g x = x
```

By using the instrumented semantics of Figures 5.5 and 5.6, the derivation shown in Figure 5.8 is computed. The redex trail can be graphically depicted as follows:



Observe that the values of variables have not been stored in the graph. Therefore, the associated program slice containing the expressions in the redex trail would have the following form:

```
main  = let x = Z in
        let y = g x in f y
f x = x
g x = x
```

where functions `f` *and* `g` *appear in gray because no expression in their right-hand sides belong to the slice. This is clearly wrong because the right-hand sides of both functions,* `f` *and* `g`*, have been evaluated.*

In order to overcome this problem, we denote the mapping from nodes to program positions by $r \mapsto_{\mathcal{P}}$, where $\mathcal{P}$ is a *list* of program positions. Now, the meaning of $r \mapsto_{\mathcal{P}}$ is the following:

- the program position of the expression labeling node r is the first element of list $\mathcal{P}$ and
- if the tail of $\mathcal{P}$ is not empty, then it contains the program positions of the (chain of) variables whose evaluation was required in order to reach the expression labeling r.

Therefore, in the following, a configuration of the instrumented semantics is formally defined as follows:

Definition 14 (configuration) *A configuration of the semantics is represented with a tuple* $\langle \Gamma, e, S, G, r, \mathcal{P} \rangle$ *where* $\Gamma \in$ *Heap is the current heap,* $e \in$ *Exp is the expression to be evaluated (the control),* S *is the stack,* G *is a directed graph (the trail built so far),* r *is the current reference, and* $\mathcal{P}$ *is the list of program positions associated with* e*.*

The new instrumented semantics, including the rules for failure and success, is shown in Figure 5.9.[4]

In the first two rules, VarCons and VarExp, the evaluation of a variable x is demanded. Thus, the program position (g, w) of the expression to which x is bound in the heap is added to the current list of program positions $\mathcal{P}$. Here, $\mathcal{P}$ denotes a (possibly empty) list of program positions for the *chain* of variables that finishes in the current variable x (note that, in rule VarExp, e may be a variable). For this purpose, the auxiliary function "$\bowtie$" is redefined as follows:

$$G \bowtie_{\mathcal{P}} (x \rightsquigarrow r) = \begin{cases} G[x \rightsquigarrow r] & \text{if there is no } r' \text{ such that } (x \rightsquigarrow r') \in G \\ G[r \mapsto_{\mathcal{P} \cup \mathcal{P}'}] & \text{otherwise, with } (r \mapsto_{\mathcal{P}'}) \in G \end{cases}$$

The only difference with the original definition is that, when there is already a variable pointer $x \rightsquigarrow r$ in the graph, function $\bowtie$ should update the program positions associated

[4] As in Fig. 5.5, we assume that q is always a fresh reference.

(VarCons)	$\langle \Gamma[x \mapsto_{(g,w)} t], x, S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} t], t, S, G \bowtie (x \rightsquigarrow r), r, (g,w) : \mathcal{P}\rangle$ where t is constructor-rooted
(VarExp)	$\langle \Gamma[x \mapsto_{(g,w)} e], x, S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} e], e, x : S, G \bowtie (x \rightsquigarrow r), r, (g,w) : \mathcal{P}\rangle$ where e is not constructor-rooted and $e \neq x$
(Val)	$\langle \Gamma[x \mapsto_{(g',w')} e], v, x : S, G, r, (g,w) : \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} v], v, S, G, r, (g,w) : \mathcal{P}\rangle$ where v is a value
(Fun)	$\langle \Gamma, f(\overline{x_n}), S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma, \rho(e), S, G[r \underset{q}{\mapsto_{\mathcal{P}}} f(\overline{x_n})], q, [(f, \Lambda)]\rangle$ where $f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$
(Let)	$\langle \Gamma, let\ x = e_1\ in\ e_2, S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma[y \mapsto_{(g,w.1)} \rho(e_1)], \rho(e_2), S, G[r \underset{q}{\mapsto_{\mathcal{P}}} \rho(let\ x = e_1\ in\ e_2)], q, [(g, w.2)]\rangle$ where $\rho = \{x \mapsto y\}$, y is a fresh variable and $\mathcal{P} = (g,w) : \mathcal{P}'$ for some $\mathcal{P}'$
(Case)	$\langle \Gamma, case\ x\ of\ \{\overline{p_k \rightarrow e_k}\}, S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma, x, (\{\overline{p_k \rightarrow e_k}\}, r) : S, G[r \mapsto_{\mathcal{P}} case\ x\ of\ \{\overline{p_k \rightarrow e_k}\}], q, [(g, w.1)]\rangle$ where $\mathcal{P} = (g,w) : \mathcal{P}'$ for some $\mathcal{P}'$
(Select)	$\langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \rightarrow e_k}\}, r') : S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma, \rho(e_i), S, G[r \mapsto_{\mathcal{P}} c(\overline{y_n}),\ r' \underset{q}{\mapsto_{(g,w):\mathcal{P}'}}], q, [(g, w.2.i)]\rangle$ where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$ and $(r' \mapsto_{(g,w):\mathcal{P}'}) \in G$ for some $\mathcal{P}'$
(Select-f)	$\langle \Gamma, c(\overline{y_n}), (\{\overline{p_k \rightarrow e_k}\}, r') : S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma, Fail, S, G[r \mapsto_{\mathcal{P}} c(\overline{y_n}), r' \underset{q}{\mapsto_{[\,]}}], q, [\,]\rangle$ where $\nexists i \in \{1, \ldots k\}$ such that $p_i = c(\overline{x_n})$, with $(r' \mapsto_{[\,]}) \in G$
(Success)	$\langle \Gamma, c(\overline{x_n}), [\,], G, r, \mathcal{P}\rangle \Longrightarrow \langle \Gamma, \diamond, [\,], G[r \mapsto_{\mathcal{P}} c(\overline{x_n})], \square, [\,]\rangle$

Figure 5.9: Instrumented semantics for tracing and slicing

with reference r with the program positions associated with the current occurrence of x.

In rule Val, the first element in the current list of program positions (i.e., the program position of the expression in the control) is used to label the binding that updates the current heap.

In rules Fun, Let and Case, the current list of program positions is used to label the expression in the control which is stored in the graph. Note that, in rule Fun, the list of program positions in the derived configuration is reset to $[(f, \Lambda)]$ because the expression in the control is the complete right-hand side of function f. Note also that, in rule Let, the new binding $y \mapsto_{(g,w.1)} \rho(e_1)$ which is added to the heap is labeled with the program position of expression e_1. This information may be necessary in rules VarCons and VarExp if the value of y is demanded.

In rule Select, observe that the reference r'—which is stored in the stack by rule Case—is used to determine the initial list of program positions in the derived configuration. Finally, in rules Select-f and Success an empty list of program positions is set because the control of the derived configuration contains either the special (constructor) symbol *Fail* or $\diamond$, none of which occur in the program.

In order to perform computations with the instrumented semantics, we construct an *initial configuration* and (non-deterministically) apply the rules of Figure 5.9 until a *final configuration* is reached:

Definition 15 (initial configuration) *The initial configuration of computations has the form* $\langle$`[]`$, $`main`$, $`[]`$, G_\varnothing, r, [\,]\rangle$*, where* $G_\varnothing$ *denotes an empty graph,* r *a reference, and* $[\,]$ *an empty list of program positions (since there is no call to* `main` *from the right-hand side of any function definition).*

Definition 16 (final configuration) *The final configuration of computations has the form* $\langle \Gamma, \diamond, $`[]`$, G, \Box, [\,]\rangle$*, where* Γ *is a heap containing the computed bindings and* G *is the extended redex trail of the computation.*

We denote by $\Longrightarrow^*$ the reflexive and transitive closure of $\Longrightarrow$. A derivation $C \Longrightarrow^* C'$ is *complete* if C is an initial configuration and C' is a final configuration.

Clearly, the instrumented semantics is a conservative extension of the original small-step semantics of Figure 2.6, since the extensions impose no restriction on the application of the standard part of the semantics (the first three components of a configuration).

Example 14 *Consider again the program of Example 13. By using the instrumented semantics of Figure 5.9, the derivation shown in Figure 5.10 is now computed. The redex trail can be graphically depicted as follows:*

$\langle [\,], \texttt{main}, [\,], G_\varnothing, 0, [\,]\rangle$

$\Longrightarrow_{\mathsf{Fun}} \langle [\,],\ \texttt{let x = Z in } \ldots [\,],\ \texttt{G}[0 \overset{}{\underset{1}{\mapsto}}_{[\,]} \texttt{main}],\ 1,\ [(\texttt{main}, \Lambda)]\rangle$

$\Longrightarrow_{\mathsf{Let}} \langle [\texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],\ \texttt{let y = g x in f y},\ [\,],\ \texttt{G}[1 \overset{}{\underset{2}{\mapsto}}_{[(\texttt{main},\Lambda)]} \texttt{let x = Z in} \ldots],\ 2,\ [(\texttt{main}, 2)]\rangle$

$\Longrightarrow_{\mathsf{Let}} \langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x},\ \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],\ \texttt{f y},\ [\,],\ \texttt{G}[2 \overset{}{\underset{3}{\mapsto}}_{[(\texttt{main},2)]} \texttt{let y = g x in} \ldots],\ 3,\ [(\texttt{main}, 2.2)]\rangle$

$\Longrightarrow_{\mathsf{Fun}} \langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x},\ \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],\ \texttt{y},\ [\,],\ \texttt{G}[3 \overset{}{\underset{4}{\mapsto}}_{[(\texttt{main},2.2)]} \texttt{f y}],\ 4,\ [(\texttt{f}, \Lambda)]\rangle$

$\Longrightarrow_{\mathsf{VarExp}} \langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x},\ \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],\ \texttt{g x},\ [\texttt{y}],\ \texttt{G}[\texttt{y} \rightsquigarrow 4],\ 4,\ [(\texttt{main}, 2.1), (\texttt{f}, \Lambda)]\rangle$

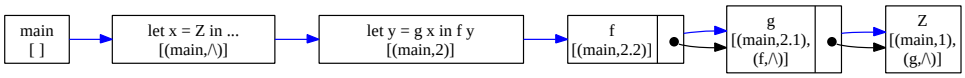
$\Longrightarrow_{\mathsf{Fun}} \langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x},\ \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],\ \texttt{x},\ [\texttt{y}],\ \texttt{G}[4 \overset{}{\underset{5}{\mapsto}}_{[(\texttt{main},2.1),(\texttt{f},\Lambda)]} \texttt{g x}],\ 5,\ [(\texttt{g}, \Lambda)]\rangle$

$\Longrightarrow_{\mathsf{VarCons}} \langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x},\ \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],\ \texttt{Z},\ [\texttt{y}],\ \texttt{G}[\texttt{x} \rightsquigarrow 5],\ 5,\ [(\texttt{main}, 1), (\texttt{g}, \Lambda)]\rangle$

$\Longrightarrow_{\mathsf{Val}} \langle [\texttt{y} \mapsto_{(\texttt{main},1)} \texttt{Z},\ \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],\ \texttt{Z},\ [\,],\ \texttt{G},\ 5,\ [(\texttt{main}, 1), (\texttt{g}, \Lambda)]\rangle$

$\Longrightarrow_{\mathsf{Success}} \langle [\texttt{y} \mapsto_{(\texttt{main},1)} \texttt{Z},\ \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]\ \diamond,\ [\,],\ \texttt{G}[5 \mapsto_{[(\texttt{main},1),(\texttt{g},\Lambda)]} \texttt{Z}],\ \square,\ [\,]\rangle$

Figure 5.10: Derivation of Example 14



Therefore, in contrast to the situation in Example 13, the associated slice is now the complete program.

The correctness of the redex trail has been proved by Braßel *et al.* [2004b]. Now, we state the correctness of the computed program positions (the proof can be found in the appendix). In what follows, given a configuration of the form $C = \langle \Gamma, e, S, G, r, \mathcal{P}\rangle$, we let $ctrl(C) = e$ and $pos(C) = \mathcal{P}$.

Theorem 17 *Let P be a program and $(C \Longrightarrow^* C')$ be a complete derivation in P, where G' is the graph in the final configuration C'. If $(r \mapsto_{\mathcal{P}} e) \in G'$, then either $\mathcal{P} = [\,]$ (with $e = \texttt{main}$) or $\mathcal{P} = [(g_k, w_k), \ldots, (g_1, k_1)]$, $k \geqslant 1$, and the following conditions hold:*

1. *there exists a subderivation $C_1 \Longrightarrow \ldots \Longrightarrow C_j$ of $C \Longrightarrow^* C'$, $j \geqslant k$, such that $pos(C_j) = [(g_k, w_k), \ldots, (g_1, w_1)]$, and*
2. *there exists a rule $g_i(\overline{x_{n_i}}) = e_i \in P$ such that $e_i|_{w_i} = ctrl(C_i)$, for all $i = 1, \ldots, j$, with $e_j|_{w_j} = e$.*

In order to prove Theorem 17 we need first the following auxiliary lemma that proves two useful invariants for the configurations in a derivation. In what follows, given a configuration of the form $C = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, we let $heap(C) = \Gamma$, $stack(C) = S$ and $graph(C) = G$.

Lemma 18 *Let P be a program and $(C_0 \Longrightarrow^* C_m)$, $m \geqslant 0$, a derivation (not necessarily complete) in P. For each configuration $C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, $0 \leqslant i \leqslant m$, the following conditions hold:*

(*I*1) *either $\mathcal{P} = [\,]$ (and $e = \texttt{main}$, $e = \diamond$, or $e = Fail$) or $\mathcal{P} = (g, w) : \mathcal{P}'$ for some $\mathcal{P}'$ and there exists a rule $g(\overline{x_n}) = e_1 \in P$ such that $e_1|_w = e$ (up to variable renaming), and*

(*I*2) *for all binding $x \mapsto_{(g',w')} e'$ in the heap Γ, there exists a rule $g'(\overline{x_k}) = e_2$ in P such that $e_2|_{w'} = e'$ (up to variable renaming).*

Proof. We prove the claim by induction on the length m of the considered computation.

Base case ($m = 0$) Then, the derivation contains only the initial configuration

$$\langle \texttt{[]}, \texttt{main}, \texttt{[]}, G_\varnothing, r, [\,] \rangle$$

and both invariants trivially hold.

Inductive case ($m > 0$) Let us consider an arbitrary derivation of the form

$$C_0 \Longrightarrow C_1 \Longrightarrow \cdots \Longrightarrow C_{m-1} \Longrightarrow C_m$$

By the inductive hypothesis, both (I1) and (I2) hold in C_{m-1}. Now, we prove that (I1) and (I2) also hold in C_m. If $pos(C_{m-1}) = [\,]$, by definition of the instrumented semantics, C_{m-1} can only be an initial configuration (with $ctrl(C_{m-1}) = \texttt{main}$) or a final one (with $ctrl(C_{m-1}) = \diamond$ or $ctrl(C_{m-1}) = Fail$). Moreover, in our case, C_{m-1} can only be an initial configuration since we know that there exists a successor configuration C_m. Therefore, only rule Fun can be applied to C_{m-1}. Here, invariant (I1) trivially holds in C_m since $pos(C_m) = [(\texttt{main}, \Lambda)]$ clearly addresses the right-hand side of function `main`. Invariant (I2) also holds because the heap is not modified.

Otherwise, assume that $C_{m-1} = \langle \Gamma, e, S, G, r, (g, w) : \mathcal{P} \rangle$. We consider the following cases depending on the applied rule:

(VarCons) Then $e = x$ is a variable. Consider that $\Gamma = \Gamma[x \mapsto_{(g',w')} t]$. It is immediate that invariant (I1) holds in the derived configuration $C_m = \langle \Gamma, t, S, G, r, (g', w') : (g, w) : \mathcal{P} \rangle$ because the invariant (I2) holds in C_{m-1}. Also, invariant (I2) trivially holds in C_m because the heap is not modified in the step.

(VarExp) Then $e = x$ is a variable and the proof is perfectly analogous to the previous case.

(Val) Then either e is a (logical) variable or a constructor-rooted term. In either case, invariant (I1) trivially holds in C_m because both the control and the program position are not changed. As for invariant (I2), since invariant (I1) holds in C_{m-1}, we know that program position (g, w) is a correct program position for v. Therefore, the update of the heap is correct and, since invariant (I2) already holds in C_{m-1}, it still holds in C_m.

(Fun) Then $e = f(\overline{x_n})$ is a function call. Since the heap is not modified, invariant (I2) trivially holds in C_m. Moreover, since the expression in the control is the renamed (by ρ) right-hand side of function f, the program position (f, Λ) is obviously correct (up to the variable renaming ρ) and invariant (I1) follows.

(Let) Then $e = (let\ x = e_1\ in\ e_2)$ is a let expression. Since invariant (I1) holds in C_{m-1}, by definition of position, $(g, w.2)$ correctly addresses the (renaming of) expression e_2 and, thus, invariant (I1) also holds in C_m. Similarly, $(g, w.1)$ correctly addresses the (renaming of) expression e_1 and, since this is the only modification on heap Γ and (I2) holds for C_{m-1}, (I2) also holds in C_m.

(Case) Then $e = case\ x\ of\ \{\overline{p_k \to e_k}\}$. Since the heap is not modified, invariant (I2) trivially holds in C_m. Moreover, by definition of position, $(g, w.1)$ correctly addresses the case argument and, thus, invariant (I1) also holds in C_m.

(Select) Then $e = c(\overline{x_n})$. Since the heap is not modified in C_m, invariant (I2) trivially holds. Moreover, it is easy to prove that the program position of the case expression that demanded the evaluation of $c(\overline{x_n})$ is (g, w') which is stored in the graph. Therefore, by definition of position, $(g, w'.2.i)$ correctly addresses the expression $\rho(e_i)$ in the i-th branch of the case expression (up to the variable renaming ρ).

(Select-f) Then $e = c(\overline{x_n})$. Since the heap is not modified in this step, invariant (I2) trivially holds in C_m. Moreover, since the list of program positions in C_m is $[\,]$, invariant (I1) also holds.

(Success) Then $e = c(\overline{x_n})$. This case follows trivially because the heap is not modified and the list of program positions in C_m is $[\,]$.

□

According to the instrumented semantics of Figure 5.9, not all expressions in the control of a configuration are stored in the graph. The following definition, slightly extended from [Braßel *et al.*, 2004b] to include program positions (and ignore parent references), formalizes when a configuration contains an expression that will be stored in the graph (in the next step of the computation).

Definition 19 *A configuration* $\langle \Gamma, e, S, G, r, \mathcal{P} \rangle$ *is* relevant *iff one of the following conditions hold:*

1. *e is a value and the stack S is not headed by a variable, or*
2. *e is a function call, a let expression, or a case expression.*

The following lemma—a simplified version of the analogous lemma in [Braßel *et al.*, 2004b] where logical features are not considered—states that, for each relevant configuration in a derivation, the associated information is stored in the graph.

Lemma 20 *Let* $(C_0 \Longrightarrow^* C_n)$, $n > 0$, *be a derivation. For each* $i \in \{0, \ldots, n\}$, *we have that* $C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$ *is a relevant configuration iff* $G_{i+1} = G[r \mapsto_{\mathcal{P}} e]$, *where* $G_{i+1} = graph(C_{i+1})$.

Finally, we prove Theorem 17:

Proof. If $\mathcal{P} = [\,]$, then the proof follows trivially by Lemma 18 since $(r \mapsto_{\mathcal{P}} \texttt{main})$ belongs to the final graph G' by Lemma 20.

Now, consider $\mathcal{P} = [(g_k, w_k), \ldots, (g_1, k_1)]$, $k \geqslant 1$. Since we have $(r \mapsto_{\mathcal{P}} e) \in G'$, by Lemma 20, there exists a relevant configuration $C_j = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$. If $\mathcal{P} = [(g_k, w_k)]$ contains only one program position, the proof follows straightforwardly by Lemma 18 (with $k = 1$). Otherwise, observe that only rules VarCons, VarExp and Val can be applied to a configuration so that either the list of program positions is not modified or new program positions are added to it. Therefore, there must be a subderivation of the form $C_1 \Longrightarrow \ldots \Longrightarrow C_j$ of $C \Longrightarrow^* C'$ with $j = k$ (if rule Val is never applied) or $j \geqslant k$ (otherwise) such that $pos(C_j) = [(g_k, w_k), \ldots, (g_1, k_1)]$, $k > 1$. By Lemma 18, we have that there exists a rule $g_i(\overline{x_{n_i}}) = e_i \in P$ such that $e_i|_{w_i} = ctrl(C_i)$, for all $i = 1, \ldots, j$, with $e_j|_{w_j} = e$, and the claim follows. □

5.3.3 Logical Features

In this section, we sketch the extension of the instrumented semantics to deal with the logical features of the language: disjunctions and flexible case structures. The new rules are shown in Figure 5.11.

Rule Or is used to non-deterministically choose one of the disjuncts. It is self-explanatory and proceeds analogously to rules Fun, Let or Case.

The original rule Case is now slightly modified to also accept flexible case expressions. In this case, a symbol f is stored in the stack preceding the case alternatives to indicate that the case expression is flexible.

In contrast to rule Select, rule Guess is used when we have a *logical variable* in the control and the alternatives of a flexible case expression on the stack. Then, this rule non-deterministically choose one alternative and continue with the evaluation of this branch. Since the name of logical variables is not relevant, the new node r is labeled with a special symbol *LogVar*. The successor of this node is then set to q which is labeled with the selected binding for the logical variable. The list of program

(Or)	$\langle \Gamma, e_1 \; or \; e_2, S, G, r, \mathcal{P}\rangle \Longrightarrow \langle \Gamma, e_i, S, G[r \underset{q}{\mapsto_{\mathcal{P}}} e_1 \; or \; e_2], q, [(g, w.i)]\rangle$ where $i \in \{1, 2\}$ and $\mathcal{P} = (g, w) : \mathcal{P}'$
(Fcase)	$\langle \Gamma, fcase \; x \; of \; \{\overline{p_k \to e_k}\}, S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma, x, (f\{\overline{p_k \to e_k}\}, r) : S, G[r \mapsto_{\mathcal{P}} (f)case \; x \; of \; \{\overline{p_k \to e_k}\}],$ $q, [(g, w.1)]\rangle$ where $\mathcal{P} = (g, w) : \mathcal{P}'$
(Guess)	$\langle \Gamma[y \mapsto_{(h,w')} y], y, (f\{\overline{p_k \to e_k}\}, r') : S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma[y \mapsto_{(\_,\_)} \rho(p_i), \overline{y_n \mapsto_{(\_,\_)} y_n}], \rho(e_i), S,$ $G[r \underset{q}{\mapsto_{\mathcal{P}}} LogVar, q \mapsto_{[\,]} \rho(p_i), y \rightsquigarrow r, r' \underset{s}{\mapsto_{(g,w):\mathcal{P}'}}], s, [(g, w.2.i)]\rangle$ where $i \in \{1, \ldots k\}$, $p_i = c(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$ and $\overline{y_n}$ are fresh variables
(Guess-f)	$\langle \Gamma[y \mapsto_{(g,w)} y], y, (f\{\overline{p_k \to e_k}\}, r') : S, G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma[y \mapsto_{(g,w)} y], Fail, S, G[r \mapsto_{\mathcal{P}} LogVar, y \rightsquigarrow r, r' \underset{s}{\mapsto_{[\,]}}], s, [\,]\rangle$
(Success-x)	$\langle \Gamma[x \mapsto_{(g,w)} x], x, [\,], G, r, \mathcal{P}\rangle$ $\Longrightarrow \langle \Gamma[x \mapsto_{(g,w)} x], \diamond, [\,], G[r \overset{p}{\mapsto}_{\mathcal{P}} LogVar, x \rightsquigarrow r], \Box, [\,]\rangle$

Figure 5.11: Instrumented semantics for tracing and slicing: logical features

positions for node q is empty because patterns—and their local variables—introduced by instantiation do not have an associated program position. Finally, observe that we use $(\_,\_)$ to denote a *null* program position in the heap.

Rule Guess-f applies when we have a logical variable in the control and (the alternatives of) a *rigid* case expression on top of the stack, i.e., when the computation *suspends*. Analogously to rule Select-f, an empty list of program positions is set in the derived configuration.

Finally, rule Success-x is the counterpart of rule Success when the computed value is a logical variable.

5.4 Computing the Slice

In this section, we formalize the concept of dynamic slice and introduce a method to compute it from the extended redex trail.

As in the previous section, we start by considering neither disjunctions nor flexible case structures; the treatment of these language features is delayed to Section 5.4.1.

We define a dynamic slice as a set of program positions:

Definition 21 (dynamic slice) *Let P be a program. A dynamic slice for P is any set $\mathcal{W}$ of program positions with $\mathcal{W} \subseteq \mathcal{P}os(P)$.*

Observe that we are not interested in producing *executable* slices but in computing the program positions of the expressions that influence the slicing criterion. This is enough for our purposes, i.e., for showing the original program with some distinguished expressions.

In the literature of dynamic slicing for imperative programs, the slicing criterion usually identifies a concrete point of the execution history, e.g., $\langle \mathtt{n} = \mathtt{2}, \mathtt{8}^1, \mathtt{x} \rangle$, where $\mathtt{n} = \mathtt{2}$ is the input for the program, $\mathtt{8}^1$ denotes the *first* occurrence of statement `8` in the execution history, and `x` is the variable we are interested in. In principle, it is not difficult to extend this notion of slicing criterion to a *strict* functional language. For instance, a slicing criterion could be given by a pair $\langle f(\overline{v_n}), \pi \rangle$, where $f(\overline{v_n})$ is a function call that occurs during the execution of the program—whose arguments are fully evaluated because we consider a strict language—and π is a restriction on the output of $f(\overline{v_n})$.

Unfortunately, extending this notion to a *lazy* functional language is not trivial. For instance, identifying a function call $f(\overline{e_n})$ in the actual execution trace is impractical because $\overline{e_n}$ are usually rather complex expressions. Luckily, if a tracing tool like the one presented in Section 5.2 is available—i.e., a tracing tool that shows function calls with arguments as much evaluated as needed in the complete computation—then we can still consider a slicing criterion of the form $\langle f(\overline{v_n}), \pi \rangle$. In this case, $f(\overline{v_n})$ should be understood as "a function call $f(\overline{e_n})$ whose arguments $\overline{e_n}$ are evaluated to $\overline{v_n}$ in the complete computation".

The connection with the tracing tool (as described in Section 5.2) should be clear: given a (sub)trace of the form

$$\begin{array}{lcl} val_1 & = & f_1(v_1^1, \ldots, v_m^1) \\ val_2 & = & f_2(v_1^2, \ldots, v_m^2) \\ \ldots & & \\ val_k & = & f_k(v_1^k, \ldots, v_m^k) \end{array}$$

a slicing criterion has the form $\langle f_i(v_1^i, \ldots, v_m^i), \pi \rangle$, where π denotes a restriction on val_i. Therefore, the user can easily produce a valid slicing criterion from the trace of a computation by only giving π (if any, since a default value $\top$ can also be used). Traces usually contain a special symbol, "_", to denote that some subexpression is missing because its evaluation was not required in the computation. Consequently, we will also accept expressions containing this special symbol (see below).

Definition 22 (slicing criterion) *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a com-*

plete derivation.[5] *A slicing criterion for P w.r.t. $(C_0 \Longrightarrow^* C_m)$ is a tuple $\langle f(\overline{pv_n}), \pi \rangle$ such that $f \in \mathcal{F}$ is a defined function symbol (arity $n \geqslant 0$), $pv_1, \ldots, pv_n \in PValue$ are partial values, and $\pi \in Pat$ is a slicing pattern. The domain PValue obeys the following syntax:*

$$pv \in PValue \ ::= \ \_ \mid x \mid c(\overline{pv_k})$$

where $c \in \mathcal{C}$ is a constructor symbol (arity $k \geqslant 0$) and "$\_$" is a special symbol to denote any non-evaluated expression.

The domain Pat of slicing patterns is defined as follows:

$$\pi \in Pat \ ::= \ \bot \mid \top \mid c(\overline{\pi_k})$$

where $c \in \mathcal{C}$ is a constructor symbol of arity $k \geqslant 0$, $\bot$ denotes a subexpression of the value whose computation is not relevant and $\top$ a subexpression which is relevant.[6]

Slicing patterns are particularly useful when we are only interested in *part* of the result of a function call. For instance, if we know that a function call should return a pair, and we are only interested in the first component of this pair, we could use the slicing pattern $(\top, \bot)$; if a function returns a list, and we are only interested in the first two elements of this list, we could use the slicing pattern $\top : (\top : \bot)$. Of course, if we are interested in the complete result, we could simply use the slicing pattern $\top$.

In order to compute a slice, we first need to identify the configuration associated with a slicing criterion (scc) because only the program positions in the next configurations are potential candidates for the slice:

$$C_1 \Longrightarrow C_2 \Longrightarrow \ldots \Longrightarrow \overbrace{\underbrace{C_i}_{\text{scc}} \Longrightarrow C_{i+1} \ldots \Longrightarrow C_n}^{\text{candidates for the slice}}$$

For this purpose, we need some auxiliary definitions. In what follows, given a configuration of the form $C = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, we let $heap(C) = \Gamma$, $stack(C) = S$, and $graph(C) = G$.

First, we introduce the notion of *lookup* configuration [Braßel *et al.*, 2004b], i.e., a configuration that demands the evaluation of a given variable for the first time in a computation.

Definition 23 (lookup configuration) *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m \geqslant 0$, be a derivation. A configuration C_i, $0 \leqslant i \leqslant m$, is a lookup configuration of $\mathcal{D}$ iff $ctrl(C_i) = x$ and there exists no configuration C_j, $j < i$, with $ctrl(C_j) = x$.*

[5] Here, and in the following, we use the notation $C_0 \Longrightarrow^* C_m$ as a shorthand to denote a derivation of m steps of the form $C_0 \Longrightarrow C_1 \Longrightarrow \ldots \Longrightarrow C_m$.

[6] Our patterns for slicing are similar to the *liveness patterns* used to perform dead code elimination by Liu and Stoller [2003].

When a lookup configuration $\langle \Gamma, x, S, G, r, \mathcal{P} \rangle$ appears in a computation, a variable pointer $x \rightsquigarrow r$ is added to the graph to point out that the value of x is *demanded* in the computation:

Definition 24 (demanded variable) *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m \geqslant 0$, be a derivation. We say that a variable x is demanded in $\mathcal{D}$ iff there exists a lookup configuration C_i, $0 \leqslant i \leqslant m$, such that $ctrl(C_i) = x$.*

The following function is useful to extract the partial value denoted by an expression. For this purpose, outer constructors are left untouched, non-demanded variables and operation-rooted terms are replaced by the special symbol "_" (i.e., "not a value"), and demanded variables are dereferenced.

Definition 25 *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation. Given an expression e, we denote the partial value of e in $\mathcal{D}$ by $val_{\mathcal{D}}(e)$, where function val is defined as follows:*

$$val_{\mathcal{D}}(e) = \begin{cases} val^v_{\mathcal{D}}(x) & \text{if } e = x \text{ and } x \text{ is demanded in } \mathcal{D} \\ c(\overline{val_{\mathcal{D}}(x_n)}) & \text{if } e = c(\overline{x_n}) \\ \_ & \text{otherwise} \end{cases}$$

The auxiliary function $val^v_{\mathcal{D}}$ is used to dereference a variable w.r.t. (the heap computed in) a derivation:

$$val^v_{\mathcal{D}}(x) \;=\; \begin{cases} val^v_{\mathcal{D}}(y) & \text{if } \Gamma[x] = y \text{ and } x \neq y \\ c(\overline{val_{\mathcal{D}}(x_n)}) & \text{if } \Gamma[x] = c(\overline{x_n}) \\ \_ & \text{otherwise} \end{cases}$$

where $\Gamma = heap(C_m)$ is the heap in the last configuration of $\mathcal{D}$.

Let us illustrate this function with some examples. Consider three derivations, $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$, in which the computed heaps in the last configurations are as follows:[7]

$$\begin{array}{lcl} \Gamma_1 & = & [\ \mathtt{x} \mapsto \mathtt{y}, \quad \mathtt{y} \mapsto (\mathtt{S\ n}), \quad \mathtt{n} \mapsto \mathtt{Z}\ ] \\ \Gamma_2 & = & [\ \mathtt{x} \mapsto \mathtt{y}, \quad \mathtt{y} \mapsto (\mathtt{S\ z}), \quad \mathtt{z} \mapsto \mathtt{Z}, \quad \mathtt{n} \mapsto \mathtt{Z}\ ] \\ \Gamma_3 & = & [\ \mathtt{m} \mapsto (\mathtt{f\ Z}), \quad \mathtt{y} \mapsto (\mathtt{S\ n}), \quad \mathtt{n} \mapsto \mathtt{Z}\ ] \end{array}$$

Here $\mathtt{Z}, \mathtt{S} \in \mathcal{C}$ are constructors, $\mathtt{f}$ is a defined function symbol, and $\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{n}, \mathtt{m}$ are variables. Assuming that variables $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are demanded in the considered computations and $\mathtt{m}, \mathtt{n}$ are not, we have:

$$\begin{array}{lcl} val_{\mathcal{D}_1}(\mathrm{C\ \ x\ \ (f\ y)}) & = & \mathrm{C\ \ (S\ \_)\ \ \_} \\ val_{\mathcal{D}_2}(\mathrm{C\ \ x\ \ n}) & = & \mathrm{C\ \ (S\ Z)\ \ \_} \\ val_{\mathcal{D}_3}(\mathrm{C\ \ m\ \ y}) & = & \mathrm{C\ \ \_\ \ (S\ \_)} \end{array}$$

We can now formally identify the configuration associated with a slicing criterion.

[7] Here we ignore the program positions that label the bindings in the heap since they are not relevant for computing the partial value.

Definition 26 (SC-configuration) *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation. Given a slicing criterion $\langle f(\overline{pv_n}), \pi\rangle$, the associated SC-configuration C_i, $0 \leqslant i \leqslant m$, must fulfill the conditions*[8]

- *$ctrl(C_i) = f(\overline{x_n})$ and*
- *$pv_i = val_{\mathcal{D}}(x_i)$ for all $i = 1, \ldots, n$.*

Intuitively speaking, the SC-configuration associated with a slicing criterion $\langle f(\overline{pv_n}), \pi\rangle$ is a configuration, C_i, whose control has the form $f(\overline{x_n})$ and the variables $\overline{x_n}$ are bound to the partial values $\overline{pv_n}$ in the considered derivation.

For instance, given the program and derivation of Example 14, the SC-configuration associated with the slicing criterion $\langle \texttt{f Z}, \top\rangle$ is the configuration

$$\begin{array}{lll} \langle \quad [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}], & \texttt{f y}, & [\,], \\ \quad \texttt{G}[2 \underset{3}{\mapsto}_{[(\texttt{main},2)]} \texttt{let y = g x in} \ldots], & 3, & [(\texttt{main}, 2.2)] \quad \rangle \end{array}$$

since its control is (`f y`) and the variable `y` is bound to `Z` in the heap $[\texttt{y} \mapsto_{(\texttt{main},1)} \texttt{Z}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$ of the final configuration of the derivation.

We also need the notion of *complete subderivation*:

Definition 27 (complete subderivation) *Let P be a program and $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$ a complete derivation in P. Let C_i, $0 \geqslant i \geqslant m$, be a configuration. The subderivation $C_i \Longrightarrow^* C_j$, $i \geqslant j \geqslant m$, is the complete subderivation for C_i in $\mathcal{D}$ iff the following conditions hold:*

1. *$ctrl(C_j) \in Value$,*
2. *$stack(C_i) = stack(C_j)$, and*
3. *$\nexists k.\ i < k < j$, $stack(C_i) = stack(C_k)$ and $ctrl(C_k) \in Value$.*

Consider, for instance, the derivation of Example 14 which is shown in Fig. 5.12. Here, the complete subderivation for the configuration

$$\langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}], \texttt{g x}, [\texttt{y}], \texttt{G}[\texttt{y} \rightsquigarrow 4], 4, [(\texttt{main}, 2.1), (\texttt{f}, \Lambda)]\rangle$$

is shown in a box (where graphs and program positions are ignored for conciseness).

Now, we can already introduce the notion of *minimal* dynamic slice:

Definition 28 (minimal dynamic slice, *mds*) *Let P be a program and let $\mathcal{D} = (C_0 \Longrightarrow^* C_n)$ be a complete derivation in P. Let $\langle f(\overline{pv_l}), \pi\rangle$ be a slicing criterion and C_i its associated SC-configuration. The minimal dynamic slice for P w.r.t. the*

[8] If there are several configurations that fulfill these conditions (e.g., if the same function is called several times in the computation with the same arguments), we choose the first of such configurations.

$\langle [\,], \texttt{main}, [\,], G_\varnothing, \texttt{0}, [\,]\rangle$

$\Longrightarrow_{\mathsf{Fun}}$	$\langle [\,],$	$\texttt{let x = Z in ...},$	$[\,],$	$\texttt{G}_1, \texttt{1}, \mathcal{P}_1\rangle$
$\Longrightarrow_{\mathsf{Let}}$	$\langle [\texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}],$	$\texttt{let y = g x in f y},$	$[\,],$	$\texttt{G}_2, \texttt{2}, \mathcal{P}_2\rangle$
$\Longrightarrow_{\mathsf{Let}}$	$\langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$	$\texttt{f y},$	$[\,],$	$\texttt{G}_3, \texttt{3}, \mathcal{P}_3\rangle$
$\Longrightarrow_{\mathsf{Fun}}$	$\langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$	$\texttt{y},$	$[\,],$	$\texttt{G}_4, \texttt{4}, \mathcal{P}_4\rangle$
$\Longrightarrow_{\mathsf{VarExp}}$	$\langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$	$\texttt{g x},$	$[\texttt{y}],$	$\texttt{G}_5, \texttt{4}, \mathcal{P}_5\rangle$
$\Longrightarrow_{\mathsf{Fun}}$	$\langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$	$\texttt{x},$	$[\texttt{y}],$	$\texttt{G}_6, \texttt{5}, \mathcal{P}_6\rangle$
$\Longrightarrow_{\mathsf{VarCons}}$	$\langle [\texttt{y} \mapsto_{(\texttt{main},2.1)} \texttt{g x}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$	$\texttt{Z},$	$[\texttt{y}],$	$\texttt{G}_7, \texttt{5}, \mathcal{P}_7\rangle$
$\Longrightarrow_{\mathsf{Val}}$	$\langle [\texttt{y} \mapsto_{(\texttt{main},1)} \texttt{Z}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$	$\texttt{Z},$	$[\,],$	$\texttt{G}_8, \texttt{5}, \mathcal{P}_8\rangle$
$\Longrightarrow_{\mathsf{Success}}$	$\langle [\texttt{y} \mapsto_{(\texttt{main},1)} \texttt{Z}, \texttt{x} \mapsto_{(\texttt{main},1)} \texttt{Z}]$	$\diamond,$	$[\,],$	$\texttt{G}_9, \square, \mathcal{P}_9\rangle$

Figure 5.12: A complete subderivation

slicing criterion $\langle f(\overline{pv_l}), \pi\rangle$ *is given by* $mds([C_i, \ldots, C_j], \pi, \{\,\}, [C_{j+1}, \ldots, C_n])$*, where* $C_i \Longrightarrow^* C_j$ *is the complete subderivation for* C_i *in* $\mathcal{D}$ *and the definition of the auxiliary function mds is shown in Fig. 5.13 (here, function hd returns the first element of a list).*

Let us now informally explain how a minimal dynamic slice is computed through a simple example.

Example 15 *Consider the program shown in Fig. 5.14. Given the slicing criterion* $\langle \texttt{g Z},\ \texttt{C} \perp \top\rangle$*, the computation described in Fig. 5.15 illustrates the way in which function mds proceeds. For simplicity, we only show the control of each configuration. Here, the computation for function* `main` *is shown in the leftmost column; then, each time a case expression demands the evaluation of its argument, the computation moves one level to the right (and, analogously, when the demanded value is computed, it moves back one level to the left). In this computation, the expressions of those configurations whose program positions are collected by mds are shown in a box.*

According to Definition 28, one could develop a slicing tool by implementing a meta-interpreter for the instrumented operational semantics (Figure 5.9) and, then, extracting the program positions from these configurations using function *mds* (Figure 5.13). Our aim in this chapter, however, is to compute the slice from the extended redex trail, *since it is already available in existing tracing tools.*

From the extended redex trail, the computation of a dynamic slice proceeds basically as follows: first, the node associated with the slicing criterion is identified (the *SC-node*, the counterpart of the SC-configuration); then, the set of program positions of the nodes which are *reachable* from the SC-node (according to the slicing pattern) are collected.

$$
\begin{array}{l}
mds([C_j], \pi, V, [C_{j+1}, \ldots, C_n]) \;=\; hd(pos(C_j)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \cup\; mds\_aux([C_{j+1}, \ldots, C_n], ctrl(C_j) \sqcap \pi)
\end{array}
$$

$$
mds([C_i, \ldots, C_j], \pi, V, [C_{j+1}, \ldots, C_n]) \;= \qquad [i < j]
$$

$$
\begin{cases}
hd(pos(C_i)) \\
\quad \cup\; mds([C_{i+1}, \ldots, C_j], \pi, V', [C_{j+1}, \ldots, C_n]) & \text{if } ctrl(C_i) = (let\ x = e\ in\ e') \\
& \text{where } V' = V \cup \{(x, \top)\} \\
\\
mds([C_{k+1}, \ldots, C_j], \pi, V, [C_{j+1}, \ldots, C_n]) & \text{if } ctrl(C_i) = x \\
& \text{and } (x, \pi') \notin V \\
& \text{where } C_i \Longrightarrow^* C_k \text{ is the} \\
& \text{complete subderivation} \\
& \text{for } C_i \text{ in } C_i \Longrightarrow^* C_j \\
\\
hd(pos(C_i)) \cup mds([C_{i+1}, \ldots, C_k], \pi', V, [\,]) & \text{if } ctrl(C_i) = x \\
\quad \cup\; mds([C_{k+1}, \ldots, C_j], \pi, V', [C_{j+1}, \ldots, C_n]) & \text{and } (x, \pi') \in V \\
& \text{where } C_i \Longrightarrow^* C_k \text{ is the} \\
& \text{complete subderivation} \\
& \text{for } C_i \text{ in } C_i \Longrightarrow^* C_j \\
& \text{and } V' = V \cup (ctrl(C_k) \sqcap \pi') \\
\\
hd(pos(C_i)) \\
\quad \cup\; mds([C_{i+1}, \ldots, C_j], \pi, V, [C_{j+1}, \ldots, C_n]) & \text{otherwise}
\end{cases}
$$

$$
mds\_aux([\,], V) \;=\; \{\,\}
$$

$$
mds\_aux([C_{j+1}, \ldots, C_n], V) \;=
$$

$$
\begin{cases}
mds([C_{j+1}, \ldots, C_k], \pi, V, [C_{k+1}, \ldots, C_n]) & \text{if } ctrl(C_{j+1}) = x \text{ and } (x, \pi) \in V, \\
& \text{where } C_{j+1} \Longrightarrow^* C_k \text{ is the} \\
& \text{complete subderivation} \\
& \text{for } C_{j+1} \text{ in } C_{j+1} \Longrightarrow^* C_n \\
\\
mds\_aux([C_{j+2}, \ldots, C_n], V) & \text{otherwise}
\end{cases}
$$

$$
e \sqcap \pi \;= \begin{cases}
\{(x_i, \pi_i) \mid \pi_i \neq \bot,\ i \in \{1, \ldots, m\}\} & \text{if } e = c(x_1, \ldots, x_m) \\
& \text{and } \pi = c(\pi_1, \ldots, \pi_m) \\
\{\overline{(x_m, \top)}\} & \text{if } e = c(x_1, \ldots, x_m) \text{ and } \pi = \top \\
\{\,\} & \text{otherwise}
\end{cases}
$$

Figure 5.13: Collecting the positions of the minimal dynamic slice (function *mds*)

```
main = let x = pair in f x

f x = case x of
      { C w1 w2 → case w1 of { Z → case w2 of { Z → Z } } }

pair = let z = zero in g z

g z = let y = one in case z of
                     { Z → case y of
                            { S w → case w
                                    { Z → let v1 = Z in
                                          let v2 = Z in
                                          C v1 v2 } } }

zero = Z

one  = let v = Z in S v
```

Figure 5.14: Program of Example 15

In order to identify the SC-node, we first need to define the counterpart of function $val_{\mathcal{D}}$:

Definition 29 *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation and $G = graph(C_m)$. The partial value associated with variable x in G is given by $Val_{\mathcal{D}}(x)$, where function $Val_{\mathcal{D}}$ is defined as follows:*

$$Val_{\mathcal{D}}(x) = \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(q) & \text{if } (x \rightsquigarrow r) \in G \text{ and } (r \underset{q}{\mapsto}) \in G \\ c(\overline{Val_{\mathcal{D}}(x_n)}) & \text{if } (x \rightsquigarrow r) \in G \text{ and } (r \mapsto c(\overline{x_n})) \in G \\ \_ & \text{otherwise} \end{cases}$$

The auxiliary function $\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}$ is used to follow the successor chain in a graph:

$$\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(r) \;=\; \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(q) & \text{if } (r \underset{q}{\mapsto}) \in G \\ c(\overline{Val_{\mathcal{D}}(x_n)}) & \text{if } (r \mapsto c(\overline{x_n})) \in G \\ \_ & \text{otherwise} \end{cases}$$

where $\Gamma = heap(C_m)$ is the heap in the last configuration of $\mathcal{D}$.

Observe the similarities between functions $val_{\mathcal{D}}$ and $Val_{\mathcal{D}}$. The main difference is that, in order to get the final value of a variable, $val_{\mathcal{D}}$ looks at the bindings in the

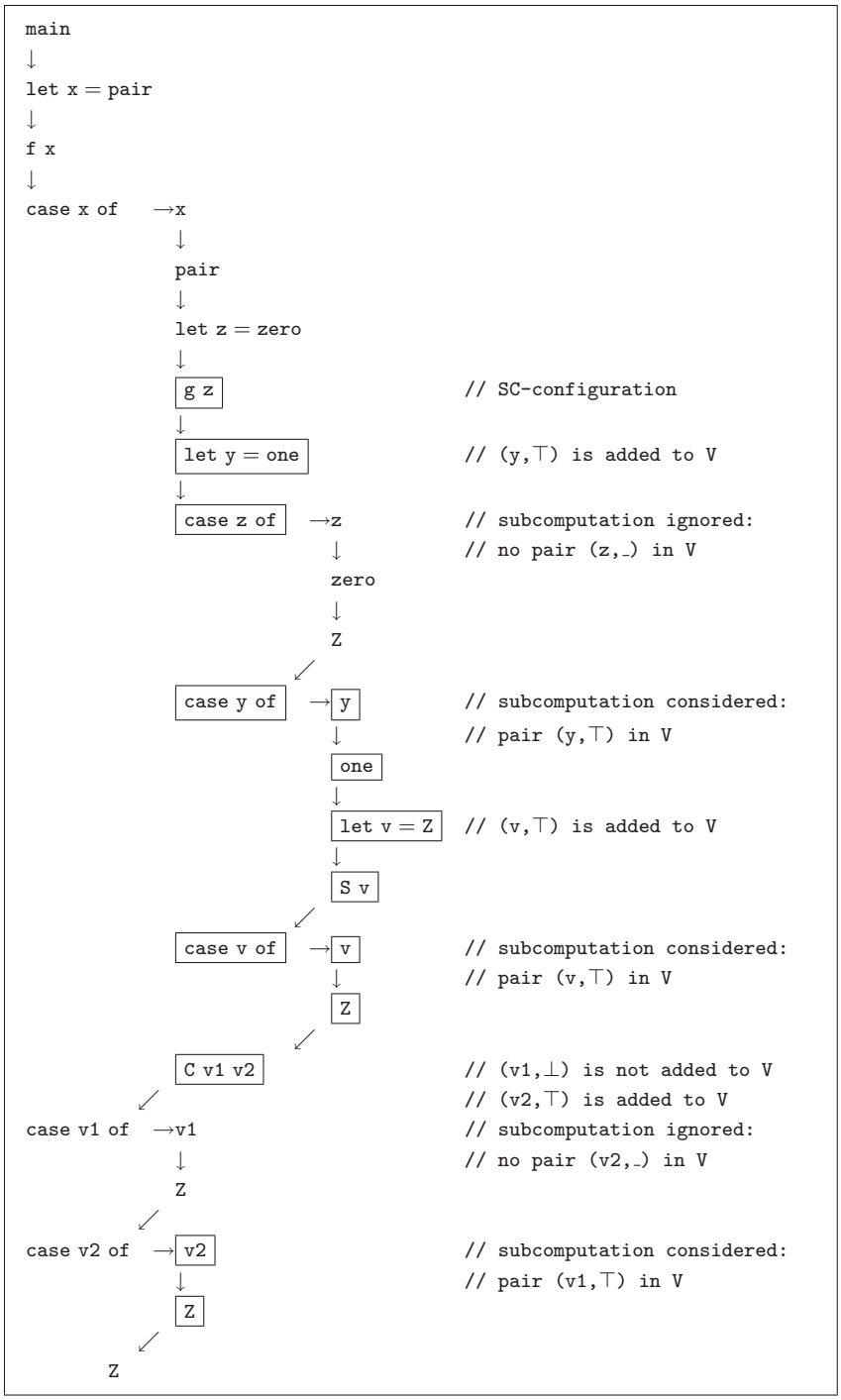


Figure 5.15: Computation of a minimal dynamic slice (Example 15)

$$
\begin{array}{l}
ds(G, r, \pi, V) \;=\; \\
\left\{\begin{array}{lll}
\mathcal{P} \;\cup\; ds(G, r_1, \pi_1, \{\,\}) & \text{if } (r \mapsto_{\mathcal{P}} c(\overline{x_n})) \in G, & \\
\quad \cup\; \ldots & \text{where } \{\overline{(x_k, \pi_k)} \mid (x', \pi') \in c(\overline{x_n}) \sqcap \pi \text{ and } \pi' \neq \bot\}, & \\
\quad \cup\; ds(G, r_k, \pi_k, \{\,\}) & \text{and } (x_j \rightsquigarrow r_j) \in G \text{ for all } j = 1, \ldots, k & \\
\mathcal{P} \;\cup\; ds(G, q, \pi, V') & \text{if } (r \underset{q}{\mapsto}_{\mathcal{P}} \mathit{let}\ x = e\ \mathit{in}\ e') \in G, \text{ where } V' = V \cup \{(x, \top)\} & \\
\mathcal{P} \;\cup\; ds(G, q, \pi, V) & \text{if } (r \underset{q}{\mapsto}_{\mathcal{P}} \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}) \in G \text{ and } (x, \pi') \notin V & \\
\mathcal{P} \;\cup\; ds(G, q', \pi', V) & \text{if } (r \underset{q}{\mapsto}_{\mathcal{P}} \mathit{case}\ x\ \mathit{of}\ \{\overline{p_k \to e_k}\}) \in G \text{ and } (x, \pi') \in V, & \\
\quad \cup\; ds(G, q, \pi, V) & \text{where } (x \rightsquigarrow q') \in G & \\
\mathcal{P} \;\cup\; ds(G, q, \pi, V) & \text{otherwise, where } (r \underset{q}{\mapsto}) \in G &
\end{array}\right.
\end{array}
$$

Figure 5.16: Collecting the positions of a dynamic slice (function *ds*)

heap of the final configuration of a computation, while $Val_{\mathcal{D}}$ looks at the graph in the final configuration of this computation.

Now, the node associated with a slicing criterion in the extended redex trail can be determined as follows:

Definition 30 (SC-node) *Let P be a program and $(C_0 \Longrightarrow^* C_m)$ a complete derivation, where $G = graph(C_m)$. Given a slicing criterion $\langle f(\overline{pv_n}), \pi\rangle$, the associated SC-node must fulfill the conditions:*[9]

- $(r \mapsto f(\overline{x_n})) \in G$ *and*
- $pv_i = Val_{\mathcal{D}}(x_i)$ *for all* $i = 1, \ldots, n$.

We now introduce a function to compute a dynamic slice from the program positions of the nodes that are reachable from the SC-node (according to the slicing pattern).

Definition 31 (dynamic slice, *ds*) *Let G be an extended redex trail. Let $\langle f(\overline{pv_n}), \pi\rangle$ be a slicing criterion and r the reference of its associated SC-node. A dynamic slice of $\langle f(\overline{pv_n}), \pi\rangle$ in G is given by $ds(G, r, \pi, \{\})$, where function ds is defined in Figure 5.16.*

The parallelism between functions *mds* and *ds* should be clear. While traversing the complete subcomputation for the slicing criterion, both functions

[9] As in Definition 26, if there are several nodes that fulfill these conditions, we choose the first one following the natural, argument-first, traversal of the graph.

$$
\begin{array}{l}
\langle [\,], \texttt{main}, [\,], G_{\varnothing}, 0, [\,]\rangle \\
\Longrightarrow \langle [\,], \texttt{let x} = \texttt{zero in f x x}, [\,], \texttt{G}[0 \underset{1}{\mapsto}_{[\,]} \texttt{main}], 1, [(\texttt{main}, \Lambda)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{main},1)} \texttt{zero}], \texttt{f x x}, [\,], \texttt{G}[1 \underset{2}{\mapsto}_{[(\texttt{main},\Lambda)]} \texttt{let x} = \texttt{zero} \ldots], 2, [(\texttt{main}, 2)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{main},1)} \texttt{zero}], \texttt{case x of} \ldots, [\,], \texttt{G}[2 \underset{3}{\mapsto}_{[(\texttt{main},2)]} \texttt{f x x}], 3, [(\texttt{f}, \Lambda)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{main},1)} \texttt{zero}], \texttt{x}, [(\{\texttt{Z} \rightarrow \texttt{g x}\}, 3)], \texttt{G}[3 \mapsto_{[(\texttt{f},\Lambda)]} \texttt{case x of} \ldots], 4, [(\texttt{f}, 1)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{main},1)} \texttt{zero}], \texttt{zero}, [\texttt{x}, (\{\texttt{Z} \rightarrow \texttt{g x}\}, 3)], \texttt{G}[\texttt{x} \rightsquigarrow 4], 4, [(\texttt{main}, 1), (\texttt{f}, 1)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{main},1)} \texttt{zero}], \texttt{Z}, [\texttt{x}, (\{\texttt{Z} \rightarrow \texttt{g x}\}, 3)], \texttt{G}[4 \underset{5}{\mapsto}_{[(\texttt{main},1),(\texttt{f},1)]} \texttt{zero}], 5, [(\texttt{zero}, \Lambda)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{zero},\Lambda)} \texttt{Z}], \texttt{Z}, [(\{\texttt{Z} \rightarrow \texttt{g x}\}, 3)], \texttt{G}, 5, [(\texttt{zero}, \Lambda)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{zero},\Lambda)} \texttt{Z}], \texttt{g x}, [\,], \texttt{G}[5 \mapsto_{[(\texttt{zero},\Lambda)]} \texttt{Z},\ 3 \underset{6}{\mapsto}_{[(\texttt{f},\Lambda)]} \texttt{case x of} \ldots], 5, [(\texttt{f}, 2.1)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{zero},\Lambda)} \texttt{Z}], \texttt{case x of} \ldots, [\,], \texttt{G}[6 \underset{7}{\mapsto}_{[(\texttt{f},2.1)]} \texttt{g x}], 7, [(\texttt{g}, \Lambda)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{zero},\Lambda)} \texttt{Z}], \texttt{x}, [(\{\texttt{Z} \rightarrow \texttt{Z}\}, 7)], \texttt{G}[7 \mapsto_{[(\texttt{g},\Lambda)]} \texttt{case x of} \ldots], 8, [(\texttt{g}, 1)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{zero},\Lambda)} \texttt{Z}], \texttt{Z}, [(\{\texttt{Z} \rightarrow \texttt{Z}\}, 7)], \texttt{G}[4 \underset{5}{\mapsto}_{[(\texttt{main},1),(\texttt{f},1),(\texttt{g},1)]} \texttt{zero}], 8, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [(zero, \Lambda), (g, 1)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{zero},\Lambda)} \texttt{Z}], \texttt{Z}, [\,], \texttt{G}[8 \mapsto_{[(\texttt{zero},\Lambda),(\texttt{g},1)]} \texttt{Z},\ 7 \underset{9}{\mapsto}_{[(\texttt{g},\Lambda)]} \texttt{case x of} \ldots], 8, [(\texttt{g}, 2.1)]\rangle \\
\Longrightarrow \langle [\texttt{x} \mapsto_{(\texttt{zero},\Lambda)} \texttt{Z}], \diamond, [\,], \texttt{G}[9 \mapsto_{[(\texttt{g},2.1)]} \texttt{Z}], \square, [\,]\rangle
\end{array}
$$

Figure 5.17: Complete computation for $\langle [\,], \texttt{main}, [\,], G_{\varnothing}, 0, [\,]\rangle$

- consider the subcomputations for those variables introduced in a let expression *after* the slicing criterion and
- ignore the subcomputations for those variables introduced in a let expression *before* the slicing criterion.

Once the value of the slicing criterion is reached, both functions use the slicing pattern to determine which further subcomputations should be considered.

Observe, however, that there is a significant difference between functions *mds* and *ds*. Consider, for instance, the following program:

```
main = let x = zero in f x x

f x y = case x of { Z → g y }

g y   = case y of { Z → Z }

zero = Z
```

and the computation which is shown in Fig. 5.17.

Given the slicing criterion $\langle \texttt{g Z}, \top\rangle$, function *mds* computes the following program positions:

$(\mathtt{f}, 2.1)$: the call `g x`;

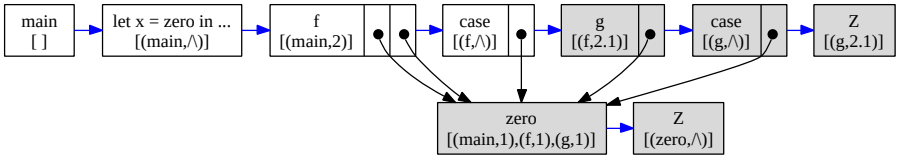
$(\mathtt{g}, \Lambda)$: the case expression in the right-hand side of function `g`;

$(\mathtt{g}, 1)$: the argument of this case expression;

$(\mathtt{zero}, \Lambda)$: the constant `Z` in the right-hand side of function `zero`;

$(\mathtt{g}, 2.1)$: the constant `Z` in the branch of the case expression of `g`.

The extended redex trail associated with this derivation is as follows:



Here, the nodes that belong to the dynamic slice (according to function *ds*) are shadowed. In contrast to function *mds*, function *ds* also gather the program position of function `zero` because we cannot distinguish in the redex trail which case expression demanded first the evaluation of variable `x`. In other words, the dynamic slices computed by *ds* are not minimal because they cannot determine when a variable is first evaluated from the information in the extended redex trail. Of course, this is due to the use of a sharing-based semantics. In a call-by-name semantics without sharing, the slices computed by *ds* would also be minimal.

Nevertheless, we think that computing slices from redex trails (rather than implementing a new *ad-hoc* technique) pays off in practice despite this small loss of precision.

Now, we prove that dynamic slices computed by function *ds* (Definition 31) are indeed complete:

Conjecture 32 (completeness) *Let P be a program, $(C_0 \Longrightarrow^* C_m)$, $m > 0$, a complete derivation, and $G = graph(C_m)$. Let $\langle f(\overline{pv_n}), \pi \rangle$ be a slicing criterion and $\mathcal{W}$ be the minimal dynamic slice computed according to Definition 28. Then, we have $ds(G, r, \pi, \{\,\}) \supseteq \mathcal{W}$, where r is the SC-node associated with the slicing criterion.*

We only prove this result for the non-logical features of the language (i.e., w.r.t. the instrumented semantics of Figure 5.9). The extension to cover the logical features does not involve additional technical difficulties.

First, we need some results from [Braßel *et al.*, 2004b] that we slightly modify to include program positions and ignore parent arrows: the notion of *successor* derivation and Propositions 34 and 35, which correspond, respectively, to Propositions 5.5 and 5.8 in [Braßel *et al.*, 2004b]. Firstly, we introduce the notion of successor derivation.

Definition 33 (successor derivation) *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a derivation. Let $C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, $0 \leq i < m$, be a relevant configuration such that e is a function call, a let expression, or a case expression. Then, $C_i \Longrightarrow^* C_j$, $i < j \leq m$, is a successor subderivation of $\mathcal{D}$ iff (i) C_j is relevant, (ii) $stack(C_i) = stack(C_j)$, and (iii) there is no relevant configuration C_k, $i < k < j$, such that $stack(C_i) = stack(C_k)$.*

Intuitively, a successor derivation represents a single reduction step (including the associated subcomputations, if any). The following result states that, for each successor subderivation in a computation, the instrumented semantics adds a corresponding successor arrow to the graph, and vice versa.

Proposition 34 *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a derivation. There exists a successor subderivation*

$$C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle \Longrightarrow^* \langle \Gamma', e', S', G', r', \mathcal{P}' \rangle = C_j$$

$0 \leq i < j < m$, in $\mathcal{D}$, iff $(r \underset{r'}{\mapsto_{\mathcal{P}}} e) \in graph(C_{j+1})$ and $(r' \underset{q}{\mapsto_{\mathcal{P}'}} e') \in graph(C_{j+1})$.

As mentioned before, when a lookup configuration $\langle \Gamma, x, S, G, r, p, \mathcal{P} \rangle$ appears in a computation, a variable pointer $x \rightsquigarrow r$ should be added to the graph. Moreover, reference r stores the *dereferenced value* of heap variable x. This is expressed by means of function Γ^* which is defined as follows:

$$\Gamma^*(x) = \begin{cases} \Gamma^*(y) & \text{if } \Gamma[x] = y \text{ and } x \neq y \\ LogVar & \text{if } \Gamma[x] = x \\ \Gamma[x] & \text{otherwise} \end{cases}$$

The next proposition formally states that, for each variable whose value is *demanded* in a computation, we have an associated variable pointer in the graph.

Proposition 35 *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a derivation. There exists a lookup configuration $C_i = \langle \Gamma, x, S, G, r, \mathcal{P} \rangle$, $0 < i < m$, in $\mathcal{D}$, iff $(x \rightsquigarrow r) \in graph(C_{j+1})$ and $r \mapsto_{\mathcal{P}} \Gamma^*(x) \in graph(C_{j+1})$, where C_j, $i \leq j < m$, is a relevant configuration and there is no relevant configuration C_k with $i < k < j$.*

Observe that all the configurations in the subderivation $C_i \Longrightarrow^* C_{j-1}$ have a variable in the control. Therefore, only rules VarCons and VarExp can be applied, which implies that their program positions are accumulated to the list of program positions in C_j.

In order to prove the completeness of the computed slice, we first need to prove that, giving a slicing criterion, the associated SC-node correctly points to the node that stores the control of the associated SC-configuration. For this purpose, we need some preliminary results.

The next lemma shows that reductions in a derivation can also be obtained from the extended redex trail by following the successor relation.

Lemma 36 *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation and $C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle$, $0 < i < m$, be a relevant configuration. If there exists a complete subderivation $C_i \Longrightarrow^* C_j$ for C_i in $\mathcal{D}$, then there exists a successor chain $(r \underset{q_1}{\mapsto}_{\mathcal{P}} e), (q_1 \underset{q_2}{\mapsto}_{\mathcal{P}_1} e_1), \ldots, (q_n \mapsto_{\mathcal{P}_n} e_n) \in graph(C_m)$ with $e_n = ctrl(C_j)$.*

Proof. Since C_i is a relevant configuration, we have that $C_i \Longrightarrow^* C_j$ can be decomposed into a sequence of successor subderivations

$$\begin{array}{rcl} C_i = \langle \Gamma, e, S, G, r, \mathcal{P} \rangle & \Longrightarrow^* & \langle \Gamma_1, e_1, S_1, G_1, q_1, \mathcal{P}_1 \rangle \\ & \Longrightarrow^* & \langle \Gamma_2, e_2, S_2, G_2, q_2, \mathcal{P}_2 \rangle \\ & \ldots & \\ & \Longrightarrow^* & \langle \Gamma_n, e_n, S_n, G_n, q_n, \mathcal{P}_n \rangle \end{array}$$

Therefore, by Proposition 34, the claim follows.

□

The next lemma shows that, for each variable x demanded in a derivation $\mathcal{D}$, we have $val_{\mathcal{D}}(x) = Val_{\mathcal{D}}(x)$.

Lemma 37 *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$, $m > 0$, be a complete derivation and $C_i = \langle \Gamma, x, S, G, r, \mathcal{P} \rangle$, $0 < i < m$, be a lookup configuration. Then, $val_{\mathcal{D}}(x) = Val_{\mathcal{D}}(x)$.*

Proof. We prove this claim by structural induction on the computed partial value pv.

(Base case) Then, pv is a constructor constant c. By definition of function val, we have: $val_{\mathcal{D}}(x) = val^v_{\mathcal{D}}(x) = val^v_{\mathcal{D}}(y_1) = \ldots = val^v_{\mathcal{D}}(y_n) = c$. Hence, we have $\Gamma^*(x) = c$. Moreover, by Proposition 35, we have that $(x \rightsquigarrow r) \in G$ and $(r \mapsto \Gamma^*(x)) \in G$, where $G = graph(C_m)$. Thus, $val_{\mathcal{D}}(x) = Val_{\mathcal{D}}(x) = c$.

(Inductive case) We now consider that the partial value pv is a constructor call $c(\overline{x_n})$. Then, $\Gamma[x] = c(\overline{x_n})$ and $val_{\mathcal{D}}(x) = val^v_{\mathcal{D}}(x) = c(\overline{val_{\mathcal{D}}(x_n)})$. By Proposition 35, we have that $(x \rightsquigarrow r) \in G$ and $(r \mapsto \Gamma^*(x)) \in G$, where $G = graph(C_m)$. Therefore, the claim follows by the inductive hypothesis since $val_{\mathcal{D}}(x_i) = Val_{\mathcal{D}}(x_i)$ for all $i = 1, \ldots, n$.

□

We now establish the equivalence between the notions of SC-configuration and SC-node for a given slicing criterion.

Theorem 38 *Let $\mathcal{D} = (C_0 \Longrightarrow^* C_m)$ be a complete derivation. Given a slicing criterion $\langle f(\overline{pv_n}), \pi \rangle$, if the associated SC-configuration is $C_i = \langle \Gamma, f(\overline{x_n}), S, G, r, \mathcal{P} \rangle$, then the associated SC-node is r.*

Proof. Both the definition of SC-configuration and SC-node require two conditions. We consider each of them separately:

(1) Trivially, $C_i = \langle \Gamma, f(\overline{x_n}), S, G, r, \mathcal{P} \rangle$ is a relevant configuration. Therefore, by Lemma 20, we have that $G_{i+1} = G[r \mapsto_{\mathcal{P}} f(\overline{x_n})]$ and, thus, $(r \mapsto_{\mathcal{P}} f(\overline{x_n})) \in \mathit{graph}(C_m)$.

(2) This case follows by Lemma 37, since it implies that $\mathit{val}_{\mathcal{D}}(x_i) = \mathit{Val}_{\mathcal{D}}(x_i)$, with $(x_i \rightsquigarrow r_i) \in \mathit{graph}(C_m)$, for all $i = 1, \ldots, n$.

□

Finally, we sketch the proof of Conjecture 32:

Proof. (Sketch) Let us consider that $C_i = \langle \Gamma_i, f(\overline{x_n}), S_i, G_i, r_i, \mathcal{P}_i \rangle$ is the SC-configuration associated with the slicing criterion. By Theorem 38, we have that $r = r_i$. Let $C_i \Longrightarrow^* C_j$ is the complete subderivation for C_i in $(C_0 \Longrightarrow^* C_m)$. Now, we show that, for each recursive call to *mds* in $\mathit{mds}([C_i, \ldots, C_j], \pi, \{\}, [C_{j+1}, \ldots, C_m])$, there is an appropriate counterpart in $\mathit{ds}(G, r, \pi, \{\,\})$.

Consider an arbitrary call $\mathit{mds}([C_l, \ldots, C_j], \pi, V, [C_{j+1}, \ldots, C_m])$. Then, we distinguish the following possibilities:

- If the control of C_l is a let expression, then the first position in $\mathit{pos}(C)$ is collected. Analogously, the set $\mathit{pos}(C)$ is collected in function *ds*. While function *mds* is now applied to the derived configuration C_{l+1}, function *ds* is applied to the successor node, which is equivalent.

- If the control of C_l is a variable and there is no pair (x, π') in V, then function *mds* ignores the program positions in the complete subderivation $C_l \Longrightarrow^* C_k$ for C_l and continues by considering the next configuration C_{k+1}. Analogously, function *ds* ignores the program positions associated with the variable argument x of a case expression when there is no pair (x, π') in V, and continues by considering the successor of the case expression.

- If the control of C_l is a variable and there is a pair (x, π') in V, then function *mds* first collects the program positions in the complete subderivation $C_l \Longrightarrow^* C_k$ for C_l and, then, continues by considering the next configuration C_{k+1}. Analogously, function *ds* first collects the program positions associated with the variable argument x of a case expression when there is a pair (x, π') in V and, then, continues by considering the successor of the case expression.

- Finally, in any other case, both functions *mds* and *ds* collect the current program positions and are recursively applied to the next configuration/node.

□

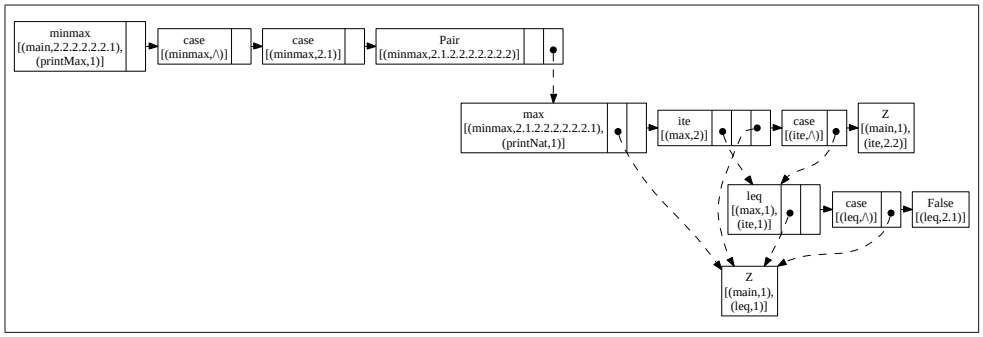


Figure 5.18: Reachable nodes w.r.t. $\langle$`minmax (Z : _ : _), Pair _ Z, 1, Pair` $\bot$ $\top\rangle$

Example 16 *Consider the (normalized version) of the program in Example 11 and the slicing criterion* $\langle$`minmax (Z : _ : _), Pair` $\bot$ $\top\rangle$. *As we have already seen in Section 5.2, the extended redex trail of Figure 5.3 is computed. Figure 5.18 shows the SC-node together with the nodes which are reachable from the SC-node according to the slicing pattern* (`Pair` $\bot$ $\top$)*, including the associated lists of program positions. The computed slice is the one already shown in Figure 5.4.*

5.4.1 Logical Features

In this section, we sketch the extension of the previous developments in this section in order to cope with the logical features of the considered functional logic language.

The first notion that should be extended is that of slicing criterion. Definition 22 is not appropriate in a lazy functional *logic* language because of non-determinism, since the same function call may be reduced to different values within the same computation. Consider, e.g., the following function definition:

```
coin = Z or (S Z)
```

Here, `coin` can be non-deterministically reduced to `Z` and (`S Z`). In order to identify a concrete occurrence of the given call, we add another element to the slicing criterion: the computed value.[10] Therefore, a slicing criterion is defined in the functional logic setting as a tuple $\langle f(\overline{v_n}), v, \pi\rangle$, where $f(\overline{v_n})$ is a function call whose arguments appear as much evaluated as needed in the complete computation, v is the computed value, and π is a pattern that determines the interesting part of the computed value, e.g., $\langle$`coin, Z,` $\top\rangle$.

Regarding the notions of SC-configuration and SC-node, only functions $val_{\mathcal{D}}$ and

[10] Observe that we can still have an expression like `coin + coin` that evaluates to the same value (`S Z`) in different ways. Here, we implicitly consider the first of such computations. This is somehow a limitation, but it is not realistic to expect that users are able to distinguish among different computations which share both the same initial function call and the final value.

$Val_{\mathcal{D}}$ should be slightly extended in order to also consider a logical variable as a value. For this purpose, auxiliary function $val^v_{\mathcal{D}}$ is defined as follows:

$$val^v_{\mathcal{D}}(x) \;=\; \begin{cases} val^v_{\mathcal{D}}(y) & \text{if } \Gamma[x] = y \text{ and } x \neq y \\ c(\overline{val_{\mathcal{D}}(x_n)}) & \text{if } \Gamma[x] = c(\overline{x_n}) \\ LogVar & \text{if } \Gamma[x] = x \\ \_ & \text{otherwise} \end{cases}$$

Correspondingly, function $\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}$ is defined as follows:

$$\mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(r) \;=\; \begin{cases} \mathsf{val}^{\mathsf{v}}_{\mathsf{G}}(q) & \text{if } (r \underset{q}{\mapsto}) \in G \\ c(\overline{Val_{\mathcal{D}}(x_n)}) & \text{if } (r \mapsto c(\overline{x_n})) \in G \\ LogVar & \text{if } (r \mapsto LogVar) \in G \\ \_ & \text{otherwise} \end{cases}$$

Finally, functions *mds* and *ds* require no significant changes, i.e., the only difference is that the underlying calculus also includes the rules of Fig. 5.11.

5.5 Program Specialization Based on Dynamic Slicing

This section extends the slicing technique introduced in the former sections. Our aim is similar to that of Reps and Turnidge [1996], who designed a program specialization method for *strict* functional programs based on *static* slicing. However, in contrast to Reps and Turnidge [1996], we consider lazy functional logic programs and our technique is based on *dynamic* slicing. We consider dynamic slicing since it is simpler and more accurate than static slicing. Informally speaking, specialization proceeds as follows: first, the user identifies a *representative* set of slicing criteria; then, dynamic slicing is used to compute a program slice for each slicing criterion; finally, the specialized program is built by conveniently unioning the computed slices (e.g., with the SDS procedure by Hall [1995]). This is a lightweight approach since dynamic slicing is simpler than static slicing (as in [Reps and Turnidge, 1996]). Unfortunately, in our previous formalization, we defined a program slice as a set of *program positions* rather than as an executable program (See Definition 21), since this is sufficient for debugging. Therefore, we also introduce an appropriate technique to extract executable slices from a set of program positions.

5.5.1 Specialization Based on Slicing

Consider again function `minmax` shown (now corrected) in Figure 5.19.

```
minmax xs = case xs of
            { y:ys → case ys of
                     { []   → Pair y y;
                       z:zs → let m = minmax (z:zs)
                              in Pair (min y (fst m))
                                      (max y (snd m)) } }

min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

fst t = case t of { Pair x y → x }
snd t = case t of { Pair x y → y }

ite x y z = case x of { True → y; False → z }

leq x y = case x of
          { Z    → True;
            (S n) → case y of { Z → False; (S m) → leq n m } }
```

Figure 5.19: Example program `minmax`

Our aim now is the specialization of this program in order to extract those statements which are needed to compute *the minimum* of a list. For this, we first consider the selection of appropriate slicing criteria and, then, the extraction of the corresponding slice (the *specialized* program).

Figure 5.20 shows the slice computed for `minmax` w.r.t. the slicing criterion $\langle$`minmax [Z, S Z],Pair` $\top$ $\bot\rangle$. This notion of slice is sufficient for debugging, where we are interested in highlighting those expressions which are related to the slicing criterion. In order to specialize programs, however, we should produce *executable* program slices. For this purpose, we need to introduce an algorithm that extracts an executable program slice from the original program and the computed set of program positions. For instance, an executable program slice associated to `minmax` is shown in Figure 5.21, where the distinguished symbol "?" is used to point out that some subexpression is missing due to the slicing process.

Informally, the specialized program is obtained as follows: those functions that were not used in any computation are completely deleted; for the remaining expressions, we follow a simple rule: if their location belongs to the computed set, it appears in the slice; otherwise, it is replaced by the special symbol "?".

Observe that some further simplifications are obvious, as for instance deleting the unused branches of `case` expressions, or replacing the expression

```
minmax xs = case xs of
            { y:ys → case ys of
                     { []   →  Pair y y;
                       z:zs → let m = minmax (z:zs)
                              in Pair (min y (fst m))
                                      (max y (snd m)) } }

min x y = ite (leq x y) x y
max x y = ite (leq x y) y x

fst t = case t of { Pair x y → x }
snd t = case t of { Pair x y → y }

ite x y z = case x of { True → y; False → z }

leq x y = case x of
          { Z    → True;
            (S n) → case y of { Z → False; (S m) → leq n m } }
```

Figure 5.20: Slice of program `minmax` w.r.t. $\langle$`minmax [Z, S Z], Pair` $\top$ $\bot\rangle$

```
let ? in Pair (min y ?) ?
```

by `Pair (min y ?) ?`, since the first argument of the `let` expression is not used. Therefore, after the specialization, we use a simplification process in order to further simplify programs. For instance, the simplified version of the program in Figure 5.21 is shown in Figure 5.22.

Clearly, producing a specialized program for a concrete computation is not always appropriate. For instance, in the slice above, function `minmax` does not work with lists one element. To overcome this problem, [Reps and Turnidge, 1996] considers *static* slicing where no input data are provided and, thus, the extracted slices preserve the semantics of the original program for any input data. Our lightweight approach is based on dynamic slicing and, thus, a *representative* set of slicing criteria—covering the interesting computations—should be determined in order to obtain program slices which are general enough.

Now, we formalize our method to extract an *executable* program slice from the computed program positions.

Definition 39 (executable slice) *Let P be a program and $\mathcal{W}$ an associated dynamic slice. The corresponding executable slice, $P_{\mathcal{W}}$, is obtained as follows:*

```
minmax xs = case xs of
            { y:ys → case ys of
                     { []   → ?;
                       z:zs → let ?
                              in Pair (min y ?) ? } }

min x y = ite (leq x ?) x ?

ite x y z = case x of { True → y; False → ? }

leq x y = case x of
          { Z    → True;
            (S n) → ? } }
```

Figure 5.21: Executable slice of program `minmax` w.r.t. $\langle$`minmax [Z, S Z], Pair` $\top$ $\bot\rangle$

```
minmax xs = case xs of
            { y:ys → case ys of
                       z:zs → Pair (min y ?) ? } }

min x y = ite (leq x ?) x ?

ite x y z = case x of { True → y }

leq x y = case x of { Z → True } }
```

Figure 5.22: Simplified executable slice of program `minmax` w.r.t. $\langle$`minmax [Z, S Z], Pair` $\top$ $\bot\rangle$

$$P_{\mathcal{W}} = \{ f(\overline{x_n}) = [\![e]\!]^{\Lambda}_{\mathcal{Q}} \mid (f,\Lambda) \in \mathcal{W} \ \textit{ and } \ \mathcal{Q} = \{p \mid (f,p) \in \mathcal{W}\} \ \}$$

where $[\![e]\!]^p_{\mathcal{Q}} = ?$ *if* $p \notin \mathcal{Q}$ *and, otherwise (i.e.,* $p \in \mathcal{Q}$*), it is defined inductively as follows:*

$$\begin{aligned}
[\![x]\!]^p_{\mathcal{Q}} &= x \\
[\![\varphi(x_1,\ldots,x_n)]\!]^p_{\mathcal{Q}} &= \varphi(x_1,\ldots,x_n) \\
[\![let\ x = e'\ in\ e]\!]^p_{\mathcal{Q}} &= let\ x = [\![e']\!]^{p.1}_{\mathcal{Q}}\ in\ [\![e]\!]^{p.2}_{\mathcal{Q}} \\
[\![e_1\ or\ e_2]\!]^p_{\mathcal{Q}} &= [\![e_1]\!]^{p.1}_{\mathcal{Q}}\ or\ [\![e_2]\!]^{p.2}_{\mathcal{Q}} \\
[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!]^p_{\mathcal{Q}} &= (f)case\ x\ of\ \{\overline{p_k \to [\![e_k]\!]^{p.2.k}_{\mathcal{Q}}}\}
\end{aligned}$$

where $\varphi \in (\mathcal{C} \cup \mathcal{F})$ *is either a constructor or a defined function symbol.*

The slice computed in this way would be trivially executable by considering "?" as a fresh 0-ary constructor symbol (since it is not used in the considered computation). However, from an implementation point of view, this is not a good decision since we should also extend all program types in order to include "?". Fortunately, in our functional *logic* setting, there is a simpler solution: we can replace "?" by a fresh constructor `failed` (which represents $\bot$).

Now, the specialization process should be clear. First, we compute dynamic slices—sets of program positions—for the selected slicing criteria. Then, an executable slice is built for the combination of the computed dynamic slices. Moreover, there are some post-processing simplifications that can be added in order to reduce the program size and improve its readability:

$$\begin{aligned}
let\ x = ?\ in\ e &\implies \rho(e) \qquad \text{if } \rho = \{x \mapsto ?\} \\
e\ or\ ? &\implies e \\
(f)case\ x\ of\ \{\overline{p_k \to e_k}\} &\implies (f)case\ x\ of\ \{\overline{p'_m \to e'_m}\} \\
&\qquad \text{if } \{\overline{p'_m \to e'_m}\} = \{p_i \to e_i \mid e_i \neq ?,\ i = 1,\ldots,k\}
\end{aligned}$$

More powerful post-processing simplifications could be added but they require a form of *amorphous* slicing (see Chapter 1).

5.6 Implementation Issues

In order to check the usefulness and practicality of the ideas presented in this chapter, we have developed a prototype implementation of the debugging tool for the multi-paradigm language Curry [Hanus, 2006a] that has been sketched in Section 5.2. It combines both tracing and slicing by extending a previous tracer for Curry programs [Braßel *et al.*, 2004b]. Three main extensions were necessary:

- First, we modified the instrumented interpreter of Braßel *et al.* [2004b] that generates the redex trail in order to also include program positions. Program

$(\|e\|)_\Lambda = \Lambda$	for all expression e
$(\|c(\overline{e_n})\|)_{i.w} = i.(\|e_i\|)_w$	if $i \in \{1,\ldots,n\}$
$(\|f(\overline{e_n})\|)_{i.w} = i.(\|e_i\|)_w$	if $i \in \{1,\ldots,n\}$
$(\|let\ x = e\ in\ e'\|)_{1.w} = p_x.(\|e\|)_w$	where p_x is the position of x in e'
$(\|let\ x = e\ in\ e'\|)_{2.w} = (\|\sigma(e')\|)_w$	where $\sigma = \{x \mapsto e\}$
$(\|(f)case\ e\ of\ \{\overline{p_n \to e_n}\}\|)_{1.w} = 1.(\|e\|)_w$	
$(\|(f)case\ e\ of\ \{\overline{p_n \to e_n}\}\|)_{2.i.w} = 2.i.(\|e_i\|)_w$	if $i \in \{1,\ldots,n\}$

Figure 5.23: Conversion of positions from normalized flat programs to flat programs

$[\|e\|]^n_\Lambda = (n, \Lambda)$	for all expression e
$[\|c(\overline{e_k})\|]^n_{i.w} = (m, i.v)$	if $i \in \{1,\ldots,k\}$ and $[\|e_i\|]^n_w = (m, v)$
$[\|f(\overline{e_k})\|]^n_{i.w} = (m, i.v)$	if $i \in \{1,\ldots,k\}$ and $[\|e_i\|]^n_w = (m, v)$
$[\|(f)case\ e\ of\ \{\overline{p_k \to e_k}\}\|]^n_{2.i.w} = (n+m, v)$	if $i \in \{1,\ldots,k\}$ and $[\|e_i\|]^l_w = (m, v)$, where $l = brs(\overline{p_{i-1} \to e_{i-1}})$

Figure 5.24: Conversion of positions from flat programs to source programs

positions are added in such a way that they do not interfere with the tracing process.

- Then, we defined algorithms to convert program positions for normalized flat programs—the programs considered in the instrumented semantics—to source Curry programs. This is necessary to be able to show program slices at an adequate level of abstraction. The kernel of the implemented conversion algorithms are shown in Figures 5.23 and 5.24. For simplicity, here we consider that let expressions are not allowed in source Curry programs, i.e., they are only used to take care of sharing.

  Given an expression e in a normalized flat program and its position w, we have that $(|e|)_w$ returns the position of e in the corresponding flat (non-normalized) program. Basically, transforming a non-normalized flat program into a flat program consists in applying inlining to every let expression. Consequently, function $(|\ |)$ only changes those positions inside a let expression in order to reflect these inlining steps. For instance, the position 2 addresses the call `f x` in the right-hand side of the following normalized rule:

  ```
  main = let x = Z in f x
  ```

  The associated position in the non-normalized rule

  ```
  main = f Z
  ```

is $(|\texttt{let x = Z in f x}|)_2 = (|\texttt{f Z}|)_\Lambda = \Lambda$.

Then, function [| |] takes an expression e in a flat program and its position w, so that $[|e|]^1_w$ returns the position of e in the corresponding source program. Transforming a flat program into a source Curry program mainly consists in replacing case expressions by pattern matching in the left-hand sides of different rules. Therefore, function [| |] only changes positions inside a case expression so that prefixes which are used to identify a particular case branch are deleted and the rule number within the function definition is determined instead. The superscript in function [| |] is used to calculate the rule number within a definition (so it is initialized to 1). Auxiliary function *brs* is used to count the number of case branches within a list of expressions. Here, we assume that source Curry programs have no case expressions and that case expressions have always a variable argument.[11]

For instance, consider the following flat rule:

$$\texttt{g x} \;=\; \texttt{case x of } \{\texttt{Z} \to \texttt{Z};\ \texttt{S y} \to \texttt{f y}\}$$

The position 2.2 addresses the call to `f y` in the right-hand side of the second branch of the case expression. Given the associated source Curry rule:

$$\begin{array}{lcl} \texttt{g Z} & = & \texttt{Z} \\ \texttt{g (S y)} & = & \texttt{f y} \end{array}$$

the position associated with `f y` is given by $[|\texttt{case x of } \{\texttt{Z} \to \texttt{Z};\ \texttt{S y} \to \texttt{f y}\}|]^1_{2.2} = [|\texttt{case x of } \{\texttt{Z} \to \texttt{Z};\ \texttt{S y} \to \texttt{f y}\}|]^1_{2.2} = (\texttt{1} + \texttt{1}, \Lambda) = (\texttt{2}, \Lambda)$ (i.e., the right-hand side of the second rule), since $[|\texttt{f y}|]^1_\Lambda = (\texttt{1}, \Lambda)$.

We note, however, that there exist some features of Curry that are not covered in our current prototype, like higher-order functions and local definitions.

- Finally, we have implemented a viewer that takes a slicing criterion and an extended redex trail and shows the associated program slice. This viewer first determines the SC-node (according to Definition 30) and then implements function *ds* to collect the set of program positions in the slice. Now, program positions for the considered normalized flat program are first converted to their associated positions in the source Curry program which is then shown to the user by highlighting the expressions whose position belong to the slice.

Let us note that the complexity of the slicing algorithm is not a main concern in this research. Our slicing technique is built on top of an existing tracer. Therefore,

[11] This is reasonable because flat programs obtained by translating source Curry programs without case expressions always fulfill it.

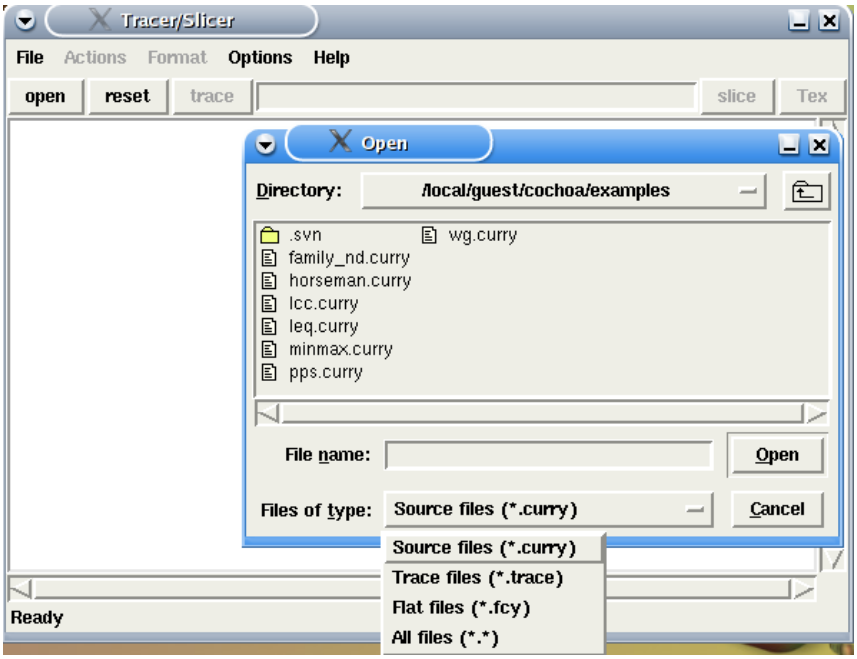


Figure 5.25: Opening a file for debugging

if an efficient implementation of the tracer is available, then also slicing would run efficiently because only a slight extension is necessary (adding program positions to redex trails). Only the viewer, which implements the computation of the SC-node and the set of program positions which are reachable from this node (clearly a linear function), requires some additional cost.

The user interface of our slicing tool has been completely developed using the GUI library for Curry [Hanus, 2000]. In particular, we have developed a new library which implements several widgets for Tcl/Tk (dialog boxes, popup menus, etc).

5.6.1 Architecture

In this section, we describe the structure of our slicing tool. It can be used both for debugging—by automatically extracting the program fragment which contains an error—and for program specialization—by generating executable slices w.r.t. a given slicing criterion.

Tracer

It is introduced in Braßel *et al.* [2004b]. The tracer executes a program using an *instrumented* interpreter. As a side effect of the execution, the *redex trail* of the

computation is stored in a file. The tracer is implemented in Haskell and accepts first-order lazy functional logic programs that can be traced either backwards or forwards. In our slicer, we only slightly extended the original tracer in order to also store in the redex trail the location (the so called *program position*), in the source program, of every reduced expression.

Viewer

Once the computation terminates (or it is aborted by the user in case of a looping computation), the viewer reads the file with the redex trail and allows the user to navigate through the entire computation. The viewer, also introduced in Braßel *et al.* [2004b], is implemented in Curry, a conservative extension of Haskell with features from logic programming including logical variables and non-determinism. The viewer is useful in our approach to help the user to identify the slicing criterion.

Slicer

Given a redex trail and a slicing criterion, the slicer outputs a set of program positions that uniquely identify the associated program slice. The slicing tool is implemented in Curry too, and includes an editor that shows the original program and, when a slice is computed, it also highlights the expressions that belong to the computed slice.

Specializer

Similarly to the slicer, the specializer also computes a set of program positions—a slice—w.r.t. a slicing criterion. However, rather than using this information to highlight a fragment of the original program, it is used to extract an executable slice (possibly simplified) that can be seen as a specialization of the original program for the given slicing criterion.

5.6.2 The Slicer in Practice

The debugging tool starts by allowing the user to select either a source file (`.curry`) or its flat representation (`.fcy`), see Figure 5.25. After a file has been loaded in, its flat representation is shown—with some pretty printing—in the main screen, see Figure 5.26.

Then, the user can start tracing the program by using the tool implemented by Braßel *et al.* [2004b] (as in Section 5.2). For this purpose, the viewer of Braßel *et al.* [2004b] is loaded in and the extended redex trail generated by the meta-interpreter is passed as an argument to it. We have implemented a new viewer that allows the user to see the computed slice.

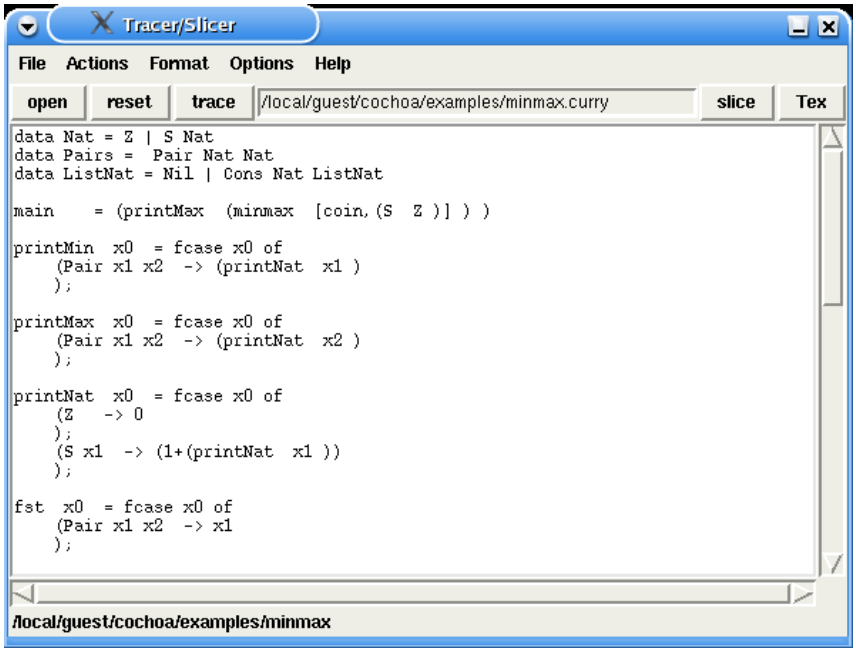


Figure 5.26: Viewing the source program

The slicing process can be initiated by providing a slicing criterion associated to an entry in a given trace. In particular, we have implemented the functions that, given a slicing criterion, compute the associated SC-node as well as the reachable nodes from the SC-node (function *ds*), according to the slicing pattern, and then return their program positions (i.e., the computed slice). The main extensions to this library allow to change the foreground colors of widgets, and provide several new dialog boxes, among others.

Once a point in the trace of the program is selected for slicing, our tool takes on and performs the slicing of the program by first collecting all the reachable nodes in the graph from the selected slicing node (SC-node).

Finally, the source program is shown in the main screen, coloring in gray those parts of the program that do not belong to the slice, as shown in Figure 5.27.

The debugging tool has several other useful features. For instance, it generates both a latex (`.tex`) and a postscript (`.ps`) version of the computed slice, which is useful for printing and discussing errors. Also, it allows the user to set some configuration parameters, such as the preferred postscript viewer, the colors for the displayed slice, etc.

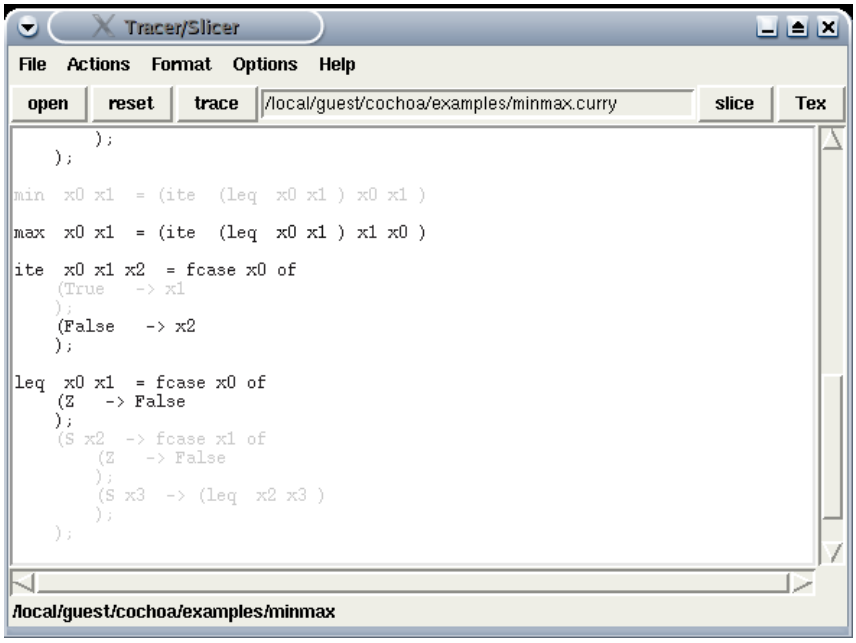


Figure 5.27: Viewing the slice of the source program

An Example of Debugging Session

In order to show the usefulness of the slicer, we now present a debugging session that combines tracing and slicing.

We consider the program shown in Fig. 5.28 (for the time being, the reader can safely ignore the distinction between gray and black text). In this program, the function `main` returns the maximum number of hits of a given web page in a span of time. Function `main` simply calls `minMaxHits` which traverses a list containing the daily hits of a given page and returns a data structure with the minimum and maximum of such a list.

The execution of the program above should return 65, since this is the maximum number of hits in the given span of time. However, the code is faulty and prints 0 instead. We can trace this computation in order to find the source of the error. The tracer initially shows the following top-level trace:

```
0 = main
0 = printMax     (Hits _ 0)
0 = prettyPrint  0
0 = if_then_else True 0 0
0 = 0
```

Each row in the trace has the form $val = exp$, where exp is an expression and val is

```
data T = Hits Int Int

main = printMax (minMaxHits webSiteHits)

webSiteHits = [0, 21, 23, 45, 16, 65, 17]

printMin t = case t of (Hits x _) -> show x
printMax t = case t of (Hits _ y) -> show y

fst t = case t of (Hits x _) -> x
snd t = case t of (Hits _ y) -> y

minMaxHits xs = case xs of
                    (y:ys) -> case ys of
                                  []     -> (Hits y y);
                                  (z:zs) -> let m = minMaxHits (z:zs)
                                            in (Hits (min y (fst m))
                                                     (max y (snd m)))
min x y = if (leq x y) then x else y
max x y = if (leq x y) then y else x

leq x y = if x==0 then False
                  else if y==0 then False else leq (x-1) (y-1)
```

Figure 5.28: Example program `minMaxHits`

the computed value for this expression.

By inspecting this trace, it should be clear that the argument of `printMax` is erroneous, since it contains 0 as the maximum number of hits. Note that the minimum (represented by "_" in the trace) has not been computed due to the laziness of the considered language. Now, if the user selects the argument of `printMax`, the following subtrace is shown:

```
0          = printMax   (Hits _ 0)
(Hits _ 0) = minMaxHits (0:_)
(Hits _ 0) = Hits       _ 0
```

From these subtraces, the user can easily conclude that the evaluation of function `minMaxHits` (rather than its definition) contains a bug since it returns `0` as the maximum of a list without evaluating the rest of the list.

At this point, the tracer cannot provide any further information about the location of the bug. This is where slicing comes into play: the programmer can use the slicer in order to isolate the slice which is responsible of the wrong result; in general, it would

Table 5.1: Benchmark results

benchmark	time	orig size	slice size	reduction (%)
minmax	11 ms.	1.035 bytes	724 bytes	69.95
horseman	19 ms.	625 bytes	246 bytes	39.36
lcc	33 ms.	784 bytes	613 bytes	78.19
colormap	3 ms.	587 bytes	219 bytes	37.31
family$_c$on	4 ms.	1453 bytes	262 bytes	18.03
family$_n$d	2 ms.	731 bytes	289 bytes	29.53
Average	**12 ms.**			**43.73**

be much easier to locate the bug in the slice than in the complete source program. For instance, in this example, the slice would contain the black text in Fig. 5.28. Indeed, this slice contains a bug: the first occurrence of `False` in function `leq` (less than or equal to) should be `True`. Note that, even though the evaluation of `minMaxHits` was erroneous, its definition was correct.

5.6.3 Benchmarking the Slicer

In order to measure the specialization capabilities of our tool, we conducted some experiments over a subset of the examples listed in

`http://www.informatik.uni-kiel.de/~curry/examples`.

Some of the benchmarks are purely functional programs (`horseman`, `family_nd`), some of them are purely logic (`colormap`, `family_con`), and the rest are functional logic programs.

Results are summarized in Table 5.1. For each benchmark, we show the time spent to slice it, the sizes of both the benchmark and its slice, and the percentage of source code reduction after slicing. As shown in the table, an average code reduction of more than 40% is reached.

5.7 Related Work

Despite the fact that the usefulness of slicing techniques has been known for a long time in the imperative paradigm, there are very few approaches to program slicing in the context of declarative languages. In the following, we review the closest to our work. Reps and Turnidge [1996] presented a backward slicing algorithm for first-order functional programs. This work is mainly oriented to perform specialization by computing slices for *part* of a function's output, which is specified by means of a projection. In contrast to our approach, they consider a *strict* language and *static* slicing, which is less useful for debugging (since it is generally less precise). Moreover,

only a *fixed* slicing criterion is considered: the output of the whole program. [Biswas, 1997a,b] considered a higher-order strict functional language and develops a technique for dynamic backward slicing based on labeling program expressions and, then, constructing an execution trace. Again, only a fixed slicing criterion is considered (although the extension to more flexible slicing criteria is discussed).

Liu and Stoller [2003] defined an algorithm for static backward slicing in a first-order strict functional language, where their main interest is the removal of dead code. Although the underlying semantics is different (lazy vs. strict), our slicing patterns are inspired by their *liveness patterns*.

Hallgren [2003] presented (as a result of the Programatica project) the development of a slicing tool for Haskell. Although there is little information available, it seems to rely on the construction of a graph of functional dependences (i.e., only function names and their dependences are considered). Therefore, in principle, it should produce bigger—less precise—slices than our technique.

Rodrigues and Barbosa [2005] introduced a novel approach to slicing functional programs which is based on the use of projection (for backward slicing) and hiding (for forward slicing) functions as slicing criteria. When these functions are composed with the original program, they lead to the identification of the intended slice. In contrast to our approach, no data structure storing program dependences is built.

The closest approach is the *forward* slicing technique for lazy functional logic programs introduced by Vidal [2003]. In this approach, a slight extension of a partial evaluation procedure for functional logic programs is used to compute program slices. Since partial evaluation naturally supports partial data structures, the distinction between static and dynamic slicing is not necessary (the same algorithm can be applied in both cases). This generality is a clear advantage of [Vidal, 2003], but it often implies a loss of precision.

Our slicing technique is based on an extension of the redex trail introduced by Braßel *et al.* [2004b]. This work introduces an instrumented operational semantics for lazy functional logic programs that generates a redex trail of a computation. This redex trail shares many similarities with the ART model of Wallace *et al.* [2001] but there is an important difference:

- The ART model of Wallace *et al.* [2001] is defined in terms of a program transformation. In contrast, Braßel *et al.* [2004b] present a direct, semantics-based definition of redex trails. This improves the understanding of the tracing process and allows one to prove properties like "every reduction occurring in a computation can be observed in the trail". Chitil [2001] presents a first approach to such a direct definition by introducing an augmented small-step operational semantics for a core of Haskell that generate ARTs. However, this work does not consider correctness issues.

Chitil [2005] has proposed a source-based trace exploration for the higher-order lazy functional language Haskell which combines ideas from algorithmic debugging, traditional stepping debuggers, and dynamic slicing. Although this proposal shares many similarities with our approach, no correctness results are provided (since no semantics-based definition of redex trails is available).

Part IV

Algorithmic Debugging

Chapter 6

Combining Program Slicing and Algorithmic Debugging

As discussed in Chapter 1, program slicing and algorithmic debugging are two of the most relevant debugging techniques for declarative languages. In this chapter, we show for functional languages how the combination of both techniques produces a more powerful debugging scheme that reduces the number of questions that programmers must answer to locate a bug.

Part of the material of this chapter has been presented in the *8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming* (PPDP'06) celebrated in Venice (Italy) in 2006 [Silva and Chitil, 2006].

6.1 Introduction

In this chapter we combine algorithmic debugging [Shapiro, 1982] with program slicing [Weiser, 1984]. By combining algorithmic debugging and program slicing the debugging process becomes more interactive, allowing the programmer to provide more information to the debugger. This information reduces the number of questions and guides the process, thus making the sequence of questions semantically related.

The main contributions of this chapter can be summarised as follows:

1. We combine both *program slicing* and *algorithmic debugging* in a single, more powerful framework in the context of functional programming.

2. We adapt the well-known program slicing concepts of slice, slicing criterion, complete slice, and so on to the augmented redex trail, a trace structure that can be the basis for both algorithmic debugging and program slicing, and we define and prove properties for this formalism.

3. We introduce an algorithm for slicing execution trees which is proven complete, in the sense that every slice produced contains all the nodes of the tree that are needed to locate a bug.

The rest of the chapter is organised as follows. In the next section we outline how algorithmic debugging can be improved by letting the programmer provide more information. In Section 6.4 we define the augmented redex trail and in Section 6.5 we define which kind of slices of an augmented redex trail we are interested in. In Section 6.6 we combine these slicing concepts with algorithmic debugging and introduce an algorithm to compute the slices we need. Then, we give an example of a debugging session that uses our technique in Section 6.7. In Section 6.8 we discuss related debugging techniques.

6.2 The Idea: Improved Algorithmic Debugging

As shown in Example 1, a conventional algorithmic debugger will ask the programmer whether an equation

$$foo\ a_1 \ldots a_n\ =\ e$$

is correct or wrong; the programmer will answer "*yes*", "*no*" or "*maybe*" depending on whether he thinks that it is respectively correct, wrong or he just does not know.

Our main objective in this research is to be able to get more information from the programmer and consequently reduce the number of questions needed to locate a bug. For this purpose, we consider the following cases:

1. **The programmer answers "*maybe*".** This is the case when the programmer cannot provide any information about the correctness of the equation.

2. **The programmer answers "*yes*".** This is the case when the programmer agrees with both the complete result and all the expressions in the left-hand side of the equation.

3. **The programmer answers "*no*".** The programmer can disagree with an equation and provide the system with additional information related to the wrong parts of the equation. This happens either because the result is wrong, or because some precondition of the equation has been violated (i.e., some parts of the left-hand side were not expected).

   - When the result is wrong, the programmer could say just "*no*", but he could also be more specific. He could exactly point to the parts of the result which are wrong. For instance, in the equation above, he could specify that only a subexpression of e is wrong, and the rest of the result

is correct. This information is useful because it could help the system to avoid questions about correct parts of the computation.

- The programmer could detect that some expression inside the arguments of the left-hand side of the equation should not have been computed. For instance, consider the equation *insert* $'b'$ "*cc*" = "*bcc*", where function *insert* inserts the first argument in a list of mutually different characters (the second argument). Clearly the result of the equation is correct; but the programmer could detect here that the second argument should not have been computed. This means that even for a correct equation the programmer could provide the system with information about bugs. In this case, the programmer could mark the second argument ("*cc*") as wrong (i.e., *inadmissible* [Naish, 1997b] for this function because it violates a precondition).
- Finally, both, the result and the arguments, could be correct, but the function name could be wrong[1], in the sense that it should have never been called. In this case, the programmer could mark the function name as wrong.

The information provided by the programmer allows the system to slice the execution tree, thus reducing the number of questions. The more information he provides, the smaller the execution tree becomes. For instance, the slice computed if the programmer answers "*no*" is usually larger than the one computed if he answers "*no, and this function should not have been computed*". Of course, as in conventional algorithmic debugging, the programmer may only give information about the wrong expressions he is able to detect. In the worst case, he can always answer "*yes*", "*no*" or "*maybe*".

In summary, in order to provide information about errors, the programmer can specify that some part of the result or the arguments, or the function name is wrong. The slicing process needed for each case is different.

In this chapter we consider higher-order term rewriting systems that we introduce in the next section.

6.3 Computation Graphs

In this chapter, we represent terms and expressions as graphs; and we represent computations with graphs rewriting. Therefore, we need to formally define this formalism. Some of the definitions in this section are taken from [Chitil and Luo, 2006].

[1] As we consider a higher-order language, this is a special case of a wrong argument, the function name being the first (wrong) part of the function application.

Since we consider higher-order TRSs, we only need to define the domains of *variables* $\mathcal{X}$ and *constructor terms* $\mathcal{C}$ (we do not distinguish between operation and constructor symbols). Except the opposite is stated, we keep the same definitions and notation of Chapter 2.

A *term* is a variable, a constructor, or an application of two terms. A *rewrite rule* is a pair of terms, written by $l \to r$, such that $root(l) \notin \mathcal{X}$ and $Var(r) \subseteq Var(l)$. A finite set of rewrite rules is a *higher-order term rewriting system.*

6.4 The Augmented Redex Trail

In the first phase of algorithmic debugging the program is executed and the execution tree is built. However, the execution tree is not actually built directly. Direct construction of the execution tree during the computation would not be very efficient; in particular, the nodes of the execution tree are labeled with equations that contain many repeated subexpressions and, for a lazy functional language, the tree nodes are constructed in a rather complex order. So instead of the final execution tree a more elaborated trace structure is constructed that avoids duplication of expressions through sharing.

The *augmented redex trail (ART)* [Sparud and Runciman, 1997; Wallace *et al.*, 2001] is such a trace structure. The ART is used in the Haskell tracer Hat [Sparud and Runciman, 1997; Wallace *et al.*, 2001] and has similarities with the trace structure constructed by the earlier algorithmic debugger Freja [Nilsson, 1998]. The ART has been designed so that it can be constructed efficiently and after its complete construction the execution tree needed for algorithmic debugging can be constructed from it easily. In addition, an ART contains more information than an execution tree. An execution tree can be constructed from an ART but not vice versa. The ART supports several other views of a computation besides the execution tree. Nonetheless an ART is not larger than an execution tree with sharing. An ART is a complex graph of expression components that can represent both eager or lazy computations. In this chapter we will use the following definition of ART, which is different from the one introduced by Braßel *et al.* [2004b] and used in Chapter 5.

Definition 40 (Augmented redex trail) *A* ***node*** n *is a sequence of the letters* l*,* r *and* t*; which stands respectively for left, right and top. There is also a special node* $\perp$ *(bottom).*

A ***node expression*** $\mathcal{T}$ *is a function symbol, a data constructor, a node or an application of two nodes.*

A ***trace graph*** *is a partial function* $\mathcal{G} : n \mapsto \mathcal{T}$ *such that its domain is prefix-closed (i.e., if* $ni \in dom(\mathcal{G})$*, then* $n \in dom(\mathcal{G})$*), but* $\perp \notin \mathcal{D}om(\mathcal{G})$*. We often regard a trace graph as a set of tuples* $\{(n, \mathcal{G}(n)) \mid n \in dom(\mathcal{G})\}$

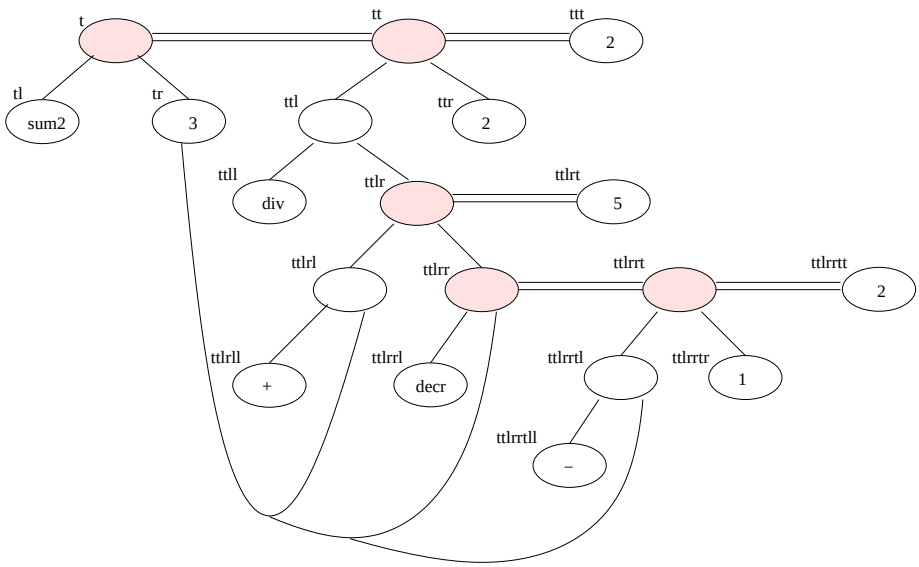


Figure 6.1: Example of ART

A node can ***represent*** *several terms, because any nodes n and nt are considered equal. A node* ***approximates*** *a term if it represents this term, except that some subterms may be replaced by* $\perp$*.*

A node n plus a node nt represent a ***reduction step*** *of the program, if node n represents the redex and node nt approximates the reduct, and for each variable of the corresponding program rule there is one shared node.*

A trace graph $\mathcal{G}$ *where node* ϵ *(empty sequence) approximates the start term M and every pair of nodes n and nt represents a reduction step with respect to the program P, is an* ***augmented redex trail (ART)*** *for start term M and program P.*

The sequence of the letters l, r and t of every node is used as identifier, thus every node has a unique sequence of these letters. Any application of two terms t_1 and t_2 represented with the sequence x, has an arc to these terms whose nodes are identified respectively with the sequences xl and xr; except if the terms were already in the EDT. When a term t_1 represented by a node x is reduced to another term t_2 through a single reduction step, t_2 is identified with the sequence xt. We often call this kind of nodes ended with t as reductions. Finally, a node labeled with $\perp$ represents a non-evaluated term.

For example, consider the graph depicted in Figure 6.1. This graph is part of the ART generated from the program in Figure 1.1. In particular, it corresponds to the function call "$sum2$ 3" producing the result "2". As specified in Definition 40,

each node is identified by a sequence of letters 't', 'l' and 'r'[2]. Every node contains the expression it represents, except for a node which represents an application of two expressions connected to it by simple lines. A double line represents a single reduction step. For instance, node $ttlrrt$ represents the function call "$3-1$" that has been reduced to "2".

In the following, a (atom) denotes a function symbol or constructor, and n, m and o denote nodes. We usually write $n \in \mathcal{G}$ instead of $n \in \mathcal{D}om(\mathcal{G})$ when it is clear from the context. We also say that a node $n \in \mathcal{G}$ is reduced iff $nt \in \mathcal{G}$. Given a node $n \in \mathcal{G}$ where $\mathcal{G}(n)$ is an application of nodes $m \cdot o$, we respectively refer to m and o as $\mathit{left}(n)$ and $\mathit{right}(n)$.

The parent of a node is the node which created it, that is, given a reduction step $t \to s$, and a node n representing a subterm of s, then, the parent of n is the node which represents t. Formally:

$$\begin{aligned} \mathit{parent}(nl) &= \mathit{parent}(n) \\ \mathit{parent}(nr) &= \mathit{parent}(n) \\ \mathit{parent}(nt) &= n \end{aligned}$$

The recursive application of function *parent* produces the ancestors of a node:

$$\mathit{ancestors}(n) \quad = \quad \begin{cases} \{m\} \cup \mathit{ancestors}(m) & \text{if } m = \mathit{parent}(n) \text{ defined} \\ \varnothing & \text{otherwise} \end{cases}$$

Finally, to extract the most evaluated form represented by a node, function *mef* is defined by:

$$\mathit{mef}(\mathcal{G}, n) = \begin{cases} a & \text{if } \mathcal{G}(n) = a \\ \mathit{mef}(\mathcal{G}, m) & \text{if } \mathcal{G}(n) = m \\ \mathit{mef}(\mathcal{G}, m) \cdot \mathit{mef}(\mathcal{G}, o) & \text{if } \mathcal{G}(n) = m \cdot o \text{ and } nt \notin \mathcal{G} \\ \mathit{mef}(\mathcal{G}, nt) & \text{if } nt \in \mathcal{G} \end{cases}$$

Intuitively, function *mef* traverses the graph $\mathcal{G}$ following reduction edges (n to nt) as far as possible. Note that this function is similar to function *val* introduced in Definition 25, but *mef* is defined for a higher-order language.

Now we see how easily the execution tree is obtained from the ART. We first formally define execution tree.

Definition 41 (execution tree) *Let $\mathcal{G}$ be an ART. The execution tree $\mathcal{E}$ associated to $\mathcal{G}$ is a tree whose nodes are labeled with equations constructed with the auxiliary*

[2]For easier reading we here call the first node t; within the ART of the complete computation the node is actually $trtrtrtrlr$.

function. equ is used to extract expressions from the nodes of the ET, and it is defined as follows:

$$equ(\mathcal{G}, n) = \begin{cases} a & \text{if } \mathcal{G}(n) = a \\ mef(\mathcal{G}, m) \cdot mef(\mathcal{G}, o) & \text{if } \mathcal{G}(n) = m \cdot o \end{cases}$$

The equations have the form $l = r$ such that:

- *the root node is $equ(\mathcal{G}, \epsilon) = mef(\mathcal{G}, t)$, if $t \in \mathcal{G}$,*
- *for each reduction step n to nt in $\mathcal{G}$ there is a node $equ(\mathcal{G}, n) = mef(\mathcal{G}, nt)$ in $\mathcal{E}$, and*
- *for each pair of nodes $n'_1, n'_2 \in \mathcal{E}$, with n'_1 labeled $equ(\mathcal{G}, n_1) = r_1$ and n'_2 labeled $equ(\mathcal{G}, n_2) = r_2$; n'_1 is the parent of n'_2 iff $n_2 = n_1 to$ where $o \in \{r, l\}^*$.*

The reduced nodes of the ART become the nodes of the execution tree. The function *parent* expresses the child–parent relationship between these nodes in the execution tree. The equation of a node n can be reconstructed by applying the function *mef* to $left(n)$ and $right(n)$ to form the left-hand side, and to nt to form the right-hand side. For instance, for the ART in Figure 6.1 the algorithmic debugger produces the execution tree in Figure 6.2. It contains five questions, one for each reduction step (reduced nodes have been shaded). For example, the question asked for node *ttlr* is "$3 + 2 = 5$?".

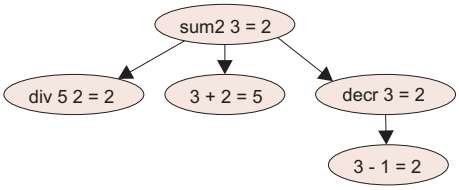


Figure 6.2: Execution tree of the ART in Figure 6.1

Definition 42 (nodes associated to expressions) *Let $\mathcal{G}$ be an ART and $\mathcal{E}$ the execution tree associated to $\mathcal{G}$. A node $n \in \mathcal{G}$ is associated to an expression $e \in \mathcal{E}$ iff n represents e and $\nexists\, n' \in \mathcal{G} \mid n' = no$, $o \in \{r, l, t\}^*$ and n' represents e.*

The actual implementation of ARTs includes more detailed information than described here; however, this extra information is not relevant for the purposes of this research. As an example of a complete ART, the execution tree in Figure 1.3 was generated from the graph shown in Figure 6.3.

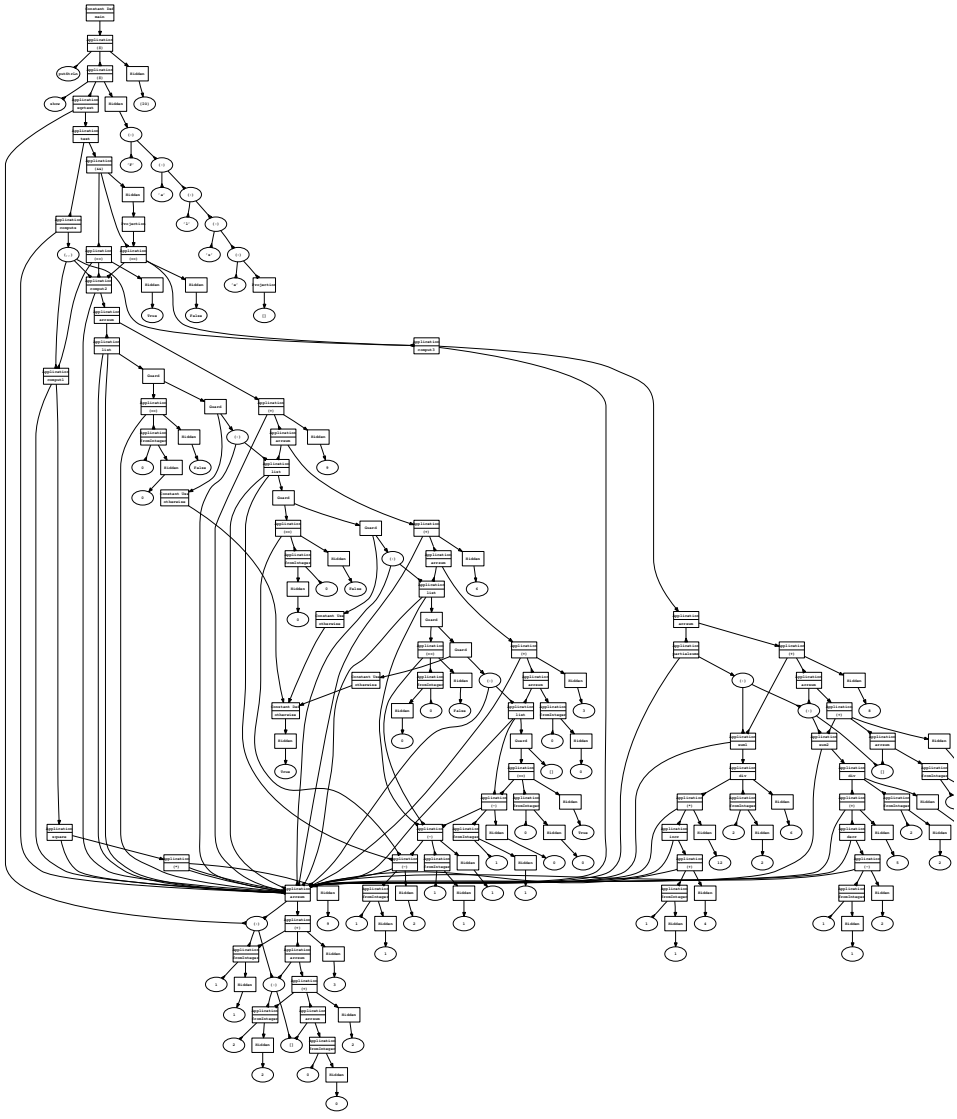


Figure 6.3: ART of the program in Figure 1.1

6.5 Slicing the ART

Program slicing techniques have been based on tree-like data structures such as the *program dependence graphs* (see Chapter 1). In the context of execution trees we want to define the slicing mechanism over their underlying ART. This provides us with two advantages:

1. At this level we have much more information about the computation which allows the production of more precise slices. It is straightforward to extract an execution tree from the sliced ART [Chitil, 2005].

2. When we define the slicing technique at the ART level, it can be used for any other purpose (e.g., for proper program slicing which has not been defined for these graphs, for program specialisation, etc.).

Now, we define the notion of *slice* and some properties in the context of ARTs.

Definition 43 (slice) *Let $\mathcal{G}$ be an ART. A slice of $\mathcal{G}$ is a set of nodes $\{n_1, \ldots, n_k\} \subseteq \mathcal{D}om(\mathcal{G})$.*

To compute slices w.r.t. a particular node of interest we need to identify which nodes in the computation have influenced this node of interest. We introduce the following definition:

Definition 44 (Influence between nodes) *Let $\mathcal{G}$ be an ART for start term M and program P. Node n_1 influences node n_2, written $n_1 \approx\!\!\!\!\rightarrow n_2$ iff for all ARTs $\mathcal{G}'$ for start term M and program P with $n_2 \in \mathcal{G}'$ we have $n_1 \in \mathcal{G}'$.*

Note that we have not fixed any evaluation order for constructing ARTs, but functional programs are deterministic. Hence different ARTs differ in which parts of the computation are how much evaluated.

Influence is transitive and reflexive.

Lemma 45 (Transitivity and reflexivity $\approx\!\!\!\!\rightarrow$) *Let $\mathcal{G}$ be an ART for start term M and program P. For nodes n_1, n_2 and n_3 in $\mathcal{G}$:*

- *$n_1 \approx\!\!\!\!\rightarrow n_2$ and $n_2 \approx\!\!\!\!\rightarrow n_3$ implies $n_1 \approx\!\!\!\!\rightarrow n_3$*
- *$n_1 \approx\!\!\!\!\rightarrow n_1$*

Proof. We consider influence for a graph $\mathcal{G}$ with start term M and program P. Let $\mathcal{G}'$ be any ART for the same start term and program such that $n_3 \in \mathcal{G}'$. With $n_2 \approx\!\!\!\!\rightarrow n_3$ follows that node n_2 is in $\mathcal{G}'$. Then with $n_1 \approx\!\!\!\!\rightarrow n_2$ follows that node n_1 is in $\mathcal{G}'$. So according to Definition 44, $n_1 \approx\!\!\!\!\rightarrow n_3$. Reflexivity trivially holds by definition of influence.

□

We perform program slicing on the ART $\mathcal{G}$ without knowing the program P whose computation is described by $\mathcal{G}$. So we cannot compute other ARTs $\mathcal{G}'$ for start term M and program P; furthermore, there is a large, possibly infinite number of such $\mathcal{G}'$s. Our aim is to find an algorithm that obtains a slice from the given ART $\mathcal{G}$ ensuring completeness (i.e., all nodes that influence the node of interest belong to the slice). Therefore we have to approximate the slice that influences a given node:

Definition 46 (Weak influence between nodes) *Given an ART $\mathcal{G}$. Reduction n_1 weakly influences node n_2, written $n_1 \rightsquigarrow n_2$ iff either*

1. *$n_2 = n_1 o$ for some $o \in \{l, r, t\}^*$, or*
2. *there exist $m, o \in \{l, r, t\}^*$ such that $m \in ancestors(n_2)$, $n_1 = mo$ and o does not start with t.*

The idea is to include all prefixes of the given node (case 1) and all nodes that may be needed for the reduction step of an ancestor. Therefore, if an ancestor of n_2 is an application, then, all the nodes involved in the arguments of this application do weakly influence n_2 (case 2).

Lemma 47 (Transitivity and reflexivity of $\rightsquigarrow$) *Let $\mathcal{G}$ be an ART for start term M and program P. For nodes n_1, n_2 and n_3 in $\mathcal{G}$:*

- *$n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow n_3$ implies $n_1 \rightsquigarrow n_3$*
- *$n_1 \rightsquigarrow n_1$*

Proof. Reflexivity is trivial. For transitivity let $n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow n_3$. We perform case analysis according to the two conditions of Definition 46:

- $n_2 = n_1 o_1$ and $n_3 = n_2 o_2$. Then $n_3 = n_1 o_1 o_2$, so $n_1 \rightsquigarrow n_3$.
- $n_2 = n_1 o_1$ and $n_2 = m o_2$ with $m \in ancestors(n_3)$. If $m = n_1 p$, then $n_1 p p' = n_3$ because m is an ancestor of n_3 and thus $n_1 \rightsquigarrow n_3$. Otherwise, $n_1 = mp$. Then $n_2 = mpo_1$. Since o_2 does not start with t, p cannot start with t. So $n_1 \rightsquigarrow n_3$ following the second part of Definition 46.
- $n_1 = mo_1$ where $m \in ancestors(n_2)$ and $n_3 = n_2 o_2$. Since $ancestors(n_2) \subseteq ancestors(n_3)$, we have $m \in ancestors(n_3)$. So $n_1 \rightsquigarrow n_3$.
- $n_1 = m_1 o_1$ where $m_1 \in ancestors(n_2)$ and $n_2 = m_2 o_2$ where $m_2 \in ancestors(n_3)$. So $m_1 o' = n_2 = m_2 o_2$. If $m_2 = m_1 o$, then $m_1 \in ancestors(n_3)$, so $n_1 \rightsquigarrow n_3$. Otherwise, $m_1 = m_2 o$. Since o_2 does not start with t, also o does not start with t. Furthermore, $n_1 = m_2 o o_1$ and $m_2 \in ancestors(n_3)$. So $n_1 \rightsquigarrow n_3$.

□

Definition 48 (Slice of influence) *Given an ART $\mathcal{G}$ and a node $n \in \mathcal{G}$ the slice of $\mathcal{G}$ w.r.t. node n is $slice(\mathcal{G}, n) = \{m \in \mathcal{G} \mid m \rightsquigarrow n\}$.*

Proposition 49 (Slice of influence)
$\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$ is an ART for the same start term and program as $\mathcal{G}$.

In order to prove this proposition we need first the following auxiliary lemmas:

Lemma 50 (Prefix-closed) *Let $\mathcal{G}$ be an ART with $n \in \mathcal{G}$. Then $slice(\mathcal{G}, n)$ is prefix-closed.*

Proof. Let us assume that $mo \in slice(\mathcal{G}, n)$, i.e., $mo \rightsquigarrow n$. Because m is a prefix of mo, it follows with the first condition in Definition 46 that $m \rightsquigarrow mo$. Then, by Lemma 47, $m \rightsquigarrow n$. So $m \in slice(\mathcal{G}, n)$.

□

Lemma 51 (Reductions in slice) *Let $\mathcal{G}$ be an ART with $n \in \mathcal{G}$. For any reduction step m to mt in $\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$, node m represents the same redex and node mt approximates the same reduct as in $\mathcal{G}$.*

Proof. We show that all relevant nodes exist unchanged in $\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$: From the second part of Definition 46, it follows that all nodes reachable from m in $\mathcal{G}$ are also in $slice(\mathcal{G}, n)$. Moreover, by Definition 48 $\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$ is a subset of $\mathcal{G}$. Therefore, m represents the same redex as in $\mathcal{G}$. The fact that $mt \in slice(\mathcal{G}, n)$ suffices already for mt to approximate the reduct of m; since $\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\}$ is a subset of $\mathcal{G}$ it must be the same reduct as in $\mathcal{G}$.

□

Now, we can prove Proposition 49:
Proof. According to lemma 50 $slice(\mathcal{G}, n)$ is prefix-closed; and according to lemma 51 for every node $nt \in \mathcal{D}om(\{(m, \mathcal{G}(m)) \mid m \in slice(\mathcal{G}, n)\})$, the node n represents the redex and node nt approximates the reduct of the reduction step n to nt.

□

Weak influence approximates influence:

Proposition 52 (Influence between nodes)
Let $\mathcal{G}$ be an ART with $n \in \mathcal{G}$. Then $slice(\mathcal{G}, n) \supseteq \{m \in \mathcal{G} \mid m \rightsquigarrow n\}$.

Proof. We prove that $m \not\rightsquigarrow n$ implies $m \not\rightsquigarrow n$. According to Proposition 49 $slice(\mathcal{G}, n)$ is an ART for the same start term and program as $\mathcal{G}$. Moreover, we know that $n \in slice(\mathcal{G}, n)$ because the reflexivity property of Lemma 47. From $m \not\rightsquigarrow n$, it follows that $m \notin slice(\mathcal{G}, n)$. Hence, according to Definition 44, $m \not\rightsquigarrow n$ because there exists an ART to which n belongs and m does not.

□

So we will compute slices of an ART $\mathcal{G}$ that weakly influence a node n.

6.6 Combining Program Slicing and Algorithmic Debugging

The combination of algorithmic debugging and program slicing can lead to a much more accurate debugging session. An algorithmic debugger can interpret information from the programmer about the correctness of different parts of equations, and, by means of a program slicer, use this information to slice execution trees, thus reducing the number of questions needed to find a bug. In this section, we show how program slicing can be combined with algorithmic debugging.

6.6.1 The Slicing Criterion

To define the slicing technique we need to define first a slicing criterion in this context which specifies the information provided by the programmer. During algorithmic debugging, the programmer is prompted and asked about the correctness of some series of equations such as the following:

$$\begin{array}{lcl} f_1\ v_1^1 \ldots v_i^1 & = & val_1 \\ f_2\ v_1^2 \ldots v_j^2 & = & val_2 \\ \ldots & & \\ f_n\ v_1^n \ldots v_k^n & = & val_n \end{array}$$

Then, the programmer determines for each equation whether it is correct or wrong. In particular, when the programmer identifies a wrong (sub)expression inside an equation, he may produce a slice of the execution tree w.r.t. this wrong expression. Therefore, from the point of view of the programmer, a slicing criterion is a wrong (sub)expression: he should be able to specify (i.e., mark) which parts of the equations shown during algorithmic debugging are wrong. However, from the point of view of

the debugger, the internal representation of the execution tree is an ART and thus a slicing criterion should be a pair of nodes $\langle m, n\rangle$ in the ART denoting where the slice starts (m) and where it ends (n).

To automatically produce a slicing criterion from the expression marked by the programmer, we use slicing patterns. Wee need to redefine the domain *Pat* of slicing patterns of Chapter 5 so that higher-order is considered. It is defined as follows:

$$\pi \in Pat \ ::= \ \bot \mid \top \mid \pi_1\pi_2$$

where $\bot$ denotes a subexpression that is irrelevant and $\top$ a relevant subexpression.

Note that a (partially or completely evaluated) value is an instance of a slicing pattern where atoms have been replaced by $\bot$ and $\top$. Patterns are used to distinguish between correct and wrong parts of equations, and hence to find the nodes of the ART which could be the cause of the bug. For instance, once the programmer has selected some (wrong) subexpression, the system automatically produces a pattern from the whole expression where the subexpression marked by the programmer have been replaced by $\top$, and the rest of subexpressions have been replaced by $\bot$. Because algorithmic debugging finds a single bug at a time, we only consider a single expression marked in π; different wrong expressions could be produced by different bugs.

Example 17 *Consider the following equation from the execution tree of Figure 1.3:* `computs 3 = (9,9,8)`

If the programmer marks `8` *as wrong, the pattern generated for the expression* `(9,9,8)`*, internally represented as* `(((T·9)·9)·8)`*, would be* `(⊥· ⊤)`*, where* `T` *is the n-ary tuple constructor.*

As shown in the previous example, the pattern allows us to discard those subexpressions which are correct. With the information provided by the pattern it is easy to find the nodes in the ART which are associated with wrong expressions. The following definition introduces function *patNodes* which identifies the node in the ART associated with the wrong expression marked by the programmer.

Definition 53 (*patNodes*) *Given an ART $\mathcal{G}$, a pattern π for an expression M and a node n such that $mef(\mathcal{G}, n) = M$, $patNodes(\mathcal{G}, \pi, n)$ finds in the ART the node associated with the wrong expression specified in π. patNodes is defined as follows:*

$$patNodes(\mathcal{G}, \pi, n) = \begin{cases} \{\} & \text{if } \pi = \bot \\ patNodes(\mathcal{G}, \pi, nt) & \text{if } \pi \neq \bot \text{ and } nt \in \mathcal{G} \\ patNodes(\mathcal{G}, \pi, m) & \text{if } \pi \neq \bot,\ nt \notin \mathcal{G} \text{ and } \mathcal{G}(n) = m \\ \{n\} & \text{if } \pi = \top,\ nt \notin \mathcal{G} \text{ and } \mathcal{G}(n) \neq m \\ patNodes(\mathcal{G}, \pi_1, n_1) & \text{if } \pi = \pi_1 \cdot \pi_2,\ nt \notin \mathcal{G} \\ \cup\ patNodes(\mathcal{G}, \pi_2, n_2) & \text{and } \mathcal{G}(n) = n_1 \cdot n_2 \end{cases}$$

After $patNodes(\mathcal{G}, \pi, n)$ has found in the ART the node associated with the wrong expression specified in π, the slicer can produce a slice for this node which contains all those nodes in $\mathcal{G}$ that could influence the wrong expression.

For instance, given the equation $foo\ M = N$, the programmer should be able to specify that he wants to produce a slice from either foo, some subexpression of M or some subexpression of N. Therefore, we distinguish between three possible ways of defining a slicing criterion:

- The programmer marks a subexpression of N and the associated pattern π is generated; then, the algorithmic debugger computes the tuple $\langle m, n\rangle$ where $\{n\} = patNodes(\mathcal{G}, \pi, m)$ such that $mef(\mathcal{G}, left(m))\ mef(\mathcal{G}, right(m)) = foo\ M$ and, thus, m must be a reduced node.
- The programmer marks a subexpression of M and the associated pattern π is generated; then, the algorithmic debugger computes the tuple $\langle \epsilon, n\rangle$ where $\{n\} = patNodes(\mathcal{G}, \pi, m)$ such that $mef(\mathcal{G}, m) = M$ is the second argument of the application $foo\ M$.
- The programmer marks the function name foo, and the algorithmic debugger produces a tuple $\langle \epsilon, n\rangle$ where $mef(\mathcal{G}, n) = foo$ is the first argument of the application $foo\ M$.

We need to distinguish between errors in the right-hand side of an equation and errors in the left-hand side because the slices for them are essentially different. In the former case, we only care about new nodes, because we trust the arguments of the function call in the left-hand side of the equation and, thus, all the nodes which were computed before this function call could not cause the bug; however, in the later case, the error could be in any previous node that *could have influenced* (rather than *had influenced*) the value of the incorrect argument or function name.

Definition 54 (Slicing criterion)
Let $\mathcal{G}$ be an ART. A slicing criterion for $\mathcal{G}$ is a tuple $\langle m, n\rangle$; such that $m, n \in \mathcal{G}$ and $n = mo$, where $o \in \{l, r, t\}^*$.

A slicing criterion points out two nodes in the graph. In contrast, slicing patterns can select a set of nodes of interest. Therefore, from a tuple $\langle m, s\rangle$ generated from a slicing pattern, we produce a set of slicing criteria $\{\langle m, n\rangle\}$ such that $n \in s$. Later, we compute the slice associated to $\langle m, s\rangle$ from the union of the slices of $\{\langle m, n\rangle\}$.

The two nodes specified in a slicing criterion delimit the area of the graph where the bug must be. The objective of the slicer is to find these nodes and collect all the nodes which are weakly influenced by the first node and that weakly influence the second node. This kind of slice is known in the literature as a *chop* (see Section 1.2.6):

$$collect(\mathcal{G},m,n) = \begin{cases} \underline{path(\mathcal{G},m,n)} & \text{if } parent(m) = parent(n) \\ & \text{or if } parent(m) \text{ and } parent(n) \\ & \text{are undefined} \\ \{p\} \cup collect(\mathcal{G},m,p) & \text{otherwise, where } parent(n) = p \\ \cup\ subnodes(\mathcal{G},p)\ \underline{\cup\ path(\mathcal{G},pt,n)} & \end{cases}$$

$$subnodes(\mathcal{G},n) = \begin{cases} subn(\mathcal{G},m) \cup subn(\mathcal{G},o) & \text{if } \mathcal{G}(n) = m \cdot o, m = nl \text{ and } o = nr \\ subn(\mathcal{G},m) & \text{if } \mathcal{G}(n) = m \cdot o, m = nl \text{ and } o \neq nr \\ subn(\mathcal{G},o) & \text{if } \mathcal{G}(n) = m \cdot o, m \neq nl \text{ and } o = nr \\ \{\} & \text{otherwise} \end{cases}$$

$$subn(\mathcal{G},n) = \begin{cases} \{n\} \cup subnodes(\mathcal{G},n) \cup subn(\mathcal{G},nt) & \text{if } nt \in \mathcal{G} \\ \underline{\{n\} \cup\ subnodes(\mathcal{G},n)} & \text{if } nt \notin \mathcal{G} \end{cases}$$

$$\underline{path(\mathcal{G},n,n')} = \begin{cases} \underline{\{n\}} & \text{if } n = n' \\ \underline{\{n'\} \cup path(\mathcal{G},n,m)} & \text{if } n' = mi,\ m \in \{l,r,t\}^* \text{ and } i \in \{l,r,t\} \end{cases}$$

Figure 6.4: Function *collect* to slice ARTs

Definition 55 (Chop of influence)
Let $\mathcal{G}$ be an ART and $\langle m,n\rangle$ a slicing criterion for $\mathcal{G}$. The chop $\mathcal{S}$ of $\mathcal{G}$ w.r.t. $\langle m,n\rangle$ is $chop(\mathcal{G},m,n) = \{n' \in \mathcal{G} \mid m \rightsquigarrow n' \text{ and } n' \rightsquigarrow n\}$.

Obviously $slice(\mathcal{G},n) = chop(\mathcal{G},\epsilon,n)$.

6.6.2 The Slicing Algorithm

Figure 6.4 shows the formal definition of the slicing algorithm for ARTs. For the sake of convenience, we give in this figure two versions of the algorithm that can be differentiated with the underlining. On the one hand, this algorithm is able to compute a slice from an ART. On the other hand, we present a conveniently modified version which only computes those nodes that are useful during algorithmic debugging. We first describe the algorithm to slice ARTs (for the time being the reader can ignore the underlining):

Function *collect* computes $chop(\mathcal{G},m,n)$. It collects all the nodes for which m is a prefix that belong to one of the following sets:

- The nodes which are in the path from m to n,

- The ancestors of n, and
- All the nodes which participated in the evaluation of the arguments of the ancestors of n (we call such a set of nodes the *subnodes*).

Function *collect* proceeds by identifying the parent of the incorrect node (i.e., the function that introduced it) and its subnodes; then, it recursively collects all the nodes which could influence the parent (all its ancestors). Functions *subnodes* and *subn* are used to compute all those nodes which participated in the evaluation of the arguments of a given application node n. Finally, function *path* computes the path between two given nodes.

The algorithm presented in Figure 6.4 is a program slicing technique for ARTs. However, the slices it produces contain nodes which are useless for algorithmic debugging. In algorithmic debugging we are only interested in reductions, i.e., reduced nodes, because they are the only nodes which yield questions during the algorithmic debugging process [Chitil, 2005]. For this reason, we also introduce a slightly modified version which only computes those nodes from the ART which are needed to produce the execution trees used during algorithmic debugging.

Firstly, only reduced nodes must be computed, thus the underlined expressions in Figure 6.4 should be omitted from the algorithm. And, secondly, we must ensure that the produced slice is not empty. This is ensured by the fact that the slicing criterion defined for algorithmic debugging always takes a reduced node as the starting point. Therefore, the slice is never empty because at least the (reduced) node associated with the left-hand side of the wrong equation is included in the slice.

6.6.3 Completeness of the Slicing Algorithm

The following result states the completeness of the slicing algorithm:

Theorem 56 (Completeness) *Given an ART $\mathcal{G}$ and a slicing criterion $\langle m, n\rangle$ for $\mathcal{G}$, $collect(\mathcal{G}, m, n) = chop(\mathcal{G}, m, n)$.*

Proof. We prove the theorem by showing that:

1. $chop(\mathcal{G}, m, n) \subseteq collect(\mathcal{G}, m, n)$ and
2. $collect(\mathcal{G}, m, n) \subseteq chop(\mathcal{G}, m, n)$

We prove first that $chop(\mathcal{G}, m, n) \subseteq collect(\mathcal{G}, m, n)$:

We consider two possible cases according to *collect*'s definition. Let us assume first that either $parent(m) = parent(n)$ or $parent(m)$ and $parent(n)$ are undefined. In this case, $collect(\mathcal{G}, m, n) = path(\mathcal{G}, m, n)$, thus it only collects (first rule of function *collect*) the nodes which are in the path from m to n.

Now, we prove that $chop(\mathcal{G}, m, n) = path(\mathcal{G}, m, n)$. Because m and n have the same parent, and m is a prefix of n (by Definition 54), we know that $n = mo$ where $o \in \{l, r\}^*$. Then, by Definitions 46 and 55 only the prefixes of n with m as prefix belong to $chop(\mathcal{G}, m, n)$. These nodes form the path between m and n. Then, $chop(\mathcal{G}, m, n) = path(\mathcal{G}, m, n) = collect(\mathcal{G}, m, n)$. Therefore, the claim holds.

If, on the contrary, $parent(m) \neq parent(n) = p$ or $parent(m)$ is undefined and $parent(n) = p$, rule 2 from *collect* is applied. In this case, all the nodes that belong to the set $\mathcal{S} = \{p\} \cup collect(\mathcal{G}, m, p) \cup subnodes(\mathcal{G}, p) \cup path(\mathcal{G}, pt, n)$ are collected.

We prove the first claim by contradiction assuming that $\exists n' \in chop(\mathcal{G}, m, n)$ and $n' \notin \mathcal{S}$. Because $n' \notin \mathcal{S}$, we know that n' is not an ancestor of n nor a subnode of an ancestor of n. Thus, by Definition 46, n' must be a prefix of n. Moreover, we know that n' is not in the path between m and n, thus it must be a prefix of both n and m. But this is a contradiction w.r.t. the definition of chop (Definition 55), because $m \rightsquigarrow n'$ and thus n' cannot be a prefix of m. Then, $chop(\mathcal{G}, m, n) \subseteq collect(\mathcal{G}, m, n)$

Now, we prove that $collect(\mathcal{G}, m, n) \subseteq chop(\mathcal{G}, m, n)$:

We consider again two possible cases according to *collect*'s definition. Let us assume first that either $parent(m) = parent(n)$ or $parent(m)$ and $parent(n)$ are undefined. In this case, $collect(\mathcal{G}, m, n) = path(\mathcal{G}, m, n)$, thus it only collects (first rule of function *collect*) the nodes which are in the path from m to n.

Now, we prove that $chop(\mathcal{G}, m, n) = path(\mathcal{G}, m, n)$. Because m and n have the same parent, and m is a prefix of n (by Definition 54), we know that $n = mo$ where $o \in \{l, r\}^*$. Then, by Definitions 46 and 55 only the prefixes of n with m as prefix belong to $chop(\mathcal{G}, m, n)$. These nodes form the path between m and n. Then, $chop(\mathcal{G}, m, n) = path(\mathcal{G}, m, n) = collect(\mathcal{G}, m, n)$. Therefore, the claim holds.

If, on the contrary, $parent(m) \neq parent(n) = p$ or $parent(m)$ is undefined and $parent(n) = p$, rule 2 from *collect* is applied. In this case, all the nodes that belong to the set $\mathcal{S} = \{p\} \cup collect(\mathcal{G}, m, p) \cup subnodes(\mathcal{G}, p) \cup path(\mathcal{G}, pt, n)$ are collected.

We assume now that $\exists n' \in \mathcal{S}$ and $n' \notin chop(\mathcal{G}, m, n)$. Then, either p or a node in $collect(\mathcal{G}, m, p)$, $subnodes(\mathcal{G}, p)$ or $path(\mathcal{G}, pt, n)$ does not belong to $chop(\mathcal{G}, m, n)$. First, we know that m is a prefix of n in the initial call (by Definition 54) and in all the recursive calls by rule 2 of $collect(\mathcal{G}, m, n)$); and $parent(m) \neq parent(n) = p$ thus $n = mo$ being o a sequence of letters containing at least one t. Then, we know that $m = p$ or m is a prefix of p. Therefore $p \in chop(\mathcal{G}, m, n)$ and $n' \neq p$. In addition, by rule 2 of Definition 46, we know that the subnodes of p weakly influence n, and they are weakly influenced by m because m is their prefix. Hence $subnodes(\mathcal{G}, p) \subseteq chop(\mathcal{G}, m, n)$ and thus $n' \notin subnodes(\mathcal{G}, p)$. Moreover, because m is a prefix of n and by rule 1 of Definition 46 we have that $\forall m' \in path(\mathcal{G}, m, n), m \rightsquigarrow m' \rightsquigarrow n$ and consequently $path(\mathcal{G}, m, n) \subseteq chop(\mathcal{G}, m, n)$, thus $n' \neq m'$. Then, n' does not belong to $collect(\mathcal{G}, m, p)$, and thus, $\forall n' \in \mathcal{S}, n' \in chop(\mathcal{G}, m, n)$ and the claim holds.

□

As discussed before, we cannot guarantee minimality (i.e., ensure that only nodes that influence the slicing criterion are collected) w.r.t. Definition 44. This loss of precision is unavoidable because during algorithmic debugging we have available only one ART (the wrong computation). However, the algorithm in Figure 6.4 is minimal w.r.t. Definition 46 (i.e., it ensures that only nodes that weakly influence the slicing criterion are collected). Nevertheless, even if the underlying ART corresponds to a lazy evaluation language (this is the case, for instance, of a debugging session in Haskell) and, thus, not needed terms are represented by $\perp$ in the ART, the slice produced is not minimal w.r.t. Definition 44. This phenomenon is due to *sharing*: A previously computed reduced node can appear as argument in a non-strict position of a function call. Then, it is not possible to know which part of this reduction was done before and which part was done after the start of the slice. Therefore, only when the slicing criterion starts at the beginning of the computation (classical backward slicing) the slice produced is minimal, and thus it contains all and only the nodes that could influence the slicing criterion.

Theorem 57 (Completeness of execution trees)
Given an execution tree $\mathcal{E}$ for an ART $\mathcal{G}$, an equation $l = r \in \mathcal{E}$ and an expression e that is a subexpression of l or r, marked as wrong during an algorithmic debugging session, then $collect(\mathcal{G}, epsilon, n)$, where $n \in \mathcal{G}$ is associated to e, contains the (buggy) nodes which caused e.

Proof. By Theorem 56 we have that $collect(\mathcal{G}, epsilon, n) = chop(\mathcal{G}, epsilon, n)$, and hence, by Definition 55, $collect(\mathcal{G}, epsilon, n) = \{n' \in \mathcal{G} \mid epsilon \rightsquigarrow n' \text{ and } n' \rightsquigarrow n\}$. With the first rule of Definition 46 holds that $\forall m \in \mathcal{G}, epsilon \rightsquigarrow m$ and, thus, $collect(\mathcal{G}, epsilon, n) = \{n' \in \mathcal{G} \mid n' \rightsquigarrow n\}$. Moreover, by Proposition 52 we have that $collect(\mathcal{G}, epsilon, n) \supseteq \{n' \in \mathcal{G} \mid n' \approx\!\!\!\!\rightarrow n\}$ and hence, $collect(\mathcal{G}, epsilon, n)$ contains the (buggy) nodes which caused e.

□

6.7 The Technique in Practice

In this section we show an example of how to integrate our technique with an algorithmic debugger. Consider again the example program in Figure 1.1 and its associated execution tree in Figure 1.3. The debugging session using our technique is depicted in Figure 6.5.

```
Starting Debugging Session...

(1) main = False? NO
(2) sqrtest [1,2] = False? NO
(3) test (9,9,8) = False? YES (The programmer selects "8")
(4) computs 3 = (9,9,8)? NO
(5) comput3 3 = 8? NO
(6) listsum [6,2] = 8? YES
(7) partialsums 3 = [6,2]? NO (The programmer selects "2")
(8) sum2 3 = 2? NO
(9) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

Figure 6.5: Debugging session

This debugging session is very similar to the one in Figure 1.2. There are only two differences: The programmer is allowed to provide the system with more information and he does in questions '3' and '7'; and there are less questions to answer, the latter being the consequence of the former.

Internally, when the programmer selects an expression, he is indirectly defining a slicing criterion. Then, the system uses this slicing criterion to slice the execution tree, thus reducing the number of questions. In the example, when the programmer selects the expression "8"[3], the system automatically removes from the execution tree eleven possible questions: all the subtrees of the equations "*comput*1 3 = 9" and "*comput*2 3 = 9" (see dark nodes on the left of Figure 1.3). Similarly, when the programmer selects the expression "2", the system automatically removes from the execution tree two possible questions: the subtree of the equation "*sum*1 3 = 6" (questions 21 and 22 in Figure 1.3). Note that the slicing technique is completely transparent for the programmer.

In general, the amount of information deleted from the tree is directly related to the size of the data structures in the equations, because when the programmer selects one subexpression, the computation of the other subexpressions can be avoided. Another important advantage of combining program slicing and algorithmic debugging is that it allows the programmer to guide the questions of the algorithmic debug-

[3]Note that, even when the equation is correct, the programmer can mark an expression as inadmissible, providing the system with useful information about the bug.

ger. Traditionally, the programmer had to answer the questions without taking part in the process. Thanks to the slicing mechanism, the programmer can select which (sub)expressions he wants to inspect (i.e., because they are wrong or just suspicious), thus guiding the algorithmic debugger and producing a sequence of questions semantically related for the programmer.

For large computations a trace is huge. Hat stores its ART on disk, so that the size of the ART is not limited by the size of the main memory. In contrast, the Mercury debugger materialises parts of the execution tree at a time, by rerunning the computation. Because a slice can be a large part of the ART, collecting all its nodes could be very expensive for both implementation choices. However, for directing the algorithmic debugging process, we do not need all nodes at once. We only have to determine them one at a time to determine the next question. We are in a different situation when we want to highlight in the program source the program slice associated with the trace slice. Such an operation can take time linear in the size of the trace and is thus unsuitable for an interactive tool, as the source-based algorithmic debugger of Hat demonstrates (see related work).

6.8 Related Work

The idea of improving algorithmic debugging by allowing the programmer to provide specific information about the location of errors is not new. In fact, it was introduced by Pereira almost two decades ago [Pereira, 1986]. The original work of Pereira defined—in the context of Prolog—how an algorithmic debugger could take advantage of additional information from the programmer in order to reduce the number of questions. Surprisingly, later research in this subject practically ignored this work The first attempt to adapt the ideas of Pereira to the imperative programming paradigm was [Fritzson *et al.*, 1992]. In this work, the authors propose the use of program slicing in order to reduce execution trees, thus reducing the number of questions in algorithmic debugging. Dynamic program slicing was the technique that Pereira needed in his development, but his work was done in 1986. Even though program slicing was published in 1984 [Weiser, 1984], dynamic program slicing was introduced only in 1988 [Korel and Laski, 1988].

Although the use of program slicing in algorithmic debugging was shown to be very useful, the technique by Fritzson et al. is still limited, because the programmer is only allowed to specify which variables have a wrong value, but he is not allowed to specify which part of the value is wrong. This information could notably improve the effectiveness of the technique as it was shown in Example 6.5. In addition, the authors do not give details about the dynamic slicing algorithm used, so it is not clear whether the variables selected by the programmer must be output variables,

because they only considered the use of program slicing in wrong equations (the concept of *inadmissibility* was lost). Later the ideas introduced by Fritzson et al. have been applied to the logic language paradigm in [Paakki *et al.*, 1994] and extended in different ways. In particular, Kokai *et al.* [1997] have defined the IDTS system which integrates algorithmic debugging, testing and program slicing techniques in a single framework. Our work can be seen as an adaptation of these ideas to the functional paradigm. Functional programs impose different technical challenges (i.e., generalised use of higher-order functions, absence of nondeterminism and unification) which implies the use of different formalisms compared to logic programs. For instance, while ARTs directly represent the sequence of function unfoldings, which allows us to trace data flows, the slicing of Prolog proof trees requires explicit annotation of the tree with information about the relations between terms. These differences produce an essentially different slicing processes.

MacLarty [2005] have implemented an algorithmic debugger for the functional logic language Mercury which allows the programmer to specify which parts of an equation are wrong (including subterms). In particular, they have defined a strategy called "*subterm dependency tracking*" which, given a wrong term, finds the 'origin' of this subterm in the execution tree. This information is used by the tool for determining which questions are more likely to be wrong, thus improving the algorithmic debugging process (see Section 1.2.3). Although this process can be considered as a kind of program slicing, they have not defined it formally, and, for the case of wrong input arguments, the method has only been proposed but not implemented nor proved correct.

The source-based trace explorer of Hat [Chitil, 2005] implements algorithmic debugging and can display a simple program slice that contains the bug location and that shrinks when more questions are answered. However, the programmer cannot indicate wrong subexpressions and the slice does not direct the algorithmic debugging process.

To the best of our knowledge, our technique is the first formal approach that 1) is able to slice execution trees for either input or output expressions, 2) copes with complete expressions and subexpressions and 3) is proven complete. We consider functional programs, a paradigm that completely lacked debugging techniques combining algorithmic debugging and program slicing.

Chapter 7

A Classification of Program Slicing and Algorithmic Debugging Techniques

This chapter introduces two classifications. Firstly, the program slicing techniques presented in Section 1.2.6 are compared and classified in order to identify relationships between them. Secondly, three new algorithmic debugging strategies are introduced, and they are compared and classified together with those presented in Section 1.2.3.

Part of the material of this chapter has been presented in the *International Symposium on Logic-based Program Synthesis and Transformation* (LOPSTR 2006) celebrated in Venice (Italy) in 2006 [Silva, 2006].

7.1 Introduction

In this chapter we discuss and compare the program slicing techniques and algorithmic debugging strategies of Sections 1.2.6 and 1.2.3.

Firstly, following the approach of Harman *et al.* [1996], the techniques in Section 1.2.6 are compared in order to clarify and establish the relations between them. This comparison gives rise to a classification of techniques which can help to guide future research directions in this field. In particular, we compare and classify the techniques aiming at identifying relations between them in order to answer questions like *Is one technique a particular case of another? Is one technique more general or more expressive than another? Are they equivalent but expressed with different formalisms?*

Secondly, we introduce three new algorithmic debugging strategies that can reduce

the number of questions asked during an algorithmic debugging session.

Finally, we classify and study the costs of the new algorithmic debugging strategies and those of Section 1.2.3. Based on this study, we introduce a parameterizable algorithm which allows us to dynamically combine different algorithmic debugging strategies in a single debugging session.

The chapter has been structured as follows: The next section revisits the classification introduced by Harman *et al.* [1996] and we extend it with new slicing techniques and dimensions. The analysis provides useful information that allows us to classify the slicing techniques and establish relations between them. In Section 7.3 three new algorithmic debugging strategies are introduced. Finally, in Section 7.4 we present a comparison of algorithmic debugging strategies and we study their costs.

7.2 Classifying Slicing Techniques

In this section we use a classification by Harman *et al.* [1996] in order to compare and classify all slicing techniques presented in Section 1.2.6. Table 7.1 extends Harman et al's classification with new dimensions in order to be able to classify new techniques. In the table, we have omitted those slicing criteria which have been identified as a particular case of another technique in Section 1.2.6 and thus, they do not impose additional restrictions or dimensions in the table. For instance, we omit in the table "End Slicing" because it is a particular case of simultaneous slicing. In addition, we only include in the table the most general form of slicing for each technique. For instance, in the case of quasi-static slicing, the original definition by Vengatesh considered slicing only "*at the end of the program*"; here, in contrast, we consider a generalization (see Section 1.2.6) in which any point of the program can be selected for slicing. Similarly, we consider simultaneous slicing as a set of any slicing criteria (in contrast to simultaneous dynamic slicing where all the criteria are dynamic); and we consider conditioned amorphous slicing [Harman and Danicic, 1997] which generalizes static amorphous slicing.

In the table, following Harman et al's terminology, the sets $\mathcal{I}$, $\mathcal{S}$ and $\mathcal{N}$ refer respectively to the set of all program variables, possible initial states and iterations. For each slicing technique (first column),

- column *Var. (Variables)* specifies the number of variables participating in a slicing criterion $\mathcal{C}$, where $\mathcal{P}(I)$ indicates that a subset of I participates in $\mathcal{C}$,
- column *Initial States* shows the number of initial states considered by the corresponding slicing criterion,
- column *Slice Points* describes the number of program points included in $\mathcal{C}$, and hence the number of slices actually produced (and combined) to extract the

final slice. Here, *n derived* means that *n* different points are implicitly specified (see Section 1.2.6); and *1+n* means that 1 program point is specified together with n breakpoints,

- column *I.C. (Iteration Counts)* indicates, for the specified slice points, which iterations are of interest,
- column *Dir. (Direction)* states the direction of the traversal to produce the slice which can be backwards (B), forwards (F) or a combination of them. Here, despite some techniques accept both directions, we assign to every technique the direction specified in its original definition (see Section 1.2.6),
- column *P. (Precision)* is marked if the slice is precise, in the sense that it does not contain statements which cannot influence (or, in forward slicing, be influenced by) the slicing criterion. Finally,
- column *Sy/Se Pres. (Syntax/Semantics Preserving)* contains boolean parameters that indicate respectively whether the slice produced is a projection of the program syntax or it preserves a projection of the original semantics. Here, (e) means that the slice produced is executable.

The comparison of slicing techniques presented in Table 7.1 is a powerful tool for the study of current slicing techniques and the prediction of new ones. For instance, the last five rows in Table 7.1 correspond to new slicing techniques predicted by analyzing the information of the table.

The first new technique is point slicing. In all the slicing techniques presented so far one or more variables of the program are part of the slicing criterion. In contrast, point slicing does not consider any variable in the slicing criterion. Point slicing selects a sentence of the program and computes backwards (resp. forwards) all the sentences that could have been executed before (respectively after) it. This is useful in debugging. For instance, during print debugging [Agrawal, 1991] the programmer places print statements in some strategic points of the program, and then executes it in order to see if some of them have been reached. Usually, a print statement is placed in a part of the code in order to check if it is executed or not. A point slice w.r.t. this statement can be useful in order to know which possible paths of the program could reach the execution of this sentence. Note that print slicing does not follow control or data dependences, but control flows, and thus it is not subsumed by any of the other slicing techniques in the table. As an example, the statement "`print ('Executed');`" does not influence any other statement, and hence, it will not belong to any slice taken w.r.t. a subset of the variables of the program. The conditioned version of point slicing restricts the possible paths by limiting the initial states of the execution.

Slicing Technique	**Var.**	**Initial States**	**Slice Points**	**I.C.**	**Dir.**	**P.**	**Sy/Se Pres.**
Static Slicing	$\mathcal{P}(I)$	$\{S\}$	1	$\{\mathcal{N}\}$	B	y	$y/y(e)$
Dynamic Slicing	$\mathcal{P}(I)$	1	1	1	B	y	$y/y(e)$
Forward Slicing	$\mathcal{P}(I)$	$\{S\}$	1	$\{\mathcal{N}\}$	F	y	y/y
Quasi Static Slicing	$\mathcal{P}(I)$	$\mathcal{P}(S)_{\star}$	1	$\{\mathcal{N}\}$	B	y	$y/y(e)$
Conditioned Slicing	$\mathcal{P}(I)$	$\mathcal{P}(S)_{\star\star}$	1	$\{\mathcal{N}\}$	B	y	$y/y(e)$
Decomposition Slicing	1	$\{S\}$	n derived	$\{\mathcal{N}\}$	B	y	y/y
Chopping	$\mathcal{P}(I)$	$\{S\}$	pair	$\{\mathcal{N}\}$	$B \wedge F$	y	y/n
Relevant Slicing	$\mathcal{P}(I)$	1	1	1	B	n	y/n
Hibrid Slicing	$\mathcal{P}(I)$	$\mathcal{P}(S)_{\star\star\star}$	1+n	$\{\mathcal{N}\}$	B	y	$y/y(e)$
Intraprocedural Slicing	$\mathcal{P}(I)$	$\{S\}$	1	$\{\mathcal{N}\}$	B	n	$y/y(e)$
Interprocedural Slicing	$\mathcal{P}(I)$	$\{S\}$	1	$\{\mathcal{N}\}$	B	y	$y/y(e)$
Simultaneous Slicing	$\mathcal{P}(I)$	$\mathcal{P}(S)$	n	$\{\mathcal{N}\}$	$B \vee F$	y	$y/y(e)$
Dicing	$\mathcal{P}(I)$	$\{S\}$	n	$\{\mathcal{N}\}$	B	y	y/n
Abstract Slicing	$\mathcal{P}(I)$	$\mathcal{P}(S)_{\star\star}$	1	$\{\mathcal{N}\}$	B	y	$y/y(e)$
Amorphous Slicing	$\mathcal{P}(I)$	$\mathcal{P}(S)_{\star\star}$	1	$\{\mathcal{N}\}$	B	y	$n/y(e)$
New Techniques							
Point Slicing	$\varnothing$	$\{S\}$	1	$\{\mathcal{N}\}$	B	n	$y/y(e)$
Conditioned Point	$\varnothing$	$\mathcal{P}(S)_{\star\star}$	1	$\{\mathcal{N}\}$	B	n	$y/y(e)$
Conditioned Chopping	$\mathcal{P}(I)$	$\mathcal{P}(S)_{\star\star}$	pair	$\{\mathcal{N}\}$	$B \wedge F$	y	y/n
Forward Amorphous	$\mathcal{P}(I)$	$\mathcal{P}(S)_{\star\star}$	1	$\{\mathcal{N}\}$	F	y	$n/y(e)$
Forward Point	$\varnothing$	$\{S\}$	1	$\{\mathcal{N}\}$	F	n	$y/y(e)$

$\star$ all agree on input prefix $\star\star$ all agree on condition $\star\star\star$ all agree on breakpoints

Table 7.1: Classification of program slicing techniques

Conditioned chopping is a very general form of slicing which subsumes both conditioned slicing and chopping, and hence, dynamic chopping (neither defined yet). Dynamic chopping can be useful, for instance, in debugging. When tracing a computation, a programmer usually proceeds (big) step by step until a wrong subcomputation is found (e.g., a procedure call that returned a wrong result). Dynamic chopping comes in handy at this point because it can compute the sentences that influenced the wrong result from the procedure call. This dynamic chop is, in general, much smaller than the static chop because it only considers a particular execution.

Finally, the last two rows of the table, forward amorphous and forward point slicing correspond to the forward versions of these techniques.

The information in the table can also be used to identify relations between slicing techniques. We have identified some relations and represented them in the graph of

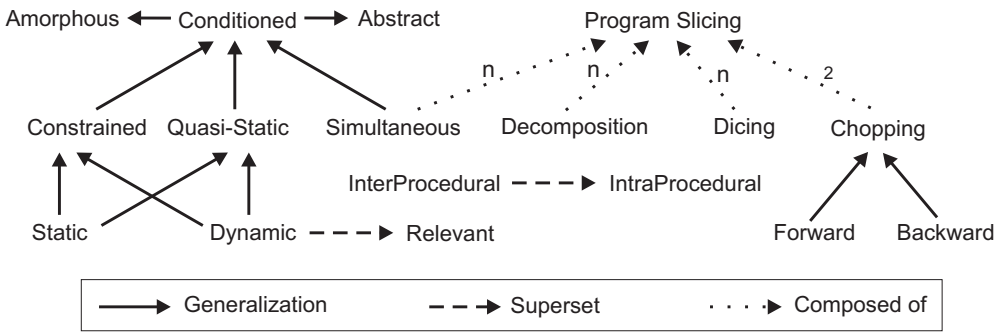


Figure 7.1: Relationships between program slicing techniques

Figure 7.1. There are three kinds of arrows in the graph:

Generalization ($S_1 \longrightarrow S_2$): A slicing technique S_2 generalizes another technique S_1 iff all the slicing criteria that can be specified with S_1 can also be specified with S_2.

Superset ($S_1 \dashrightarrow S_2$): The slice produced by a slicing technique S_2 is a superset of the slice produced by another technique S_1 iff all the statements in S_1 also belong to S_2.

Composed of ($S_1 \overset{n}{\cdots\triangleright} S_2$): A slicing technique S_1 is composed of n other techniques S_2 iff the slicing criterion of S_1 contains n slicing criteria of S_2. In the graph, for clarity, "Program Slicing" represents different slicing techniques. For instance, chopping is composed of two slicing techniques (forward slicing and backward slicing) and dicing is composed of n slicing techniques (one backward slice of an incorrect value, and $n-1$ backward slices of a correct value).

The classification points out abstract slicing as one of the most general slicing techniques thanks to its use of predicates and conditions. While chopping generalizes forward and backward slicing, an abstract chop [Hong *et al.*, 2005] generalizes forward and backward abstract slicing. Hence abstract chopping subsumes static, dynamic, backward and forward slicing.

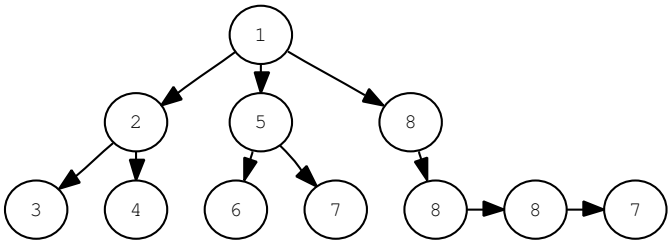
7.3 Three New Algorithmic Debugging Strategies

7.3.1 Less YES First

This section introduces a new variant of top-down search which further improves heaviest first. It is based on the fact that every equation in the ET is associated with

a rule of the source code (i.e., the rule that the debugger identifies as buggy when it finds a buggy node in the ET). Taking into account that the final objective of the process is to find the program's rule which contains the bug—rather than a node in the ET—and considering that there is not a relation one-to-one between nodes and rules because several nodes can refer to the same rule, it is important to also consider the node's rules during the search. A first idea could be to explore first those subtrees with a higher number of associated rules (instead of exploring those subtrees with a higher number of nodes).

Example 18 *Consider the following ET:*



where each node is labeled with its associated rule and where the oracle answered NO to the question in the root of the tree. While heaviest first selects the right-most child because this subtree has four nodes instead of three, less YES first selects the left-most child because this subtree contains three different rules instead of two.

Clearly, this approach relies on the idea that all the rules have the same probability of containing the bug (rather than all the nodes). Another possibility could be to associate a different probability of containing the bug to each rule, e.g., depending on its structure: Is it recursive? Does it contain higher-order calls?.

The probability of a node to be buggy is $q \cdot p$ where q is the probability that the rule associated to this node is wrong, and p is the probability of this rule to execute incorrectly. Therefore, under the assumption that all the rules have the same probability of being wrong, the probability P of a branch b to contain the bug is:

$$P = \frac{\sum_{i=1}^{n} p_i}{R}$$

where n is the number of nodes in b, R is the number of rules in the program, and p_i is the probability of the rule in node i to produce a wrong result if it is incorrect. Clearly, if we assume that a wrong rule always produces a wrong result[1] (i.e., $\forall i.p_i = 1$), then

[1]This assumption is valid for instance in those flattened functional languages where all the conditions in the right-hand side of function definitions have been distributed between its rules. This is relatively frequent in internal languages of compilers, but not in source languages.

the probability is $\frac{r}{R}$ where r is the number of rules in b, and thus, this strategy is (on average) better than heaviest first. For instance, in Example 18 the left-most branch has a probability of $\frac{3}{8}$ to contain a buggy node, while the right-most branch has a probability of $\frac{2}{8}$ despite it has more nodes.

However, in general, a wrong rule can produce a correct result, and thus we need to consider the probability of a wrong rule to return a wrong answer. This probability has been approximated by the debugger Hat-delta (see Section 1.2.3) by using previous answers of the oracle. The main idea is that a rule answered NO n times out of m is more likely to be wrong than a rule answered NO n' times out of m if $n' < n \leqslant m$.

Here, we use this idea in order to compute the probability of a branch to contain a buggy node. Hence, this strategy is a combination of the ideas from both heaviest first and Hat-delta. However, while heaviest first considers the structure of the tree and does not take into account previous answers of the user, Hat-delta does the opposite; thus, the advantage of less YES first over them is the use of more information (both the structure of the tree and previous answers of the user).

A direct generalization of Hat-delta for branches would result in counting the number of YES answers of a given branch; but this approach would not take into account the number of rules in the branch. In contrast, we proceed as follows:

When a node is set correct, we mark its associated rule and all the rules of its descendants as correctly executed. If a rule has been executed correctly before, then it will likely execute correctly again. The debugger associates to each rule of the program the number of times it has been executed in correct computations based on previous answers. Then, when we have to select a child to ask, we can compute the total number of rules in the subtrees rooted at the children, and the total number of answers YES for every rule.

This strategy selects the child whose subtree is less likely to be correct (and thus more likely to be wrong). To compute this probability we calculate for every branch b a weight w_b with the following equation:

$$w_b = \sum_{i=1}^{n} \frac{1}{r_i^{(YES)}}$$

where n is the number of nodes in b and $r_i^{(YES)}$ is the number of answers YES for the rule r of the node i.

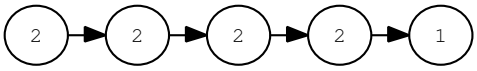
As with heaviest first, we select the branch with the biggest weight, the difference is that this equation to compute the weight takes into account previous answers of the user. Moreover, we assume that initially all the rules have been answered YES once, and thus, at the beginning, this strategy asks those branches with more nodes, but it becomes different as the number of questions asked increases.

As pointed out by Davie and Chitil [2006], independently of the considered strategy, taking into account the rule associated with each node can help to avoid many unnecessary questions. Consider the following example:

Example 19 *The ET associated with function* `append`*:*

```
(1)  append [ ] y = y
(2)  append (x:xs) y = x:append xs y
```

w.r.t. the call "`append [1,2,3,4] [5,6]`*" is the following:*



*where each node is labeled with its associated rule. Here, the bug can be found with only one question to the node labeled "*1*". If the answer is NO the bug is in rule* 1*, if the answer is YES then we can stop the search because the rest of the nodes have the same associated rule (*2*) and, thus, the bug must necessarily be in rule* 2*. Note that the use of this information can significantly improve the search. With a list of n numbers as the first argument, other strategies could need n+1 questions to reach the bug.*

Davie and Chitil have generalized this idea and introduced the tree compression technique [Davie and Chitil, 2006]. This technique proceeds as follows: For any node n_1 with associated rule r_1 and child n_2 with associated rule r_1, it replaces node n_2 with its children (i.e., the children of n_2 become the children of n_1). Therefore, the tree compression technique prunes ETs even before starting to ask any question.

With this strategy, the sequence of 9 questions asked for the ET in Figure 1.3 is:

```
 Starting Debugging Session...
(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(4)  computs 3 = (9,9,8)? NO
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

7.3.2 Divide by YES & Query

The same idea used in less YES first can be applied in order to improve divide & query. Instead of dividing the ET into two subtrees with a similar number of nodes, we can divide it into two subtrees with a similar weight. The problem that this strategy tries to address is the D&Q's assumption that all the nodes have the same probability of containing the bug. In contrast, this strategy tries to compute this probability.

By using the equation to compute the weight of a branch, this strategy computes the weight associated to the subtree rooted at each node. Then, the node which divides the tree into two subtrees with a more similar weight is selected. In particular, the node selected is the node which produces a least difference between:

- $w/2$ and the heaviest node whose weight is less than or equal to $w/2$
- $w/2$ and the lightest node whose weight is greater than or equal to $w/2$

where w is the weight of the suspicious area in the ET.

As with D&Q, different nodes could divide the ET into two subtrees with a similar weights; in this case, we could follow another strategy (e.g., Hirunkitti) in order to select one of them.

We assume again that initially all the rules have been answered YES once. Therefore, at the beginning this strategy is similar to D&Q, but the differences appear as the number of answers increases.

Example 20 *Consider again the ET in Example 18. Similarly to D&Q, the first node selected is the top-most "8" because only structural information is available. Let us assume that the answer is YES. Then, we mark all the nodes in this branch as correctly executed. Therefore, the next node selected is "2"; because, despite the subtrees rooted at "2" and "5" have the same number of nodes and rules, we now have more information which allows us to know that the subtree rooted at "5" is more likely to be correct since node "7" has been correctly executed before.*

The main difference with respect to D&Q is that divide by YES & query not only takes into account the structure of the tree (i.e., the distribution of the program rules between its nodes), but also previous answers of the user.

With this strategy, the sequence of 5 questions asked for the ET in Figure 1.3 is:

```
Starting Debugging Session...
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
```

```
(24) decr 3 = 2? YES

Bug found in rule: sum2 x = div (x + (decr x)) 2
```

7.3.3 Dynamic Weighting Search

Subterm dependency tracking (see Section 1.2.3) relies on the idea that if a subterm is marked, then the error will likely be in the sequence of functions that produced and passed the incorrect subterm until the function where the programmer found it. However, the error could also be in any other equation previous to the origin of the dependency chain.

Here, we propose a new strategy which is a generalization of subterm dependency tracking and which can integrate the knowledge acquired by other strategies in order to formulate the next question.

The main idea is that every node in the ET has an associated weight (representing the probability of being buggy). After every question, the debugger gets information that changes the weights and it asks for the node with a higher weight. When the associated weight of a node is 0, then this node leaves the suspicious area of the ET. Weights are modified based on the assumption that those nodes of the tree which produced or manipulated a wrong (sub)term, are more likely to be wrong than those that did not. Here, we compute weights instead of probabilities w.l.o.g. because this will ease a later combination of this strategy with previous ones. We assume initially that all the nodes have a weight 1 and that a weight 0 means "*out of the suspicious area*".

Computing Weights from Subterms

Firstly, as with subterm dependency tracking, we allow the oracle to mark a subterm from an equation as wrong (instead of the whole equation). Let us assume that the programmer is being asked about the correctness of the equation in a node n_1, and he marks a subterm s as wrong (or inadmissible). Then, the suspicious area is automatically divided into four sets. The first set contains the node, say n_2, that introduced s into the computation and all the nodes needed to execute the equation in node n_2. The second set contains the nodes that, during the computation, passed the wrong subterm from equation to equation until node n_1. The third set contains all the nodes which could have influenced the expression s in node n_2 from the beginning of the computation. Finally, the rest of the nodes form the fourth set. Since these nodes could not produce the wrong subterm (because they could not have influenced it), the nodes in the fourth set are extracted from the suspicious area and, thus, the new suspicious area is formed by the sets 1, 2 and 3.

Each subset can be assigned a different probability of containing the bug. Let us show it with an example.

Example 21 *Consider the ET in Figure 1.3, where the oracle was asked about the correctness of equation 3 and he pointed out the computed subterm "*`8`*" as inadmissible. Then, the four sets are denoted in the figure by using different shapes and colors:*

- ***Set 1:*** *those nodes which evaluated the equation 20 to produce the wrong subterm are denoted by an inverted trapezium.*
- ***Set 2:*** *those nodes that passed the wrong subterm until the programmer detected it in the equation 3 are denoted by an ellipse.*
- ***Set 3:*** *those nodes that could influence the wrong subterm are denoted by a trapezium.*
- ***Set 4:*** *the rest of nodes are denoted by a grey rectangle.*

The source of a wrong subterm is the equation which computed it. From our experience, all the nodes involved in the evaluation of this equation are more likely to contain the bug. However, it is also possible that the functions that passed this wrong term during the computation should have modified it and they did not. Therefore, they could also contain the bug. Finally, it is also possible (but indeed less likely) that the equation that computed the wrong subterm had a wrong argument and this was the reason why it produced a wrong subterm. In this case, this inadmissible argument should be further inspected. In the example, the wrong term "`8`" was computed because equation 20 had a wrong argument "`[6,2]`" which should be "`[6,3]`"; the nodes which computed this wrong argument have a trapezium shape.

Consequently, in the previous example, after the oracle marked "`8`" as wrong in equation 3, we could increase the weight of the nodes in the first subset with 3, the nodes in the second subset with 2, and the nodes in the third subset with 1. The nodes in the fourth subset can be extracted from the suspicious area because they could not influence the value of the wrong subterm and, consequently, their probability of containing the bug is zero (see Section 6.6).

These subsets of the ET are in fact slices of different parts of the computation. In order to extract these slices, we introduce some expressions which use the definitions introduced in Section 6.4.

Lemma 58 (Origin of an expression in ETs) *Given an ET $\mathcal{E}$ for an ART $\mathcal{G}$ and an expression $e \in \mathcal{E}$, $e \neq$ main associated to $n \in \mathcal{G}$ marked as wrong or inadmissible during an algorithmic debugging session, then, the equation in $\mathcal{E}$ that introduced e into the computation is:*

$$equ(\mathcal{G}, m) = mef(\mathcal{G}, mt) \qquad \text{where } m = parent(n)$$

We often call this equation "the origin of e".

Proof. Firstly, e cannot be further evaluated because it appears in an equation of $\mathcal{E}$ and from Definition 42 holds that n represents e in $\mathcal{G}$. Let us assume that $l = r$ is the equation in $\mathcal{E}$ that introduced e into the computation (the origin of e). Then, either $e \in l$ or $e \in r$.

Since we assume that the computation starts in the distinguished function *main* which have no arguments, and, because new expressions are only introduced in right-hand sides, if $e \in l$ then $e = main$. But one of the premises of the lemma is that $e \neq main$, and thus, $e \in r$.

Let n' be the node associated to r. Since e cannot be further evaluated then $parent(n) = parent(n')$. Therefore, $m = parent(n)$ is the node associated to l, and $mt = n'$ is the node associated to r; then, following Definition 41, $equ(\mathcal{G}, m) = mef(\mathcal{G}, mt)$ is $l = r$.

□

Note that e does not necessarily appear in the equation $l = r \in \mathcal{E}$ which is the origin of e, because e could be a subexpression of l' in the equation $l' = r' \in \mathcal{E}$ where the oracle identified e. For instance, consider the question "$3 - 1 = 2$?" produced from the ART in Figure 6.3. If the oracle marks 1 as inadmissible, then the equation which introduced 1 into the computation is "*decr* $3 = 2$?" which not includes 1 due to the normalization process.

Each of the four subsets is computed as follows:

- The equations in $\mathcal{E}$ which belong to the Set 1 are those which are in the subtree rooted at the origin of e.

- Given an ET $\mathcal{E}$ and two equations $eq, eq' \in \mathcal{E}$ where the expression e appears in eq and the equation eq' is the origin of e, then, if the oracle marks e as wrong or inadmissible during an algorithmic debugging session, the equations in $\mathcal{E}$ which belong to the Set 2 are all the equations which are in the path between eq and eq' except eq and eq'.

- The Set 4 contains all the nodes that could not influence the expression e, and thus, it can be computed following the approach of Section 6.6.1. In particular, by using the algorithm defined in Figure 6.4 a slice w.r.t. the expression e can be computed. All the equations $equ(\mathcal{G}, n) = mef(\mathcal{G}, nt)$ in $\mathcal{E}$ associated to reductions n to nt in $\mathcal{G}$ that do not belong to the slice form Set 4.

- Finally, Set 3 contains all the equations in $\mathcal{E}$ that do not belong to the Sets 1, 2 and 4.

This strategy can integrate information used by other strategies (e.g., previous answers of the oracle) in order to modify nodes' weights. In the next section we introduce an algorithm to combine information from different strategies.

7.4 Classifying Algorithmic Debugging Strategies

7.4.1 Combining Algorithmic Debugging Strategies

In Figure 7.3 we can see that each strategy uses different information during the search. Hence a combination of strategies can produce more efficient new strategies. The algorithm in Figure 7.2 is a simple generic procedure for algorithmic debugging. For simplicity, it assumes that the information provided by the oracle is stored in a data structure '*info*' which is combined by means of the operator $\cup$. Essentially, *info* stores all the answers and expressions marked by the oracle, the information about which nodes belong to the suspicious area and, in general, all the information needed for the strategies being used as, for instance, the number of YES answers for every node. It uses the following three functions:

bugFound(info,ET): It is a simple test to check if the bug can be found with the information provided by the oracle so far.

askQuestion(node): It prompts the oracle with the question of a node and gets new information.

computeWeights(info,ET): This function determines the new weights of the nodes and selects the heaviest node w.r.t. a particular strategy.

Therefore, the strategy used in the algorithm is only implemented by *computeWeights*. Now, we propose a combination of strategies to implement this function.

Function *computeWeights* proceeds by computing the weight of every node, and then returning the node with the maximum weight. We compute the weight of a node from the sum $a + b + c + d + e + f$ where each component is the result of applying a different strategy plus an addend. For instance:

$a = a' +$ Optimizations
$b = b' +$ Computing Weights from Subterms
$c = c' +$ Divide by YES & Query
$d = d' +$ Less YES First
$e = e' +$ Hirunkitti
$f = f' +$ Heaviest First

Input: an execution tree (ET)
Output: a node
Initialization: Nodes' weight set to "1"; info = $\varnothing$
Repeat
    $node = computeWeights(info, ET)$;
    $info = info \cup askQuestion(node)$;
Until $bugFound(info, ET)$
Return: buggy node

Figure 7.2: Algorithmic debugging procedure

where $a' \ldots f'$ are the addends and every strategy produces a weight between 0 and a bound C. It is not difficult to modify a given strategy to produce a weight for each node. For instance, heaviest first can be modified in order to assign a higher weight to the nodes with more descendants. Hence, in Example 18, nodes labeled 5 and 2 would be assigned a weight of three, whilst nodes 3, 4, 6 and 7 would be assigned a weight of two. In addition, we must ensure that weights are not greater than the bound C, thus, in order to assign a weight to node n, we can use the equation:

$$\frac{C \cdot weight(n)}{weight(r)}$$

where r is the root node of the ET. Other strategies can be modified in a similar way, e.g., Less YES first assigns a higher weight to the nodes whose subtree is more likely to contain the buggy node, and D&Q assigns a higher weight to the nodes that are closer to the center of the ET.

Here, we consider *Optimizations* as another strategy because, independently of the strategy used, it is always possible to apply some optimizations during the debugging process. Some of them are obvious, as for instance excluding the first question (`main`) from the suspicious area (because its answer is always NO), or propagating answers to duplicate questions. For instance, in the ET of Figure 1.3 questions 11, 19 and 27 are identical and, thus, if one of them is answered with YES or NO, so are the others. Other optimizations, on the other hand, are less obvious and they can influence the order of the questions as it was shown in Example 19. Optimizations should be used to select a particular node by assigning to it a weight C; and to eliminate nodes which cannot be buggy (i.e., nodes equal to a node answered YES) by assigning them a weight zero. Therefore, the addend of optimizations should be the greatest addend.

For instance, the previous combination of strategies with the addends $a' = 47C$, $b' = 23C$, $c' = 11C$, $d' = 5C$, $e' = 2C$ and $f' = 0$ would apply the optimizations first (if

possible), then, whenever two nodes have the same weight, computing weights from subterms would be applied to break the tie, and so on. This is due to the fact that these addends guarantee the following relations:

$a > b + c + d + e + f$,
$b > c + d + e + f$,
$c > d + e + f$,
$d > e + f$,
$e > f$

Note the importance of C—but not of the value of C—to ensure the previous relations. In contrast, with the addends $a' = b' = ... = f'$ all the strategies would be applied at the same time.

The best combination of strategies depends in each case on the current context (i.e., branching factor of the tree, previous answers of the oracle, etc) thus the combination of strategies used by the debugger should also be dynamically modified during the algorithmic debugging process. Therefore, the debugger itself should state the addends dynamically after every question. In the next section we compare the cost of a set of strategies in order to theoretically determine their addends.

7.4.2 Comparing Algorithmic Debugging Strategies

A summary of the information used by every strategy is shown in Figure 7.3. The meaning of each column is the following:

- '*(Str)ucture*' is marked if the strategy takes into account the distribution of nodes (or rules) in the tree;
- '*(Rul)es*' is marked if the strategy considers the rules associated with nodes;
- '*(Sem)antics*' is marked if the strategy follows an order of semantically related questions, the more marks the more strong relation between questions;
- '*(Ina)dmissibility*' is marked if the strategy accepts "inadmissible" answers;
- '*(His)tory*' is marked if the strategy considers previous answers in order to select the next node to ask (besides cutting the tree);
- '*(Div)isible*' is marked if the strategy can work with a subset of the whole ET. ETs can be huge and thus, it is desirable not to explore the whole tree after every question. Some strategies allow us to only load a part of the tree at a time, thus significatively speeding up the internal processing of the ET; and hence, being much more scalable than other strategies that need to explore the

Strategy	Str.	Rul.	Sem.	Ina.	His.	Div.	Cost
Single Stepping	-	-	-	-	-	✓	n
Top-Down Search	-	-	✓	-	-	✓	$b \cdot d$
Top-Down Zooming	-	-	✓✓	-	-	✓	$b \cdot d$
Heaviest First	✓	-	✓	-	-	-	$b \cdot d$
Less YES First	✓	✓	✓	-	✓	-	$b \cdot d$
Divide & Query	✓	-	-	-	-	-	$b \cdot log_2 n$
Biased Weighting D&Q	✓	-	-	-	-	-	$b \cdot log_2 n$
Hirunkitti's D&Q	✓	-	-	-	-	-	$b \cdot log_2 n$
Divide by YES & Query	✓	✓	-	-	✓	-	-
Hat-delta	-	✓	-	-	✓	-	n
Subterm Dependency Track.	-	-	✓✓✓	✓	-	-	n
Dynamic Weighting Search	✓	✓	-	✓	✓	-	n

Figure 7.3: Comparing algorithmic debugging strategies

whole tree before every question. For instance, top-down can load the nodes whose depth is less than d, and ask d questions before loading another part of the tree. Note, however, that some of the non-marked strategies could work with a subset of the whole ET if they where restricted. For instance, heaviest first could be restricted by simply limiting the search for the heaviest branch to the loaded nodes of the ET. Other strategies need more simplifications: less YES first or Hat-delta could be restricted by only marking as correctly executed the first d levels of descendants of a node answered YES; and then restricting the search for the heaviest branch (respectively node) to the loaded nodes of the ET. Finally,

- '*Cost*' represents the worst case query complexity of the strategy. Here, n represents the number of nodes in the ET, d its maximum depth and b its branching factor.

The cost of single stepping is too expensive. Its worst case query complexity is order n, and its average cost is $n/2$.

Top-down and its variants have a cost of $b \cdot d$ which is significantly lower than the one of single stepping. The improvement of top-down zooming over top-down is based on the time needed by the programmer to answer the questions; their query complexity is the same. Note that, after the tree compression described in Section 7.3.1, no zoom is possible and, thus, both strategies are identical.

In contrast, while in the worst case the costs of top-down and heaviest first are equal, in the mean case heaviest first performs an improvement over top-down. In particular, on average, for each wrong node with b children $s_i, 1 \leq i \leq b$:

- Top-down asks $\frac{b+1}{2}$ of the children.
- Heaviest first asks $\frac{\sum_{i=1}^{b} weight(s_i) \cdot pos(s_i)}{\sum_{i=1}^{b} weight(s_i)}$ of the children.

  where function *weight* computes the weight of a node and function *pos* computes the position of a node in a list containing it and all its brothers which is ordered by their weights.

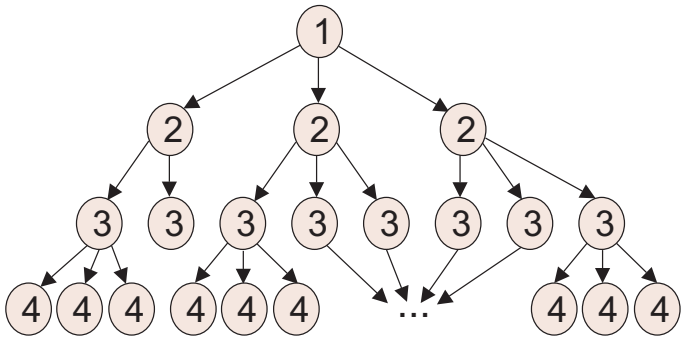
In the case of less YES first, the improvement is based on the fact that the heaviest branch is not always the branch with a higher probability of containing the buggy node. While heaviest first and less YES first have the same worst case query complexity, their average cost must be compared empirically.

D&Q is optimal in the worst case, with a cost order of $(b \cdot (log_2 n))$.

The cost of the rest of strategies is highly influenced by the answers of the user.

The worst case of Hat-delta happens when the branching factor is 1 and the buggy node is in the leaf of the ET. In this case the cost is n. However, in normal situations, when the ET is wide, the worst case is still close to n; and it occurs when the first branch explored is answered YES, and the information of YES answers obtained makes the algorithmic debugger explore the rest of the ET bottom up. It can be seen in Example 22.

Example 22 *Consider the following ET:*



If we use the version of Hat-delta which counts correct computations, then the first node asked would be the left-most 2. If it is answered YES, then all the nodes in this branch are marked as correctly executed, and thus, rule 2 has been correctly executed once, rule 3 has been correctly executed twice and rule 4 has been correctly executed three times. Therefore, the rest of the nodes are asked bottom up. In this case, the cost is $n - \frac{(d-1)d}{2}$.

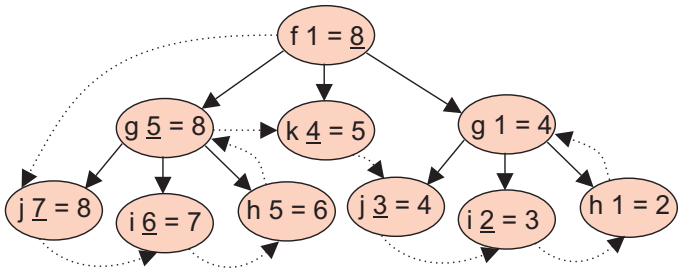
Despite subterm dependency tracking is a top-down version enriched with additional information provided by the oracle, this information (that we assume correct here to

compute the costs) could make the algorithmic debugger ask more questions than with the standard top-down. In fact, this strategy—and also dynamic weighting search if we assume that top-down is used by default—has a worst case query complexity of n because the expressions marked by the programmer can make the algorithmic debugger explore the whole ET. The next example shows the path of nodes asked by the algorithmic debugger in the worst case.

Example 23 *Consider the following program:*

```
f x = g (k (g x))
g x = j (i (h x))
h x = x+1
i x = x+1
j x = x+1
k x = x+1
```

whose ET w.r.t. the call `f 1` *is:*

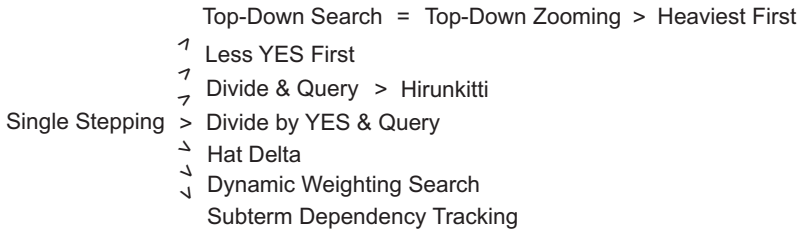


Here, the sequence of nodes asked with subterm dependency tracking is depicted with dotted arrows and the expressions marked by the programmer as wrong or inadmissible are underlined. Then, the debugging session is:

```
Starting Debugging Session...
f 1 = 8? NO  (The user selects "8")
j 7 = 8? YES (The user selects "7")
i 6 = 7? YES (The user selects "6")
h 5 = 6? YES
g 5 = 8? YES (The user selects "5")
k 4 = 5? YES (The user selects "4")
j 3 = 4? YES (The user selects "3")
i 2 = 3? YES (The user selects "2")
h 1 = 2? YES
g 1 = 4? NO
Bug found in rule:
g x = j (i (h x))
```

And, thus, all the ET's nodes are asked.

As a summary, the following diagram shows the relation between the costs of the strategies discussed so far:



This diagram together with the costs associated to every strategy in Figure 7.3 allows algorithmic debuggers to automatically determine which strategy to use depending on the structure of the ET. This means that the algorithmic debugger can dynamically change the strategy used during a debugging session. For instance, after every question the new maximum depth (d) of the ET and its remaining number of nodes (n) is computed; if $d < log_2 n$ then heaviest first is used in order to select a new node, else Hirunkitti is applied. If we use this result in the combination of strategies proposed in Section 7.4.1 we can conclude that if $d < log_2 n$ then $f' < e'$.

Chapter 8

Conclusions and Future Work

We conclude by summarizing the main contributions of this thesis and pointing out several directions for further research.

8.1 Conclusions

Bugs are a human symptom. No matter which language or methodology we use, we will involuntarily introduce bugs in our specifications. It is not due to our lack of knowledge or undiscipline; the reason relies on the human incapacity to predict all the future possible consequences of our actions in the present. The automatization of software development tasks is gradually decreasing the number of initial bugs in programs, nevertheless, until the day that all the software development process is fully automated—while humans participate in the process—programs will contain bugs.

This pessimism is not unfounded (some discussions can be found, e.g., in [Agrawal, 1991]), and shows the intrinsically necessity of debugging tools. Software can be a very sophisticated and complex beast, what justifies the more than thirty years of research and efforts to produce usable debugging tools. Every new programming language or technique opens a number of doors for new research: program slicing produced static slicing, dynamic slicing, quasi-static slicing, simultaneous dynamic slicing, conditioned slicing and many other approaches; profiling produced program counter sampling, block counting and compiler profilers, including techniques to measure CPU usage, call counts, call cost, memory usage, I/O operations, etc; tracing produced breakpoints, execution backtracking, and has been combined with a number of techniques such as program slicing and algorithmic debugging; all of them towards the search of the same thing: finding and eliminating bugs from programs.

After all, the task of debugging still remains complex. The problem is not solved by far, and it calls for further investigation. One of the main problems of current

debugging tools is that after generating and using all the data structures (e.g., redex trails, instrumented programs, etc.) needed for debugging, they become mostly useless, and they must be generated and analyzed again for a different program or computation. This is one of the causes that many debugging tools can be directly classified as abandonware. Some bugs can be easily found, others, on the contrary, can be extremely difficult. If the debugging tool is not close to the user, or it requires much knowledge or work to be used, it is not used in practice. In general, many programmers are too lazy to learn and use complex debugging tools that hopefully will help them to find a bug that maybe they can find with a print statement.

Learning to use a debugging tool is a long term investment; it becomes interesting while you use them more and more, because both your knowledge about the debugger and the knowledge of the debugger[1] about your programs increase. Unfortunately, there is currently a strong tendency to reject this investment. This situation could change with the improvement of debugging tools rather than its intrinsic techniques. As stated earlier [Wadler, 1998], there is a wide gap between researchers and programmers: The formers put much efforts in defining debugging techniques and little efforts in producing prototypes that, at the end, are far from the programmers needs.

Debugging tools help the user to solve problems; if they introduce new problems they become useless. For instance, if the debugger needs a complex specification (e.g., about the intended semantics of the program) it can enter into a "*buggy search of bugs*" where the programmer has accidentally introduced a bug in the specification for the debugger. Therefore, one strong requirement of debugging tools is the usability. They must be easy to use and provide the user with concrete information. Modern graphical interfaces can play a crucial role in order to increment the usage rate of debugging tools.

In this thesis, we have developed some debugging techniques that (hopefully) constitute a step forward in the search for the free-bugs software. In short, the main contributions of the thesis are:

1. Profiling Techniques.

   We have introduced a novel program transformation to measure the cost of lazy functional (logic) computations. This is a difficult task mainly due to the *on-demand* nature of lazy evaluation. The approach is simple, precise—i.e., it computes exact costs rather than upper/lower bounds—, and fully automatic. However, the symbolic costs computed are only based on function unfoldings.

   Therefore, we introduced another profiling scheme for modern functional logic languages which is more advanced and is able to analyze a set of different symbolic costs.

[1] This is the case, e.g., of modern declarative debuggers, which maintain a database of incremental acquired knowledge about programs.

Firstly, we instrument a natural (big-step) semantics for functional logic programs which covers laziness, sharing and non-determinism. This contrasts with [Sansom and Jones, 1997], where logical features are not considered, and [Albert and Vidal, 2002], where sharing is not covered (which drastically reduces its applicability). The instrumented semantics associates a symbolic cost to each basic operation (e.g., variable updates, function unfoldings, case evaluations). While this *cost semantics* provides a formal basis to analyze the cost of a computation, the implementation of a cost-augmented interpreter based on it would introduce a huge overhead. Therefore, we also introduced a sound transformation that instruments a program such that its execution—under the standard semantics—yields not only the corresponding results but also the associated costs.

2. Program Slicing Techniques.

   We have presented a new scheme to perform dynamic slicing in lazy functional logic programs which is based on redex trails. For this purpose, we have introduced a non-trivial notion of slicing criterion which is adequate in the context of lazy evaluation. We have also formalized the concept of minimal dynamic backward slicing in this context. We have defined the first (complete) dynamic slicing technique for the Curry language, we have shown how tracing (based on redex trails) and slicing can be combined into a single framework, and, we have presented a lightweight approach to program specialization based on the dynamic slicing technique.

   Finally, we have compared a wide variety of program slicing based techniques in order to classify them according to their properties and in order to establish a hierarchy of slicing techniques. In particular, we have extended a comparative table of slicing techniques by Harman et al. with new dimensions and techniques. The classification of techniques presented not only shows the differences between the techniques, but it also allows us to predict future techniques that will fit a combination of parameters not used yet.

   With the information provided in this classification we have studied and identified three kinds of relations between the techniques: composition, generalization and set relations. This study allows us to answer questions like: Is one technique more general than another technique? Is the slice produced by one technique included in the slice of another technique? Is one technique composed of other techniques?

3. Algorithmic Debugging Techniques.

   We have presented a technique to improve algorithmic debugging by combining it with program slicing. Firstly, we have adapted some program slicing concepts

(and proven some of their properties) to the context of augmented redex trails. Based on this adaptation, we have introduced an algorithm to slice ARTs and proven its completeness. We have slightly modified the slicing algorithm for the case of algorithmic debugging in order to optimise the number of nodes in slices so that only profitable nodes to build execution trees are included in the slice. With the information provided by the user the introduced algorithm produces a slice which only contains the relevant parts of the execution tree, thus reducing the number of questions.

We have introduced three new strategies and some optimizations for algorithmic debugging. Less YES first tries to improve heaviest first and divide by YES & query tries to improve D&Q by considering previous answers of the oracle during the search. Dynamic weighting search allows the user to specify the exact part of an equation which is wrong. This extra information can produce a much more accurate debugging session.

We have introduced an algorithm to combine different strategies, and we have classified and studied their costs in order to determine how they should be combined in different contexts. The study has determined which strategies are (on average) better. In particular, the result of the study points out heaviest first, less YES first, Hirunkitti and divide by YES & query and Hat Delta as the most efficient. The theoretical comparative has produced objective criteria to dynamically change the strategy used during a debugging session depending on the ET' structure.

8.2 Future Work

There are several interesting directions for continuing the research presented in this thesis. Let us briefly enumerate some of them:

1. Profiling Techniques.

   In Chapter 3, we only sketched the use of logical variables for reasoning about the complexity of a program. This idea can be further developed and, perhaps, combined with partial evaluation so that we get a simpler representation of the time-equations for a given expression. The extension of our developments to cope with higher-order functions seems essential to be able to analyze typical lazy functional (logic) programs. For instance, this can be done by *defunctionalization*, i.e., by including an explicit (first-order) application operator.

2. Program Slicing.

   There are several interesting topics for future work concerning the slicing technique presented in Chapter 5. On the one hand, we plan to define appropriate

extensions in order to cover the additional features of Curry programs (higher-order, constraints, built-in functions, etc). On the other hand, we plan to investigate the definition of appropriate methods for the *static* slicing of functional logic programs. In this case, we think that our approach could still be used but a sort of *abstract* redex trail becomes necessary for guaranteeing its finiteness even when no input data are provided.

Another interesting extension of this work consists in adapting our developments to *pure* functional languages. For instance, Haskell tracers (e.g., Sparud and Runciman [1997]; Wallace *et al.* [2001]) usually rely on some program transformation to instrument a source program so that its execution—under the standard semantics—generates the associated redex trail. The inclusion of program positions within these approaches can also be done in a similar way as in our approach. Then, the implementation of a dynamic slicer mainly amounts to implement an algorithm to compute reachable nodes in a graph.

For the specialization technique we plan to extend the current method in order to perform more aggressive simplifications (i.e., a form of amorphous slicing). This extended form of slicing can be very useful in the context of pervasive systems where resources are limited and, thus, code size should be reduced as much as possible. Again, another promising topic is the definition of program specialization based on *static* slicing rather than on *dynamic* slicing. In particular, it would be interesting to establish the strengths and weaknesses of each approach.

3. Algorithmic Debugging.

   The classification and comparison of algorithmic debugging strategies presented in Chapter 7 must be further investigated through empirical experiments with real programs. An empirical comparison of all the strategies would allow us to determine a weighting for their combination. With the knowledge acquired from this experiment we will be able to approximate the strategies' weights and to determine how they should change and on which factors this change depends.

This thesis has been written in English, which is not the author's native language. The author apologizes for any mistakes which may be due to this circumstance.

Acknowledgements

Chapter 9

Acknowledgements

When I started to work in this thesis, my motivation lied me making me think that it was going to be easy. Some time later, I was completely lost in the middle of nothing: without any idea of how to step forward, or if there was any sense in my work. This horrible feeling has invaded me many times, but I have always found a hand that hold me and encouraged me to go on.

These hands are the real authors of this work.

The unconditional and continuous support of my parents has always hold me, and I am completely sure that it will continue holding me for the rest of my life. This is one of these debts that you know that you will never be able to paid.

Probably, none of my family will never understand a word of this thesis, but they have always being asking me about its problems and my solutions. This love is one of the hands that I use to remember that I cannot defraud them.

Also my friends, Amparo, Carlos, Lolita, Vivi, and, fortunately, many others—who often know what I think much better than I do—are able to transform my biggest problems into tiny sand grains with their friendly advices.

But all these hands together were not enough!

Sometimes, I had to fight with technical problems that were much bigger than I was. In these situations I only had to knock on one of the front red doors of knowledge and get an incredible precise answer. Yes, I never was alone: I had a suit of brains, each of them an expert in its field, helping me to find the path to hide solutions. By door order: Cesar Ferri, José Orallo, Santiago Escobar, Salvador Lucas, María José Ramírez, María Alpuente, Javier Oliver, Marisa Llorens, Carlos Herrero, Pepe Iborra, Raúl Gutierrez, Antonio Bella, Beatriz Alarcón, Salvador Tamarit, Gustavo Arroyo and Diego Cheda.

All of them are my colleagues, who always help me when I need it, and who I admire for their work. Thanks indeed to all of you.

I have a special thank for three people that have been very close to me during the thesis. They had the ability to make my work easier.

First, I want to thank my formerly officemate Alicia Villanueva for her friendly support and patience. Secondly, I want to thank my colleague Guadalupe Ramos who has provide me many ideas in our discussions, and who always listened to me with incredible patience, indeed when he was sure that I was wrong and he was right. I am sure you will succeed with your projects. I will miss you a lot. Thirdly, I am greatly thankful to my friend Vicent Estruch who is a model of person that I will always admire. I feel very lucky to be his friend.

All of them constitute the ELP research group that I am proud to belong to.

I also want to thank my coauthors and colleagues for their valuable contributions to this thesis. Specially to Claudio Ochoa and Demis Ballis for being as they are. During the first year they gave me power for the rest of the thesis. Also to Michael Hanus, Frank Huch, Bernd Braßel and Sebastian Fischer for giving me the first push in Kiel. Finally, to Olaf Chitil, Thomas Davie and Yong Luo for kindly inviting me to their university and make my research period at Canterbury a pleasure.

When something finishes, you always think in how it started. I will always thank the patience of Isidro Ramos and Pepe Carsí who introduced me in the research world for the first time.

Finally, there has been a hand who was the most strong holding me. It is the infinite tender love that I continuously get from Vanessa. She made me ignore the problems because I always knew that it really did not matter if I was going to succeed on my work. After all, she was going to be there. Proud of me...

Valencia, 2007
Josep Silva

Agradecimientos

Chapter 10

Agradecimientos

Cuando comencé a trabajar en esta tesis, mi motivación me mintió haciéndome creer que iba a ser una tarea sencilla. Algún tiempo después, estaba completamente perdido en medio de la nada: sin saber cómo seguir adelante, o si había algún sentido en mi trabajo. Este horrible sentimiento me ha invadido muchas veces, pero siempre he encontrado una mano que me ha levantado y me ha animado a seguir.

Estas manos son los verdaderos autores de este trabajo.

El continuo e incondicional apoyo de mis padres siempre me ha ayudado a continuar, y estoy totalmente seguro de que continuará haciéndolo durante el resto de mi vida. Esta es una de esas deudas que sabes que nunca podrás pagar.

Probablemente, nadie en mi familia entenderá nunca una palabra de esta tesis; sin embargo, siempre me han estado preguntando acerca de los problemas que encontraba y las soluciones que iba planteando. Este amor es una de las manos que utilizo para recordar que no puedo defraudarles.

También mis amigos, Amparo, Carlos, Lolita, Vivi, y, afortunadamente, muchos otros—que a menudo saben lo que pienso mucho mejor que yo mismo—son capaces de transformar mis mayores problemas en insignificantes granos de arena con sus cariñosos consejos.

Pero todas estas manos juntas no fueron suficientes!

Algunas veces, tuve que enfrentarme con problemas técnicos que eran mucho más grandes que yo. En esos momentos, sólamente tuve que llamar a una de las puertas rojas del conocimiento que tengo en frente, y obtener una increiblemente precisa respuesta. Sí, nunca estuve sólo: Tenía una batería de cerebros, cada uno de ellos experto en una materia, ayudándome a encontrar el camino a las respuestas ocultas. Por orden de puerta: Cesar Ferri, José Orallo, Santiago Escobar, Salvador Lucas, María José Ramírez, María Alpuente, Javier Oliver, Marisa Llorens, Carlos Herrero, Pepe Iborra, Raúl Gutierrez, Antonio Bella, Beatriz Alarcón, Salvador Tamarit, Gustavo

Arroyo y Diego Cheda.

Todos ellos son mis compañeros, que siempre me han ayudado cuando lo he necesitado, y a los cuales admiro por su trabajo. Gracias de verdad a todos vosotros.

Tengo un agradecimiento especial para tres personas que han estado muy cerca de mi durante la tesis. Ellas tuvieron la habilidad de hacer mi trabajo más fácil.

Primero, quiero agradecer a mi antigua compañera de despacho Alicia Villanueva su ayuda y paciencia. Segundo, quiero dar las gracias a mi colega José Guadalupe Ramos que me ha dado buenas ideas durante nuestras conversaciones, y que siempre me ha escuchado con increible paciencia, incluso cuando estaba seguro de que él tenía razón y yo me equivocaba. Estoy seguro de que tendrás éxito con tus proyectos. Te echaré mucho de menos. Tercero, le estoy enormemente agradecido a mi amigo Vicent Estruch que es un modelo de persona, y al cual siempre admiraré. Me siento muy afortunado de ser su amigo.

Todos ellos constituyen el grupo de investigación ELP al cual estoy orgulloso de pertenecer.

También quiero agradecer a mis coatores y colegas por sus valiosas contribuciones a esta tesis. Especialmente a Demis Ballis y Claudio Ochoa por ser como son. Durante el primer año de tesis me dieron energias suficientes para el resto de la tesis. También a Michael Hanus, Frank Huch, Bernd Braßel y Sebastian Fischer por darme el primer empujón en Kiel. Finalmente, a Olaf Chitil, Thomas Davie y Yong Luo por invitarme amablemente a su universidad y hacer que mi periodo de investigación allí fuera un placer.

Cuando algo termina, siempre piensas en cómo empezó. Siempre agradeceré a Isidro Ramos y Pepe Carsí el haberme introducido en el mundo de la investigación.

Finalmente, ha habido una mano sujetándome más fuerte que todas las demás. Es el infinito y tierno amor que continuamente recibo de Vanessa. Ella me ha hecho ignorar los problemas porque siempre he sabido que no importaba si al final iba a tener éxito o no en mi trabajo. Después de todo, ella iba a estar ahi. Orgullosa de mi...

Valencia, 2007
Josep Silva

To German Vidal,
my advisor, my instructor, my mentor, my director;
my tutor, my monitor, my teacher, my professor;
my leader, my guide, my trend and my friend.
The man who stepped on my work:
fetter my theorems, counterexampled my proofs,
criticized my presentations,
and never valued my implementations.
Who made me read when I wanted to write.
Who shut me up when I wanted to talk.
Who always told me that I was wrong,
who made me leave my country for long
and live where I never wanted to go.
Who introduced me to people I admired or so.
Who made me stop when I wanted to run.
Who was serious when I wanted fun.
Who forced me to study things I thought were useless the most.
The man who found me when I was lost.
Without any apparent reason
converted my life into a hell, into a prison,
and made feel like a slave.
This man made me enter into a black cave,
a maze without any exit: research.
He made me mind, and search,
where there was nothing to find...
And, when I couldn't resist,
when I was about to desist,
he took my hand and told me that
I was a doctor.

Bibliography

Agrawal H., 1991. Towards Automatic Debugging of Computer Programs. Technical Report Ph.D. Thesis, Purdue University.

Agrawal H., DeMillo R.A., and Spafford E.H., 1991. An Execution-Backtracking Approach to Debugging. *IEEE Softw.*, 8(3):21–26.

Agrawal H., Horgan J.R., Krauser E.W., and London S., 1993. Incremental Regression Testing. In *Proc. of the Conference on Software Maintenance (ICSM'93)*, pages 348–357. IEEE Computer Society, Washington, DC, USA.

Albert E., 2001. *Partial Evaluation of Multi-Paradigm Declarative Languages: Foundations, Control, Algorithms and Efficiency*. Ph.D. thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.

Albert E., Antoy S., and Vidal G., 2001. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042.

Albert E., Hanus M., and Vidal G., 2002. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1).

Albert E., Huch M.H.F., Oliver J., and Vidal G., 2005. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829.

Albert E., Silva J., and Vidal G., 2003. Time Equations for Lazy Functional (Logic) Languages. In *Proc. of the 2003 Joint Conference on Declarative Programming (AGP 2003)*, pages 13–24. Università degli Studi di Reggio Calabria (Italy).

Albert E. and Vidal G., 2001. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*. To appear.

Albert E. and Vidal G., 2002. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'01)*, pages 148–167. Springer LNCS 2372.

Alpuente M., Hanus M., Lucas S., and Vidal G., 1999. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of the Fourth ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 273–283. ACM Press.

Antoy S., 1992. Definitional Trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632.

Antoy S., Echahed R., and Hanus M., 2000. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822.

Antoy S. and Hanus M., 2000. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems*, pages 171–185. Springer LNCS 1794.

Av-Ron E., 1984. *Top-Down Diagnosis of Prolog Programs*. Ph.D. thesis, Weizmanm Institute.

Baader F. and Nipkow T., 1998. *Term Rewriting and All That*. Cambridge University Press.

Bates S. and Horwitz S., 1993. Incremental Program Testing Using Program Dependence Graphs. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 384–396. ACM Press, New York, NY, USA.

Bergeretti J. and Carré B., 1985. Information-Flow and Data-Flow Analysis of While-Programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61.

Binkley D., 1995. Reducing the Cost of Regression Testing by Semantics Guided Test Case Selection. In *Proc. of the International Conference on Software Maintenance (ICSM'95), Opio (Nice), France, October 17-20, 1995*, pages 251–.

Binkley D., 1998. The Application of Program Slicing to Regression Testing. *Information and Software Technology*, 40(11-12):583–594.

Binkley D. and Gallagher K.B., 1996. Program Slicing. *Advances in Computers*, 43:1–50.

Binkley D. and Harman M., 2004. A Survey of Empirical Results on Program Slicing. *Advances in Computers*, 62:105–178.

Binkley D., Horwitz S., and Reps T., 1995. Program Integration for Languages with Procedure Calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35.

Binks D., 1995. *Declarative Debugging in Gödel*. Ph.D. thesis, University of Bristol.

Biswas S., 1997a. A Demand-Driven Set-Based Analysis. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 372–385. ACM Press.

Biswas S., 1997b. *Dynamic Slicing in Higher-Order Programming Languages*. Ph.D. thesis, Department of CIS, University of Pennsylvania.

Bjerner B. and Holmström S., 1989. A Compositional Approach to Time Analysis of First-Order Lazy Functional Programs. In *Proc. of Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM Press.

Bourdoncle F., 1993. Abstract Debugging of Higher-Order Imperative Languages. *ACM SIGPLAN Notices*, 28(6):46–55.

Braßel B., Hanus M., Huch F., Silva J., and Vidal G., 2004a. Runtime Profiling of Functional Logic Programs. In *Proc. of the 14th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 178–189.

Braßel B., Hanus M., Huch F., and Vidal G., 2004b. A Semantics for Tracing Declarative Multi-Paradigm Programs. In *Proc. of the 6th Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'04)*, pages 179–190. ACM Press.

Caballero R., 2005. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13. ACM Press, New York, USA.

Caballero R., 2006. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*, pages 63–76. Electronic Notes in Theoretical Computer Science.

Canfora G., Cimitile A., and De Lucia A., 1998. Conditioned Program Slicing. *Information and Software Technology*, 40(11-12):595–608. Special issue on program slicing.

Canfora G., Cimitile A., De Lucia A., and Lucca G.A.D., 1994. Software Salvaging Based on Conditions. In *Proc. of the International Conference on Software Maintenance (ICSM'94), Victoria, BC, Canada, 1994*, pages 424–433.

Chen T.Y. and Cheung Y.Y., 1993. Dynamic Program Dicing. In *Proc. of the International Conference on Software Maintenance (ICSM'93)*, pages 378–385.

Chitil O., 2001. A Semantics for Tracing. In *13th Int'l Workshop on Implementation of Functional Languages (IFL'01)*, pages 249–254. Ericsson Computer Science Laboratory.

Chitil O., 2005. Source-Based Trace Exploration. In *16th Int'l Workshop on Implementation of Functional Languages (IFL'04)*, pages 126–141. Springer LNCS 3474.

Chitil O. and Luo Y., 2006. Proving the Correctness of Declarative Debugging for Functional Programs. In *Trends in Functional Programming (TFP'06)*.

Choi J.D., Miller B., and Netzer R., 1991. Techniques for Debugging Parallel Programs with Flowback Analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530.

Comini M., Levi G., and Vitiello G., 1994. Abstract Debugging of Logic Programs. In L. Fribourg and F. Turini, editors, *Proc. Logic Program Synthesis and Transformation and Metaprogramming in Logic 1994*, volume 883 of *Lecture Notes in Computer Science*, pages 440–450. Springer-Verlag, Berlin.

Conway T., Henderson F., and Somogyi Z., 1995. Code Generation for Mercury. In *In Proc. of the International Logic Programming Symposium*, pages 242–256.

Cousot P. and Cousot R., 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252.

Cutillo F., Fiore P., and Visaggio G., 1993. Identification and Extraction of "Domain Independent" Components in Large Programs. In *In Proc. of the Working Conference on Reverse Engineering (WCRE'93)*, pages 83–92.

Davie T. and Chitil O., 2005. Hat-delta: One Right Does Make a Wrong. In A. Butterfield, editor, *Draft Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, page 11. Tech. Report No: TCD-CS-2005-60, University of Dublin, Ireland.

Davie T. and Chitil O., 2006. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*.

De Lucia A., 2001. Program slicing: Methods and applications. In *Proc. of First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149. IEEE Computer Society Press, Los Alamitos, California, USA.

De Lucia A., Fasolino A.R., and Munro M., 1996. Understanding Function Behaviors through Program Slicing. In A. Cimitile and H.A. Müller, editors, *Proc. of 4th Workshop on Program Comprehension*, pages 9–18. IEEE Computer Society Press.

Dershowitz N. and Jouannaud J., 1990. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam.

Ferrante J., Ottenstein K., and Warren J., 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

Field J., Ramalingam G., and Tip F., 1995. Parametric Program Slicing. In *Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 379–392. ACM Press.

Fritzson P., Shahmehri N., Kamkar M., and Gyimóthy T., 1992. Generalized Algorithmic Debugging and Testing. *LOPLAS*, 1(4):303–322.

Futamura Y., 1999. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391. Reprint of article in *Systems, Computers, Controls* 1971.

Gallagher K. and Lyle J., 1991. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761.

Gallagher K. and Lyle J., 1993. Software Safety and Program Slicing. In *Proc. of 8th Annual Conference on Computer Assurance Practical Paths to Assurance (Compass'93)*, pages 71–80. National Institute of Standards and Technology, Gaithersburg, MD.

Gaucher F., 2003. Slicing Lustre Programs. Technical report, VERIMAG, Grenoble.

Gill A., 2000. Debugging Haskell by Observing Intermediate Data Structures. In *Proc. of the 4th Haskell Workshop*. Technical report of the University of Nottingham.

Giovannetti E., Levi G., Moiso C., and Palamidessi C., 1991. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377.

Gorlick M. and Kesselman C., 1987. Timing Prolog Programs without Clock. In *Proc. of the 4th Symposium on Logic Programming (SLP'87)*, pages 426–434.

Gupta R. and Soffa M., 1995. Hybrid Slicing: An Approach for Refining Static Slices Using Dynamic Information. In *Proc. of Third ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'95)*, pages 29–40. ACM Press.

Gyimóthy T., Beszédes Á., and Forgács I., 1999. An Efficient Relevant Slicing Method for Debugging. In *ESEC/FSE-7: Proc. of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 303–321. Springer-Verlag, London, UK.

Hall R., 1995. Automatic Extraction of Executable Program Subsets by Simultaneous Dynamic Program Slicing. *Autom. Softw. Eng*, 2(1):33–53.

Hallgren T., 2003. Haskell Tools from the Programatica Project. In *Proc. of the ACM Workshop on Haskell (Haskell'03)*, pages 103–106. ACM Press.

Hanus M., 1994. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628.

Hanus M., 1997. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM.

Hanus M., 2000. A Functional Logic Programming Approach to Graphical User Interfaces. In *Proc. of the Int'l Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753.

Hanus M., 2006a. Curry: An Integrated Functional Logic Language (Version 0.8.2 of March 28, 2006). Available at: `http://www.informatik.uni-kiel.de/~curry/`.

Hanus M., 2006b. FlatCurry: An Intermediate Representation for Curry Programs. Available at: `http://www.informatik.uni-kiel.de/~curry/flat`. Accessed on November 13th 2006.

Hanus M., Antoy S., Engelke M., Höppner K., Koj J., Niederau P., Sadre R., and Steiner F., 2006. PAKCS 1.7.3: The Portland Aachen Kiel Curry System—User Manual. Technical report, U. Kiel, Germany.

Hanus M. and Prehofer C., 1999. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75.

Harman M. and Danicic S., 1997. Amorphous Program Slicing. In *Proc. of the Fifth International Workshop on Program Comprehension*, pages 70–79. IEEE Computer Society Press.

Harman M., Danicic S., Sivagurunathan Y., and Simpson D., 1996. The Next 700 Slicing Criteria. In *Proc. of 2nd UK Workshop on Program Comprehension (Durham University, UK, July 1996)*.

Harman M. and Gallagher K., 1998. Program Slicing. *Information and Software Technology*, 40(11-12):577–582. Special issue on program slicing.

Harman M. and Hierons R., 2001. An Overview of Program Slicing. *Software Focus*, 2(3):85–92.

Hermenegildo M., Puebla G., Bueno F., and López-García P., 2003. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, pages 127–152. Springer LNCS 2694.

Hirunkitti V. and Hogger C.J., 1993. A Generalised Query Minimisation for Program Debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*, pages 153–170. Springer LNCS 749.

Hong H., Lee I., and Sokolsky O., 2005. Abstract Slicing: A New Approach to Program Slicing Based on Abstract Interpretation and Model Checking. In *Proc. of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 25–34. IEEE Computer Society.

Horwitz S., Reps T., and Binkley D., 1988. Interprocedural Slicing Using Dependence Graphs. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 35–46. ACM Press SIGPLAN Notices, volume 23 (7), Atlanta, GA.

Jackson D. and Rollins E., 1994. Chopping: A Generalization of Slicing. Technical Report CS-94-169, Carnegie Mellon University, School of Computer Science.

Jones N., 1996. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503.

Jones N., Gomard C., and Sestoft P., 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ.

Jones S. and Métayer D.L., 1989. Compile-Time Garbage Collection by Sharing Analysis. In *Proc. Int'l Conf. on Functional Programming Languages and Computer Architecture*, pages 54–74. ACM Press.

Kamkar M., 1993. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. Ph.D. thesis, Linköping University.

Kamkar M., Fritzson P., and Shahmehri N., 1993. Interprocedural Dynamic Slicing Applied to Interprocedural Data How Testing. In *Proc. of the International Conference on Software Maintenance (ICSM'93)*, pages 386–395. IEEE Computer Society, Washington, DC, USA.

Klop J., 1992. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press.

Kokai G., Harmath L., and Gyimothy T., 1997. Algorithmic Debugging and Testing of Prolog Programs. In *Workshop on Logic Programming Environments*, pages 14–21.

Kokai G., Nilson J., and Niss C., 1999. GIDTS: A Graphical Programming Environment for Prolog. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 95–104. ACM Press.

Korel B. and Laski J., 1988. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163.

Kuck D., Kuhn R., Padua D., Leasure B., and Wolfe M., 1981. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*, pages 207–218.

Lakhotia A., 1993. Rule-Based Approach to Computing Module Cohesion. In *Proc. of the 15th International Conference on Software Engineering (ICSE'93)*, pages 35–44. IEEE Computer Society Press, Los Alamitos, CA, USA.

Lanubile F. and Visaggio G., 1993. Function Recovery Based on Program Slicing. In *Proc. of the International Conference on Software Maintenance (ICSM'93)*, pages 396–404. IEEE Computer Society, Washington, DC, USA.

Laski J., 1990. Data Flow Testing in STAD. *J. Syst. Softw.*, 12(1):3–14.

Launchbury J., 1993. A Natural Semantics for Lazy Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press.

Lindenstrauss N. and Sagiv Y., 1997. Automatic Termination Analysis of Logic Programs. In *Proc. of 12th Int'l Conf. on Logic Programming (ICLP97)*, pages 63–77.

Liu Y. and Gómez G., 1998. Automatic Accurate Time-Bound Analysis for High-Level Languages. In *Proc. of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embbeded Systems*, pages 31–40. Springer LNCS 1474.

Liu Y. and Stoller S., 2003. Eliminating Dead Code on Recursive Data. *Science of Computer Programming*, 47:221–242.

Lloyd J.W., 1994. Combining Functional and Logic Programming Languages. In *Proc. of the 1994 International Symposium on Logic programming (ILPS'94)*, pages 43–57. MIT Press, Cambridge, MA, USA.

Loogen R., López-Fraguas F., and Rodríguez-Artalejo M., 1993. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of 5th Int'l Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714.

López-Fraguas F. and Sánchez-Hernández J., 1999. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631.

Lyle J., 1984. *Evaluating Variations on Program Slicing for Debugging.* Ph.D. thesis, University of Maryland.

Lyle J. and Weiser M., 1987. Automatic Program Bug Location by Program Slicing. In *Proc. of the Second International Conference on Computers and Applications*, pages 877–882. Peking, China.

MacLarty I., 2005. *Practical Declarative Debugging of Mercury Programs.* Ph.D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne.

Maeji M. and Kanamori T., 1987. Top-Down Zooming Diagnosis of Logic Programs. Technical Report TR-290, ICOT, Japan.

Merlo E., Girard J., Hendren L., and de Mori R., 1993. Multi-Valued Constant Propagation for the Reengineering of User Interfaces. In *Proc. of the International Conference on Software Maintenance (ICSM'93)*, pages 120–129. IEEE Computer Society, Washington, DC, USA.

Métayer D.L., 1988. Analysis of Functional Programs by Program Transformations. In *Proc. of 2nd France-Japan Artificial Intelligence and Computer Science Symposium.* North-Holland.

Moreno-Navarro J. and Rodríguez-Artalejo M., 1992. Logic Programming with Functions and Predicates: The Language Babel. *Journal of Logic Programming*, 12(3):191–224.

Muller M., Muller T., and Roy P.V., 1995. Multiparadigm Programming in Oz. In *Proc. of Workshop on the Future of Logic Programming, International Logic Programming Symposium (ILPS'95).*

Naish L., 1997a. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming*, 1997(3).

Naish L., 1997b. A Three-Valued Declarative Debugging Scheme. In *Proc. of Workshop on Logic Programming Environments (LPE'97)*, pages 1–12.

Nilsson H., 1998. *Declarative Debugging for Lazy Functional Languages*. Ph.D. thesis, Linköping, Sweden.

Nilsson H. and Fritzson P., 1994. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370.

Nilsson H. and Sparud J., 1997. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150.

Ning J., Engberts A., and Kozaczynski W., 1994. Automated Support for Legacy Code Understanding. *Commun. ACM*, 37(5):50–57.

Ochoa C., Silva J., and Vidal G., 2004a. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 123–134. ACM Press.

Ochoa C., Silva J., and Vidal G., 2004b. Program Specialization Based on Dynamic Slicing. In *Proc. of Workshop on Software Analysis and Development for Pervasive Systems (SONDA'04)*, pages 20–31. Southantom University.

Ochoa C., Silva J., and Vidal G., 2005. Lightweight Program Specialization via Dynamic Slicing. In *Proc. of the Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 1–7. ACM Press.

Ochoa C., Silva J., and Vidal G., 2006. A Slicing Tool for Lazy Functional Logic Programs. In *Proc. of the 10th European Conference on Logics in Artificial Intelligence (JELIA'06)*, pages 498–501. Springer LNCS 4160.

Osterweil L. and Fosdick L., 1976. DAVE: A Validation Error Detection and Documentation System for Fortran Programs. *Software Practice and Experience*, 6(4):473–486.

Ottenstein K. and Ottenstein L., 1984. The Program Dependence Graph in a Software Development Environment. In *Proc. of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE'84)*, pages 177–184. ACM Press, New York, NY, USA.

Paakki J., Gyimóthy T., and Horváth T., 1994. Effective Algorithmic Debugging for Inductive Logic Programming. In S. Wrobel, editor, *Proc. of the International Conference on Inductive Logic Programming (ILP'94)*, volume 237 of *GMD-Studien*, pages 175–194. Gesellschaft für Mathematik und Datenverarbeitung MBH.

Pereira L.M., 1986. Rational Debugging in Logic Programming. In *Proc. on Third International Conference on Logic Programming*, pages 203–210. Springer-Verlag LNCS 225, New York, USA.

Peyton-Jones S.L., editor, 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Platoff M., Wagner M., and Camaratta J., 1991. An Integrated Program Representation and Toolkit for the Maintenance of C Programs. In *Proc. of the International Conference on Software Maintenance (ICSM'91)*, pages 129–137. IEEE Computer Society Press.

Ramalingam G. and Reps T., 1991. A Theory of Program Modifications. In *Proc. of the International Colloquium on Combining Paradigms for Software Development (CCPSD'91)*, pages 137–152. Springer-Verlag LNCS 494, New York, USA.

Ramalingam G. and Reps T.W., 1992. Modification Algebras. In *Proc. of the Second International Conference on Methodology and Software Technology (AMAST'91)*, pages 547–558. Springer-Verlag, London, UK.

Reps T., 1991. Algebraic Properties of Program Integration. *Sci. Comput. Program*, 17(1-3):139–215.

Reps T. and Bricker T., 1989. Illustrating Interference in Interfering Versions of Programs. *SIGSOFT Softw. Eng. Notes*, 14(7):46–55.

Reps T., Horwitz S., Sagiv M., and Rosay G., 1994. Speeding up Slicing. In *Proc. of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20. ACM Press.

Reps T. and Rosay G., 1995. Precise Interprocedural Chopping. *SIGSOFT Software Engineering Notes*, 20(4):41–52.

Reps T. and Turnidge T., 1996. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110.

Rodrigues N. and Barbosa L., 2005. Slicing Functional Programs by Calculation. In *Proc. of the Dagstuhl Seminar on Beyond Program Slicing*. Seminar n. 05451, Schloss Dagstuhl.

Rosendahl M., 1989. Automatic Complexity Analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156. ACM Press, New York, NY, USA.

Rothermel G. and Harrold M., 1994. Selecting Tests and Identifying Test Coverage Requirements for Modified Software. In *Proc. of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'94)*, pages 169–184. ACM Press, New York, NY, USA.

Sands D., 1990. Complexity Analysis for a Lazy Higher-Order Language. In *Proc. of 3rd European Symposium on Programming (ESOP'90)*, pages 361–376. Springer LNCS 432.

Sands D., 1996. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234.

Sansom M. and Jones L.P., 1997. Formally Based Profiling for Higher-Order Functional Languages. *ACM Trans. Program. Lang. Syst.*, 19(2):334–385.

Sansom P. and Peyton-Jones S., 1997. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385.

Shapiro E., 1982. *Algorithmic Program Debugging*. MIT Press.

Shimomura T., 1992. The Program Slicing Technique and its Application to Testing, Debugging and Maintenance. *Journal of the Information Processing Society of Japan*, 33(9):1078–1086.

Silva J., 2006. Algorithmic Debugging Strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, pages 134–140.

Silva J. and Chitil O., 2006. Combining Algorithmic Debugging and Program Slicing. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 157–166. ACM Press.

Somogyi Z. and Henderson F., 1999. The Implementation Technology of the Mercury Debugger. *Electronic Notes in Theoretical Computer Science*, 30(4).

Sparud J. and Runciman C., 1997. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292.

Sterling L. and Shapiro E., 1986. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press.

Tip F., 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189.

Venkatesh G.A., 1991. The Semantic Approach to Program Slicing. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 107–119. ACM Press, New York, NY, USA.

Vidal G., 2002. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62. ACM Press.

Vidal G., 2003. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Proc. of the 12th Int'l Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'02)*, pages 219–237. Springer LNCS 2664.

Voigtländer J., 2002. Conditions for Efficiency Improvement by Tree Transducer Composition. In S. Tison, editor, *13th International Conference on Rewriting Techniques and Applications, Copenhagen, Denmark, Proceedings*, volume 2378 of *LNCS*, pages 222–236. Springer-Verlag.

Wadler P., 1988. Deforestation: Transforming Programs to Eliminate Trees. In *Proc of the European Symp. on Programming (ESOP'88)*, pages 344–358. Springer LNCS 300.

Wadler P., 1998. Why No One Uses Functional Languages. *SIGPLAN Notices*, 33(8):23–27.

Wallace M., Chitil O., Brehm T., and Runciman C., 2001. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170. Universiteit Utrecht UU-CS-2001-23.

Weiser M., 1979. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* Ph.D. thesis, The University of Michigan.

Weiser M., 1981. Program Slicing. In *Proc. of the Fifth International Conference on Software Engineering (ICSE'81)*, pages 439–449.

Weiser M., 1983. Reconstructing Sequential Behavior from Parallel Behavior Projections. *Inf. Process. Lett.*, 17(3):129–135.

Weiser M., 1984. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

Xu B., Qian J., Zhang X., Wu Z., and Chen L., 2005. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36.

York Functional Programming Group, 2006. Tracing and Profiling. URL: http://www.cs.york.ac.uk/fp/profile.html. Accessed on November 13th 2006.