

A Declarative Debugger for Haskell

Bernard James Pope

Submitted in total fulfilment of the requirements of
the degree of Doctor of Philosophy

December 2006

DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING
THE UNIVERSITY OF MELBOURNE
VICTORIA, AUSTRALIA

Abstract

This thesis is about the design and implementation of a debugging tool which helps Haskell programmers understand why their programs do not work as intended. The traditional debugging technique of examining the program execution step-by-step, popular with imperative languages, is less suitable for Haskell because its unorthodox evaluation strategy is difficult to relate to the structure of the original program source code. We build a debugger which focuses on the high-level logical meaning of a program rather than its evaluation order. This style of debugging is called *declarative debugging*, and it originated in logic programming languages. At the heart of the debugger is a tree which records information about the evaluation of the program in a manner which is easy to relate to the structure of the program. Links between nodes in the tree reflect logical relationships between entities in the source code. An error diagnosis algorithm is applied to the tree in a top-down fashion, searching for causes of bugs. The search is guided by an oracle, who knows how each part of the program should behave. The oracle is normally a human — typically the person who wrote the program — however, much of its behaviour can be encoded in software.

An interesting aspect of this work is that the debugger is implemented by means of a program transformation. That is, the program which is to be debugged is transformed into a new one, which when evaluated, behaves like the original program but also produces the evaluation tree as a side-effect. The transformed program is augmented with code to perform the error diagnosis on the tree. Running the transformed program constitutes the evaluation of the original program plus a debugging

session. The use of program transformation allows the debugger to take advantage of existing compiler technology — a whole new compiler and runtime environment does not need to be written — which saves much work and enhances portability.

The technology described in this thesis is well-tested by an implementation in software. The result is a useful tool, called `buddha`, which is publicly available and supports all of the Haskell 98 standard.

Declaration

This is to certify that

- the thesis comprises only my original work towards the PhD except where indicated in the Preface,
- due acknowledgment has been made in the text to all other material used,
- the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, and appendices.

Preface

This thesis is based in part on the original work presented in the following four peer reviewed papers:

- B. Pope. Declarative debugging with Buddha. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer-Verlag, 2005. (Invited paper).
 - The debugging example in Chapter 3 is taken from this paper.
 - The program transformation algorithm in Chapter 5 is an improved version of the one discussed in this paper.
 - The method of observing values for printing in Chapter 6 is based on the technique described in this paper, and also the following paper.
- B. Pope and L. Naish. Practical aspects of declarative debugging in Haskell-98. In *Proceedings of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240. ACM Press, 2003.
 - The re-evaluation scheme and method of debugging of I/O computations in Chapter 7 are based on this paper, though greatly improved in the current presentation.

- B. Pope and L. Naish. A program transformation for debugging Haskell-98. *Australian Computer Science Communications*, 25(1):227–236, 2003.
 - This paper describes an earlier version of the debugging program transformation. The treatment of higher-order functions in this paper is the basis of the scheme discussed in Chapter 5 and Chapter 6.
- B. Pope and L. Naish. Specialisation of higher-order functions for debugging. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
 - This paper describes an earlier approach to transforming higher-order functions. It is made obsolete by the new transformation described in Chapter 5. We discuss this alternative approach in Chapter 8.

Lee Naish contributed to the development of this thesis. He was the secondary author on three of the above-mentioned papers. In addition, the following parts are based extensively on his work:

- The definition of buggy nodes in Chapter 3.
- The concepts of intended interpretations and inadmissibility in Chapter 4.
- The use of quantifiers to handle partial values in derivations in Chapter 4.

The following items have not been previously published:

- The more flexible definition of evaluation dependency in Chapter 3.
- The performance measurements in Chapters 5 and 7.
- The improved scheme for piecemeal EDT construction in Chapter 7.

Acknowledgments

Lee Naish, my supervisor, is a pioneer in the field of declarative debugging and I am honoured to work with him for so many years on this topic. Our relationship began in 1997, when I was starting my honours year and keen to work in functional programming. To my great fortune Lee had a project in debugging and was kind enough to let me join in. It was quite clear that a single honours year was not long enough, so I jumped at the chance to continue with a PhD. In my quest for the elusive debugger I have stumbled, taken many wrong paths, and backtracked frequently over well-trodden ground. Such meandering would test even the most patient of souls, yet Lee was happy to let me explore, and always ready to offer sound advice and directions when I needed them. Thank you Lee for supporting me on this long journey, I have enjoyed your friendship and guidance every step of the way.

I am also greatly indebted to Harald Søndergaard, my co-supervisor, for leading me to functional programming so many years ago and showing me how stimulating Computer Science can be. I will never forget the day that I became a functional programmer. It was in Harald's class, he was explaining with his usual enthusiasm the elegance of a line of Haskell code — though he made it seem like it was a line of poetry. From then on I was converted.

I must also thank Lee and Harald for proof reading earlier drafts of this thesis.

Along the way I have had many functional programming comrades and it would be remiss of me not to give them praise. Foremost is Kevin Glynn, a truly great friend, keen functional programmer, and Wolves supporter. I have many fond mem-

ories of my time spent with Kevin, in the office, on the soccer pitch (or in the stadium) and also in the pub. I hope that one day we will share the same continent again. I would like to thank all the members of the functional programming group for their support. A PhD can be an isolating experience and it was encouraging to have so many people to lend their ears. Thanks also to the many Mercurians who looked on with interest and provided much inspiration and competition to us lazy Haskellites.

For almost all of this degree I lived with Steve Versteeg, a student himself, who showed me how to live life to the fullest extent. We enjoyed many adventures together and helped one another forget the harsh realities of deadlines and progress reports.

Though there are many people to thank, none are more deserving of my gratitude than my beloved family.

I am fortunate to have such kind and generous parents, Jan and Brian Pope. They are my foundation in life and I am eternally indebted to them for their endless support. I want to thank them for working so hard to allow me to pursue my interests and encouraging me in whatever I choose to do (even when I go about it so slowly). Thanks also to my sister Gabrielle Pope who has stood by me and urged me along in a way that only a big sister can.

Outside of the offices and halls of the Computer Science department my life changed in a most remarkable way. I met, and later married, my darling wife Hui Nie Fu. She has been a constant source of love and happiness, and without her help I would not have completed this thesis. I look forward to our life together, especially to exploring the world and visiting her family (my new family) in Selatpanjang.

Finally, I would like to thank the Australian Federal Government for supporting this thesis with an Australian Postgraduate Award.

Contents

1	Introduction	1
1.1	No silver bullet	1
1.2	Bugs	2
1.3	Debugging	3
1.4	Debugging Haskell	7
1.5	Declarative debugging	9
1.6	Research problems	10
1.6.1	Portability	10
1.6.2	Scalability	11
1.7	Methodology	12
1.8	Contributions	13
1.9	Structure of the thesis	14
2	Haskell	15
2.1	Introduction	15
2.1.1	Outline of this chapter	16
2.2	Key features of Haskell	16
2.2.1	Syntax	16
2.2.2	Purity	17
2.2.3	Higher-order functions	18

2.2.4	Static types with polymorphism and overloading	19
2.2.5	Non-strict evaluation	20
2.2.6	Implicit memory management	22
2.3	Dynamic semantics	23
2.3.1	Term rewriting	23
2.3.2	Graph reduction	27
2.4	Monads and I/O	31
2.5	Pragmatic features	34
2.6	Final remarks	36
3	Declarative Debugging	39
3.1	Introduction	39
3.1.1	Outline of this chapter	41
3.2	Background	41
3.3	The EDT and wrong answer diagnosis	43
3.3.1	Properties of the EDT	43
3.3.2	Identification of buggy equations	45
3.3.3	Example EDTs	46
3.3.4	Big-step EDTs	48
3.3.5	An interface to the EDT	49
3.3.6	Wrong answer diagnosis	51
3.3.7	Constructing the EDT	53
3.4	An example debugging session	53
3.5	Higher-order functions	65
3.6	Pattern bindings	69
3.7	Final remarks	71
4	Judgement	75
4.1	Introduction	75
4.1.1	Outline of this chapter	77

4.2	Preliminaries	77
4.3	Partial values	78
4.4	Inadmissibility	79
4.5	Higher-order functions	81
4.6	Final remarks	85
5	EDT Construction	87
5.1	Introduction	87
5.1.1	Outline of this chapter	88
5.2	The general scheme	88
5.2.1	An example	88
5.3	Implementing the EDT	92
5.4	Transforming function bindings	94
5.5	Transforming pattern bindings	96
5.6	Transforming higher-order code	101
5.6.1	Lambda abstractions	101
5.6.2	Partial applications	102
5.7	The transformation rules	109
5.7.1	Abstract syntax	109
5.7.2	Notation used in the transformation rules	111
5.7.3	Declarations	112
5.7.4	Expressions	113
5.7.5	Types	115
5.8	Correctness	116
5.8.1	Semantics	116
5.8.2	EDT correctness	119
5.9	Performance	124
5.10	Final remarks	127

6	Observing Values	129
6.1	Introduction	129
6.1.1	Outline of this chapter	130
6.2	The requirements of observation	130
6.3	A Haskell implementation	131
6.4	An observation primitive	133
6.4.1	Cyclic values	136
6.5	Observing functional values	138
6.5.1	Intensional printing of functions	139
6.5.2	Extensional printing of functions	142
6.5.3	Combining both styles	145
6.6	Optimisation	147
6.7	Displaying large values	151
7	Practical Considerations	153
7.1	Introduction	153
7.1.1	Outline of this chapter	154
7.2	I/O	155
7.2.1	Printing IO values	155
7.2.2	EDT dependencies for IO code	159
7.2.3	Exceptions	162
7.2.4	Performance	164
7.3	Trusted functions	167
7.4	Piecemeal EDT construction	169
7.4.1	The basic idea	171
7.4.2	Problems	171
7.4.3	Implementation	173
7.4.4	Performance	181
7.4.5	Improvements	183
7.5	Final remarks	190

8	Related Work	193
8.1	Introduction	193
8.1.1	Outline of this chapter	194
8.2	Diagnostic writes	194
8.2.1	The trace primitive	195
8.2.2	The Haskell Object Observation Debugger	196
8.2.3	Graphical Hood	203
8.2.4	Limitations of diagnostic writes	204
8.3	Declarative debugging	205
8.3.1	Freya	208
8.3.2	Program transformation	212
8.4	Hat	218
8.5	Step-based debugging	224
8.6	Monitoring semantics	225
8.7	Randomised testing with QuickCheck	229
8.8	Final remarks	231
8.8.1	Classification of the different tools	231
8.8.2	Usability	233
9	Conclusion	235
9.1	Review	235
9.1.1	Chapter summary	237
9.1.2	The evolution of <code>buddha</code>	239
9.2	Future work	244
9.2.1	Printing free lambda-bound variables	244
9.2.2	Support for language extensions	246
9.2.3	Integration with an interpreter-style interface	248
9.2.4	Customisation	249
9.2.5	Improved EDT traversal	250
9.2.6	A more flexible EDT	250

Bibliography	251
A An example transformed program	265
B Higher-order functions in the intensional style	267

List of Figures

1.1	Source-to-source program transformation.	12
2.1	Computing the magnitude of a vector in Haskell.	23
2.2	Comparing normal order and applicative order term reduction sequences.	25
2.3	Graph reduction of ‘double (3 * 2)’.	27
2.4	Two candidate graph implementations of the fixed point operator. .	28
2.5	Graph reduction resulting in a cyclic data-structure.	29
2.6	A graph illustrating the sharing in the definition of <code>fibs</code>	30
3.1	Three example EDTs for the same computation, exhibiting various reduction step sizes in their nodes.	47
3.2	Nilsson’s definition of direct evaluation dependency.	49
3.3	An abstract interface to the EDT in Haskell.	50
3.4	The top-down left-to-right wrong answer diagnosis algorithm. . . .	51
3.5	A (buggy) program for converting numbers in base 10 notation to other bases.	55
3.6	An example EDT diagram produced by the ‘draw edt’ command. .	60
3.7	An EDT for the program in Figure 3.5.	61
3.8	A small buggy program with higher-order functions.	65

3.9	Two EDTs for the same computation, illustrating the different ways that functional values can be displayed.	66
3.10	An EDT with functions printed in extensional style.	67
5.1	A program for computing the area of a circle.	89
5.2	Code for constructing an EDT node.	96
5.3	A method for constructing EDT nodes for pattern bindings, which simulates the method used in Freya.	98
5.4	Code for constructing EDT nodes for pattern bindings.	100
5.5	An EDT with functions printed in intensional style.	105
5.6	Abstract syntax for core Haskell.	110
5.7	Transformation of declarations.	111
5.8	Transformation of expressions and alternatives.	114
5.9	Transformation of types.	115
5.10	Relative performance of the transformed program compared to the original, when an empty EDT is constructed.	126
6.1	A type class for generic observation.	132
6.2	Optimised transformation of function declarations.	148
6.3	Optimised transformation of types.	148
6.4	Optimised transformation of function application.	150
7.1	I0 dependencies (extensional).	160
7.2	I0 dependencies (intensional).	161
7.3	I0 dependencies (intensional, except <code>lamBind</code>).	161
7.4	Memory usage versus input size with I0 tabling enabled.	164
7.5	Running time with I0 tabling enabled relative to the original program.	165
7.6	Percentage of nodes which come from functions in trusted standard libraries.	168
7.7	Transitions between contexts.	179
7.8	EDT size versus depth bound.	182

7.9	Memory usage (relative to the original program) versus depth bound.	183
7.10	Running time (relative to the original program) versus depth bound.	184
7.11	Maximum depth of the EDT for the example programs.	185
8.1	Evaluation of an observed expression.	201
8.2	The GHood interface.	204
8.3	Freya’s construction of the EDT during reduction.	210
8.4	Redex trail for the factorial computation.	221
8.5	EDT for the factorial computation.	222
9.1	Computing the roots of a quadratic equation in Haskell.	244
9.2	An EDT for the program in Figure 9.1.	245
B.1	An example EDT diagram produced by the ‘draw edt’ command. .	269

Chapter 1


Introduction

“Four bullet holes in your starboard wing, sir,” the sergeant reported, “and one’s gone through your engine cowling and lodged in your magneto casing.”
“Sergeant, those aren’t bullet holes,” replied Barry; “a gremlin did that.”
And so, there on the Dover-London road, a new word was born.

The Gremlins

[Dahl, 1943]

1.1 No silver bullet

OMPUTER programming — especially on a large scale — is fraught with difficulty. In one of his many famous essays on Software Engineering, Brooks [1975] observed:

Digital computers are themselves more complex than most things people build: They have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do.

Likewise, a scaling-up of a software entity is not merely a repetition of the same elements in larger sizes, it is necessarily an increase in the number of different elements. In most cases, the elements interact with

each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly.

So far our most potent antidotes to the complexity of programming are high-level programming languages, and discipline. However, as yet no silver bullet has emerged, and sadly we continue to write and use programs which behave in unintended — and occasionally catastrophic — ways. Many have argued that we are experiencing a *software crisis* [Wayt Gibbs, 1994].

1.2 Bugs

Today the word *bug* is synonymous with computer malfunction. Yet the notion is quite an old one; the etymology of *bug* can be traced back at least as far as the 1800s [Shapiro, 1987]. In common parlance a bug is an unintended behaviour of a machine that is in some way related to a fault in its design or construction. Nasty bugs exhibit seemingly unpredictable patterns of behaviour. These tend to arise in systems with many parts, which interact in complex ways, making them extremely hard to explain. Such failures can be so chaotic that they might as well be caused by hoards of cackling green devils.

The truth — as far as I am aware of it — is that computer bugs are not the responsibility of gremlins, but largely of those who write programs. A famous quip, widely attributed to Nathaniel Borenstein, goes:

The most likely way for the world to be destroyed, most experts agree,
is by accident. That's where we come in; we're computer professionals.
We cause accidents.

In a way, the word *bug* distances the programmer from the fault. We find ourselves exclaiming “there’s a bug in my program!” with the same indignation as “there’s a fly in my soup!,” thus begging the question “who put it there?” Zeller prefers to use *defect* to name the incorrect parts of the program code (putting the blame back on

the programmer), and *failure* to name the externally observable malfunction which occurs when a defective program is executed [Zeller, 2005].

A study conducted in 2002 by the National Institute of Standards and Technology (a government organisation in the United States of America) estimated the annual cost of software failures to American economy at roughly 59.5 billion US dollars, which was about 0.6 percent of the country's gross domestic product at the time [NIST, 2002]. Major contributing factors in the total cost are the lost productivity of software users, and the increased resources expended in software production. Clearly computer bugs are a big problem, but what can we do about them?

1.3 Debugging

There are really two questions that need to be asked:

1. How can we make our programs less defective?
2. If a program has defects how can we find and fix them?

One school of thought is that programs should be proven correct, thus eliminating the need for debugging altogether [Hoare, 1969]. The old adage that “prevention is better than a cure” also rings true in program development. Dijkstra was particularly vocal on this point [Dijkstra, n.d.]:

Already now, debugging strikes me as putting the cart before the horse:
instead of looking for more elaborate debugging aids, I would rather try
to identify and remove the more productive bug-generators!

However, there are several problems with this approach as a complete solution:

- Proofs must be made against a formal specification of the program. This leads to the problem of debugging specifications, which is in general no simpler than debugging programs [Shapiro, 1983].

- Proofs can be difficult to develop, communicate and verify [De Millo et al., 1979].
- Right or wrong, a large amount of programming is experimental, starting with only imprecise specifications. Experimental programs may eventually evolve into more formally prescribed systems. Nonetheless, there may be a lengthy period of development where the intended behaviour of the program is only partially defined. Hence there is very little that can be proved correct.
- Programming languages may be only informally defined and most widely used contemporary languages have many pragmatic features which make proofs very difficult.
- Current proof techniques do not scale to large programs.

In the absence of correctness proofs covering entire programs, which may be unattainable for the reasons stated above, the next best strategy for ensuring the reliability of software is testing. There are many ways to go about testing, but they all share the same goal, which is to find input values which cause the program to behave incorrectly. When a program is found to fail on a particular test case the next thing we want to do is find out why, and then fix it. Explaining the reason for program failure and fixing the problem is the domain of debugging, and that is — in broad terms — the topic of this thesis.

In the early 1970s the structured programming style became popular, and it has had a big influence on programming methodology to this day. One of the most important ideas of structured programming is that programs should be decomposed into small logical units, which have a single point of entry, and whose intended behaviour is easily understood. Complex programming tasks are broken up into smaller, more tractable problems, which are solved individually, and re-combined to form a complete solution. Most popular languages since the 1970s have encouraged structured programming one way or another. Example programming units include: *procedures* in imperative languages, *functions* in functional languages, and *predicates*

in logic languages.

In this thesis we focus on functional languages, so we will hereafter refer specifically to functions as the units of a program.

Structured programming also provides a useful framework for debugging. We imagine our program as a complex machine made up of many interconnected parts. If the program fails on some input value, we know that one or more of the individual functions must be faulty. We also know that a terminating execution only calls each function in a finite number of different ways. Therefore a program execution can be regarded as a search space whose elements are individual function calls. Each function has an intended behaviour in the mind of the programmer, which can be used to judge the correctness of each call. The intended behaviour can be described in numerous ways, but the simplest, and probably most common way, is in terms of the relationship between the function's input and output values. Debugging is therefore a search through this space for calls which point to defective functions.

In practice, finding bugs in the failed executions of programs can be extremely labour intensive for two key reasons:

1. The internal behaviour of program execution is hidden, and thus hard to test manually.
2. The search space can grow very large.

The first point relates to the fact that programs are written in high-level languages but are translated into low-level machine languages for execution. Typically the machine is the hardware of the computer, although it could also be a simulated machine in software. In either case we can probe the operations of the underlying machine, but this tells us very little about the program as we know it, because many vestiges of the original source code are lost in translation.

The second point relates to the fact that programs usually involve loops of computation, such that a group of functions may repeatedly invoke one another in a cyclic fashion. Each function in the group may be called in a large number of different ways throughout the execution of the loop, and loops may be arbitrarily nested.

Even a program with only a small number of functions can produce an enormous search space.

There is a third point that exacerbates the difficulty of debugging, though it is much worse in some languages than others: interference. This occurs when the behaviour of a function is affected by an event which is not characterised by an input or output value. Interference is rife in languages with a lax attitude to side-effects. The trouble is that interference makes the behaviour of a function call highly dependent on its context. This in turn complicates debugging because a judgement about the correctness of an individual function call might require the user to consider a great number of other function calls at the same time; they may not even know *a priori* which other function calls are relevant. Also, if a function has side-effects, they must be considered as part of the behaviour of the function, in addition to its output. Side-effects can make reasoning about correctness more difficult because their relative ordering is significant. For instance, a sequence of output statements can have the wrong behaviour even though each individual statement is correct on its own, simply because the order is wrong. A corollary is that languages which limit the extent of side-effects will tend to be easier to debug.

The underlying philosophy of this thesis is that the burden of debugging can be greatly reduced if we adopt a language without side-effects, and employ a semi-automated debugging algorithm to systematically order the search space and allow mechanical search.

1.4 Debugging Haskell

Suffice it to say that the extensional properties of a functional program (what it computes as its result) are usually far easier to understand than those of the corresponding imperative one. However, the intensional properties of a functional program (how it computes its result) can often be much harder to understand than those of an imperative one, especially in the presence of higher order functions and lazy evaluation.

Heap profiling of lazy functional programs

Runciman and Wakeling [1993]

The feature-set of a language can affect the kinds of bugs that are encountered, so research into debugging tools is usually done in the context of a particular programming paradigm.

We consider the problem of debugging Haskell programs. Debugging Haskell is interesting as a topic for research because:

- Haskell is a promising language which provides several features that promote safe programming practices. Despite this relative safety, Haskell programs are not immune from bugs, and there is a need for debugging tools.
- The fundamentals of traditional debugging technology are in conflict with Haskell's key computational features: non-strict evaluation, and higher-order functions. Though many debuggers exist in the mainstream, they are of limited efficacy for Haskell.

The mainstream of programming is dominated by the so-called *imperative* languages. Programs in this paradigm are composed of commands which are stateful and destructive, hence the precise evaluation order of commands is very important. As a result, imperative programs tend to be rigidly sequential, and programmers are forced to be acutely aware of how the structure of their code relates to the steps performed by the computer as the program executes. Debugging tools for imperative programs follow suit.

In contrast, functional programs are data-oriented. They focus on the construction and transformation of data objects by (non-destructive) function application. Functional languages are often said to be *declarative* in nature. This means that the basic blocks of programs — the functions — state a relationship between their input and output values, but they do not explicitly give an order in which their operations should take place. An advantage of this model is that it allows more freedom in the way that programs are executed; lazy evaluation is a prime example. The downside is that functional programmers tend to have only a fuzzy idea of how their programs behave, step-by-step. Logical relationships, such as *X depends on Y*, which are evident in the source code, and are fundamental to the programmer’s reasoning, may not be obvious in the execution order. This means that step-wise debuggers are a bad match for such languages. The difficulty of applying existing debugging technology to lazy languages has been known since their conception. For example see the discussion in Hall and O’Donnell [1985].

One of the hallmarks of functional programming is higher-order functions. As the saying goes: functions are *first class*. Unfortunately, higher-order functions can make debugging more difficult:

- Functions are abstract data types in Haskell, which means they can only be observed indirectly via their behaviour.
- Higher-order functions complicate the relationship between the dynamic pattern of function calls at runtime and the structure of the source code.
- Determining the correctness of higher-order function invocations can be mentally taxing for the user of the debugger.

A significant challenge in the design of debugging systems for Haskell is how to reduce the cognitive load on the user, especially when many higher-order functions are involved. This is an issue that has seen little attention in the design of debugging systems for mainstream imperative languages because higher-order code is much less prevalent there.

1.5 Declarative debugging

Declarative debugging is based on a simple principle: a function is judged to be defective if some call to that function is incorrect (produces the wrong output value for its corresponding input values), *and* that call does not depend on any other incorrect function calls. Such a call is said to be buggy. The dependency relation between function calls allows a tree structure to be imposed onto the debugging search space. Nodes in the tree represent individual calls, and a special root node corresponds to the initial call which is made at the start of the program. Each node is the parent of zero or more nodes. The evaluation of the function call in a parent directly depends on all and only those function calls in its children. We adopt the terminology of Nilsson and Sparud [1997], and call this tree an *Evaluation Dependency Tree* (EDT).

Given an EDT, we can employ a diagnosis algorithm which automates the search for buggy nodes.

The purely functional nature of Haskell makes it well suited to declarative debugging for two reasons:

1. Functions make good building blocks because they are easily composed. This encourages a bottom-up style of programming where complex functions are built by connecting together the inputs and outputs of simpler ones. This leads to programs which are highly structured and well suited to hierarchical decomposition.
2. The answer produced by a given function call is totally determined by the values of its inputs; there are no side-effects. Therefore the correctness of an individual function invocation can be considered in isolation from its evaluation context.

Conversely, declarative debugging is well suited to Haskell because the structure of the EDT can reflect the structure of the source code, thus hiding the complicated operational aspects of lazy evaluation from the user.

Despite its many attractive features, declarative debugging is not the best solution for finding all kinds of bugs. In particular it is not well suited to performance tuning. The main reason is that it is much more difficult for the user to judge the correctness of a program's time and space behaviour on a call-by-call basis. We generally do not have a precise notion how much time or space an individual call is likely to need. Instead we are much better at tackling performance tuning by other means, such as the use of dedicated statistical profiling tools.

1.6 Research problems

The foundations of declarative debugging were established by Shapiro [1983] in the context of pure logic programming. Since then, non-strict purely functional languages have emerged and flourished, and owing to many similarities between the two paradigms, various people have investigated the potential for declarative debugging in the functional setting.

Whilst the topic has been reasonably well explored, the ultimate goal of usable debugging tools has remained elusive. The most significant roadblocks are portability and scalability. Portability relates to the independence of the tool to its working environment, including the computer hardware, the operating system, and the compiler. Scalability relates to the class of all programs that can be effectively debugged with the tool.

1.6.1 Portability

One way to make a portable debugger is to write it in the same language as its input programs. If someone wants to debug a program written in language X , it is a fair bet that they will have an implementation of X that works in their environment. If we are lucky, the debugger will also be able to debug itself, though this is not a primary goal. The trouble is that the language in question may not always be ideal for this task.

Debuggers are unusual programs in that they are highly reflective. In other

words, they are designed to observe and perhaps manipulate the behaviour of other programs, the debuggees. When the debugger and the debuggee share the same language, we encounter the difficult issue of self-reflection. Few general-purpose languages are good at this.

The success of Shapiro’s work is due largely to the expressive reflection facilities that are built into Prolog. Haskell is much more limited in this regard, particularly because of the discipline imposed by its type system. Haskell’s types promote safe programming practices because they provide a static consistency check, that ensures data abstraction boundaries are not broken. The reflective facilities of Prolog are difficult to mix with Haskell’s type system because they allow a program to undermine the data abstraction boundaries that the types are supposed to uphold.

1.6.2 Scalability

The scalability of a debugger has two dimensions:

1. How much of the total feature-set of the language is supported by the debugger?
2. How expensive, in terms of resource consumption, is the tool?

The most challenging features of Haskell for debugging are lazy evaluation and higher-order functions.

Prohibitive space usage is another significant hurdle. Haskell data objects inhabit many different representations during the execution of a program, starting from program expressions, and ending as computed values. A heuristic of declarative debugging is that it is easier for the user of the debugger to determine the correctness of a function call if its argument and result values are displayed in their final representation. Lazy evaluation tends to intersperse incremental evaluation over numerous data objects, which makes it very difficult to predict when an individual object will reach its final representation. The simplest solution is to be conservative, and postpone all debugging until the debuggee has terminated. This

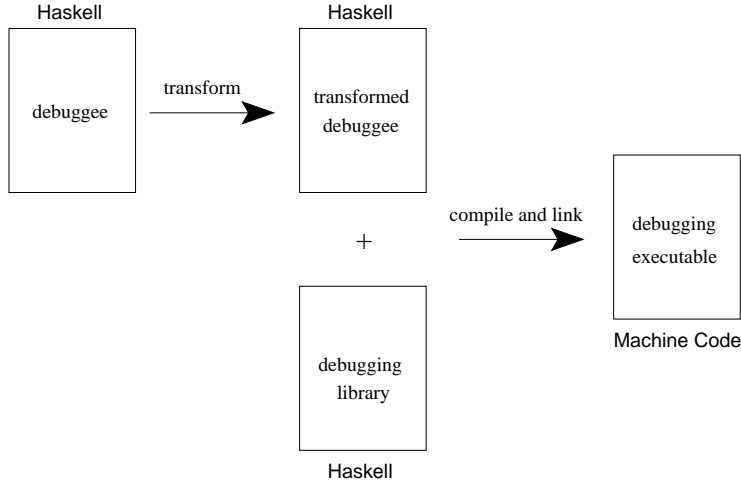


Figure 1.1: Source-to-source program transformation.

ensures that all observed values will be displayed in their final representation. Unfortunately this means that the whole EDT must be created prior to debugging. Since the EDT maintains a reference to every intermediate data object computed by a program, its size grows proportionally to the length of the program execution. On a modern machine the entire available memory can be exhausted in a matter of seconds, limiting the debugger to all but the shortest program runs. Numerous solutions to the space problem are considered in this thesis.

1.7 Methodology

We employ a source-to-source program transformation, where the debuggee is transformed into a new Haskell program which computes both the value of the debuggee *and* an EDT suitable for declarative debugging. The transformed program is compiled and linked with a bug diagnosis library and the whole package forms the debugger. This process is illustrated in Figure 1.1.

Source-to-source program transformation has two key features that make it attractive:

1. The output is Haskell which enhances the portability of the debugger.
2. The transformation algorithm is syntax directed, and relatively simple to implement, especially when compared to the complexity of a whole compiler or interpreter.

1.8 Contributions

The main contribution of this thesis is a source-to-source program transformation which facilitates declarative debugging for non-strict functional programming languages. We have demonstrated the feasibility of our approach by building a working debugger, called `buddha`, which supports full Haskell 98. We believe this was the first declarative debugger to support the whole language.

We provide a flexible approach to debugging higher-order code. For each function in the program the user has the option of printing higher-order instances of the function in one of two ways, we call them the *intensional* and *extensional* styles. The intensional style is based on a function’s term representation. The extensional style is based on the complete set of argument/result mappings for a function in given program run. The extensional style is particularly helpful in situations where new functions are built dynamically by a program. In such cases the intensional style can become unwieldy and difficult for the user to understand. We are the first to incorporate the extensional style into declarative debugging.

In order to support the extensional style of printing functions we have extended the traditional definition of the EDT that is found in the literature, incorporating a more general notion of “evaluation dependency”.

On top of the extensional style we provide a novel technique for displaying I/O values which facilitates declarative debugging of I/O computations. Earlier attempts at building declarative debuggers for non-strict purely functional languages have not

tackled this problem.

We show that the execution time overheads introduced by the transformation are within reasonable limits on a selection of non-trivial programs, and that the space usage of the EDT can be reduced by adapting previously established techniques.

We provide a formal definition of the transformation algorithm as a series of rules over a core Haskell syntax. The transformation employed by `buddha` follows these rules very closely, which makes it a remarkably concise implementation.

1.9 Structure of the thesis

The rest of this thesis proceeds as follows. Chapter 2 provides a thorough introduction to Haskell. Readers who are already familiar with Haskell, or something similar, may wish to skim this chapter. Chapter 3 introduces the key concepts of declarative debugging, and shows how `buddha` works in an example debugging session. It also formalises the concept of an EDT and shows how it is closely related to the concept of evaluation dependency. Chapter 4 discusses the intricacies of judging the correctness of function calls in the light of lazy evaluation and higher-order functions. Chapter 5 defines the program transformation employed by `buddha`, and measures its performance on a sample of five non-trivial programs. Chapter 6 shows how `buddha` implements a universal printer for Haskell data objects. Chapter 7 considers the practical aspects of debugging full Haskell 98, in particular debugging I/O computations, and keeping resource usage within reasonable limits. Chapter 8 summarises related work. Chapter 9 suggests future avenues for research, and concludes.



Chapter 2

Haskell

The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.

Why functional programming matters

[Hughes, 1989]

2.1 Introduction



ASKELL is a high-level general-purpose programming language. It is the product of a community spread across the world, consolidating many years of research in functional programming languages. Amongst many other things, it is a springboard for new language technology, a tool for program development, and a vehicle for education. And of course it is a central character in this thesis.

This chapter aims to give an overview of Haskell, concentrating on its most interesting and unique aspects, those which set it apart from the majority of other popular programming languages in use today. Haskell is far too big to describe in detail in one chapter. Indeed the Language Report — the authoritative reference for Haskell — is some 270 pages long in book form [Peyton Jones, 2002]. The best that can be hoped for in the present context is to capture the essence of the

language. Readers who are already familiar with Haskell are advised to skip directly to Chapter 3.

2.1.1 Outline of this chapter

The rest of the chapter proceeds as follows. In Section 2.2 we discuss the key features of Haskell. Then we turn our attention to semantics in Section 2.3. In Section 2.4 we consider the use of monads to integrate input and output (I/O) with the purely functional paradigm, and also abstract over various kinds of computational features. In Section 2.5 we discuss two pragmatic features of Haskell which have proven useful in the construction of `buddha`. In Section 2.6 we highlight some helpful reference material from the literature.

2.2 Key features of Haskell

2.2.1 Syntax

At its core, Haskell’s syntax is essentially the language of the Lambda Calculus [Church, 1941, Barendregt, 1984]. Layered on top of that are various programmer-friendly constructs, such as named declarations, data types, pattern matching, modules and so forth; most of which are heavily influenced by Turner’s family of languages, especially Miranda [Turner, 1985]. Perhaps the most striking feature of Haskell’s syntax — especially for those who are familiar with mainstream imperative languages — is its minimal use of punctuation. Function application is simply the juxtaposition of terms, and indentation provides grouping and delineation without the need for semi-colons and braces. A complete definition of the syntax, plus desugaring rules into a simple core language, are provided in the Language Report. For more information about Haskell’s heritage, including its syntactic inheritance, see Hudak [1989].

2.2.2 Purity

Informally, functions in Haskell behave like functions in mathematics: they are simply mappings from inputs to outputs. Unlike imperative languages, there are no side-effects — a function application cannot evaluate to, say, an integer and along the way print a message to the terminal. Actually, this view is somewhat naive, and Haskell functions differ from “mathematical” ones in a couple of ways. First, Haskell functions can diverge, by failing to terminate, or by some other kind of runtime error, such as attempting to divide a number by zero. Second, Haskell programs must at some point perform side-effects because they would be useless otherwise. The very nature of all computer programs is to change the state of their environment, *e.g.* print an image on the screen, write a file to the disk drive, or send a message over the network. The presence of divergent programs is not normally grounds for considering a language impure,¹ however side-effects are a different story. On the one hand, pure functions are not allowed to perform side-effects. On the other hand, side-effects are an essential part of every computer program. This is a long-standing problem for purely functional languages. The solution in Haskell is monads [Peyton Jones and Wadler, 1993, Peyton Jones, 2001]. The result is a stratified language, with a pure part and an impure part. The performance of impure side-effecting operations can have no observable effect on the pure part of the language. The role of monads is to interface the pure and impure parts of the program in a safe way. The intriguing thing about this approach is that all of the user’s program can be written with pure functions, and the side-effecting operations, called actions, are performed externally. It is as if the Haskell program computes an imperative program as its result, which is then passed to an external evaluator to make all its actions happen.² We discuss monads in more detail in Section 2.4.

Purely functional languages have a certain degree of theoretical elegance, but

¹Turner [2004] argues for *strong* functional languages, where all functions are totally defined (*i.e.* no divergent programs). In contrast, he classifies languages which permit divergent functions as *weak* functional languages.

²The evaluation of actions and pure functions is interleaved in typical programs, however the separation still holds: the impure parts have no impact on the pure parts.

that is not their only virtue. Purity tends to simplify the difficult task of reasoning about programs. The often promoted feature of pure languages is *referential transparency*, the property that an expression always has the same meaning regardless of the context in which it occurs. This is also a benefit for debugging programs, because the correctness of a program fragment, such as a function definition, can be considered in relative isolation from the rest of the program [Hudak, 1989]. If a definition is correct, all its uses are automatically correct, no matter where they occur. For a compiler, or any code transforming tool, the correctness of program transformations is easier to verify than for an impure language.

Sabry [1998] provides a more formal definition of purely functional languages. His requirements are generally that the language must be a conservative extension of the pure Lambda Calculus (in other words, the language must have functions), and that the meaning of the program is independent of the parameter passing mechanism used (modulo divergence). Under this definition Haskell is pure, as are subsets of Standard ML and Scheme.

2.2.3 Higher-order functions

The hallmark of the functional paradigm, pure and impure, is that functions are first class. This means that functions can be passed around like any other kind of value — they can be arguments or results of other functions and even stored within data structures. Higher-order programming opens up new opportunities for abstraction and generalisation that can make the program more modular and flexible [Hughes, 1989]. Higher-order functions are commonly used in Haskell, as evidenced by the large number of them in the standard libraries, and they are central to many programming idioms, such as monads.

Haskell’s functions are *curried*.³ This means that it is possible to view all functions as if they have only one parameter. A multi-parameter function can be turned

³Functions in the Lambda Calculus are also curried. However, the idea is due to Schönfinkel [1924]. Curry and Feys [1958] made extensive use of the idea and introduced the current notation. The term *currying* is, of course, in honour of Curry, whose first name happens to be Haskell!

into a unary function by having it return a (curried) function as its result. The benefit of currying is that it provides a very concise way to make new functions from old ones by function application. For example, since multiplication is curried, it is possible to write `(3*)`; the result is a new function that multiplies its argument by 3.

Despite the fact that Haskell functions are curried it is normal to talk of function arities that are higher than one. This is because Haskell provides syntactic sugar for function declarations which allows multiple parameters to be named together. For example, consider the `const` function:

```
const x y = x
```

We would usually say that `const` has an arity of two because it has two parameters to the left of the equals sign. When a function is given fewer arguments than its arity, the application is said to be *partial*. When sufficient arguments have been supplied, the application is said to be *saturated*.

2.2.4 Static types with polymorphism and overloading

Haskell is endowed with a rich type system with many novel aspects. Principally, the type system is based on the famous Hindley-Milner algorithm [Hindley, 1969, Milner, 1978], which at compile time attempts to infer types for all expressions in the program. The program is rejected by the compiler if type checking fails. The rigidity of the system is relaxed somewhat by the fact that functions can be polymorphic, meaning that the one definition can operate on many different types of arguments. For example, the list reverse function has type `reverse :: [a] -> [a]`. The double-colon is read as *‘has the type’*, the square brackets denote the type of lists, and the arrow denotes the type of functions. The `a` is a *type variable*, which is implicitly universally quantified over the whole type.

Type inference means that the types of expressions can be calculated without any additional annotations — the programmer is not obliged to tell the compiler

what the types are. The benefit is that program definitions are shorter and simpler, and generally easier to modify.

Functions can be overloaded by the use of type classes [Wadler and Blott, 1989, Hall et al., 1996]. A class specifies an interface which is made up of one or more type signatures. Types are made instances of classes by the provision of functions that implement the interface, specialised to the particular type in question. A classic example is equality. The standard environment of Haskell specifies a class, called `Eq`, that collects all types that have equality defined on their values. The definition of the class looks like this (simplified for presentation):

```
class Eq a where
  (==) :: a -> a -> Bool
```

The class is parameterised over types, by the variable `a`. To make some type T an instance of `Eq` we must provide an implementation of `==` such that each occurrence of `a` in the type scheme is replaced by T . For example, the boolean type with values `True` and `False`, can be made an instance of `Eq` in the following way:

```
instance Eq Bool where
  True == True   = True
  False == False = True
  x == y         = False
```

This instance declaration says what the function `==` means in the type context of booleans, but it says nothing about equality in any other type context. The kind of polymorphism exhibited by `==` is different to that of `reverse`, because the latter has the same behaviour for all type contexts in which it is used, but the former varies, and it may not even be defined for some types.

2.2.5 Non-strict evaluation

Programming languages are often characterised by how they perform parameter passing. In this regard they are said to be either strict or non-strict. A function is strict in an argument if its result is undefined whenever that argument is undefined. For example, consider some function f with one argument. If \perp stands for the

undefined value (*i.e.* a divergent computation), and $f\perp = \perp$, then f is strict in its argument, otherwise it is non-strict in its argument. A strict programming language employs a parameter passing technique which forces all functions to be strict in all of their arguments, whereas a non-strict programming does not.

Strict parameter passing is usually implemented by *eager evaluation*, also called *call-by-value*. That is, when a function call is made, all argument expressions are fully evaluated prior to entering the callee. Most languages are strict for two reasons:

1. Eager evaluation is relatively easy to implement efficiently on stock hardware.
2. The order in which side-effects are executed is more easily related to the structure of the source code under strict evaluation (compared to non-strict evaluation).

Non-strict parameter passing is much more liberal. The most common way of implementing it is *lazy evaluation*, or *call-by-need*. Under lazy evaluation, argument expressions are never evaluated unless they are needed, and if so they are evaluated once only. Multiple uses of an argument share the same value. Sometimes laziness and non-strictness are mistakenly equated. However, other strategies can be used to give non-strict behaviour, such as lenient evaluation [Tremblay, 2001], and hybrids of lazy and eager evaluation [Maessen, 2002, Ennals and Peyton Jones, 2003b].

Haskell is often called a “lazy functional language”, but this is not quite true. It is *non-strict*, though most Haskell implementations are lazy by default. An excellent reference on the topic of evaluation strategies and the pitfalls of confusing laziness with non-strictness is given in Tremblay [2001].⁴

On the surface it would appear that non-strict evaluation, especially the lazy kind, is optimally efficient because argument expressions that are never needed are never evaluated, potentially saving much work. Sometimes this is true, but in practice the advantage is mostly lost because a significant amount of additional complexity is needed in the runtime environment of the language to implement laziness.

⁴Although, he suggests that Haskell is a lazy language!

Modern computer architectures are vigorously optimised for certain types of code sequences and memory usages — especially ones that exhibit a high degree of temporal and spatial locality [Hennessy and Patterson, 1996]. Current day runtime environments for lazy languages are penalised on such hardware because they tend to do a poor job at achieving this kind of locality [Nethercote and Mycroft, 2002]. Paradoxically, experience shows that to be lazy, programs often have to work extra hard. Also, the space usage of non-strict evaluation is often much worse than what it would be under strict evaluation. First, because the size of an unevaluated expression can be much larger than its ultimate value, and second, because unevaluated expressions can unduly retain references to other heap allocated objects that would otherwise be garbage collected.

The true benefit of non-strictness — probably why it wasn't abandoned long ago for efficiency reasons — is that it promotes a more declarative style of programming. Recursive equations are more natural and can be used more liberally in a non-strict setting, allowing for such exotic things and infinite and cyclic data-structures [Tremblay, 2001]. Non-strictness also tends to decouple the interfaces between producers and consumers of data making the program more modular [Hughes, 1989].

2.2.6 Implicit memory management

The management of memory allocation is implicit in Haskell. This means that data values are added and removed by the runtime environment automatically. A technique called *garbage collection* cleans up any data that is no longer needed by the program, reclaiming its memory for future use. This removes a very large burden from the programmer, and also saves programs from a number of nasty memory related bugs. In Haskell, garbage collection is absolutely necessary for productive programming because non-strict evaluation and higher-order code make it very difficult for the programmer to safely manage memory themselves.

```
start = mag [1,2]
mag xs = sqrt (sum (map sq xs))
sum [] = 0
sum (x:xs) = x + sum xs
map f [] = []
map f (x:xs) = f x : map f xs
sq x = x * x
```

Figure 2.1: Computing the magnitude of a vector in Haskell.

2.3 Dynamic semantics

Haskell is non-strict, and thus it permits many different evaluation strategies. However, most implementations are lazy, hence that is the focus of this section. The intention is not to nail down the dynamic semantics of Haskell — indeed no (complete) formal description of it exists in the literature — but rather to introduce certain concepts and terminology that will be important for later parts of the thesis. As it happens, lazy evaluation exhibits all the properties that make such languages hard to debug.

First, we use term rewriting to show the different order in which expressions are reduced using lazy and eager strategies. Then we use graph reduction to show how sharing is normally implemented in lazy languages. We also consider cyclic values.

2.3.1 Term rewriting

Consider the program in Figure 2.1. The function `mag` computes the magnitude of a vector (represented as a list of numbers). It works as follows: each element of the vector is squared, the result is summed, and the square root is taken. Evaluating the program corresponds to demanding the value of `start`. It is assumed that `sqrt`, `+` and `*` are primitives, and that they evaluate their arguments eagerly, in a left-to-right manner.

Term rewriting is a simple way to visualise program evaluation, and is especially useful for comparing different evaluation strategies. The process begins with `start`

which is replaced by its body. The body is then “reduced” until it reaches a final state, called a *normal form*. In general we are not guaranteed to reach a normal form, so the process of reduction may continue forever in some cases. Each step in the reduction represents a simplification of the term from the previous step. The idea is to search in the current term for a reducible expression (*redex*) and replace it with an equivalent, but more evaluated form. A term is in normal form when it has no redexes, however lazy evaluators usually opt for a weaker kind of normal form; more on that later. Function definitions provide reduction rules. Redexes are terms that match the left-hand-side of a rule (the function head). For example, the first equation of `sum`, says that the term ‘`sum []`’ is a redex, and it can be replaced with 0. The second equation for `sum` says that ‘`sum (x:xs)`’ is also a redex, and it can be replaced with ‘`x + sum xs`’, where `x` and `xs` are parameters which can be replaced by arbitrary terms. Reduction rules are assumed for the primitive functions, and in particular, applications of `+`, `*` and `sqrt` do not become redexes until their arguments are fully evaluated numbers.

Given a term with multiple redexes, which one should be reduced first? The *normal order* strategy says to pick the leftmost outermost one. Whereas the *applicative order* strategy says pick the leftmost innermost one. It is well established (at least for the Lambda Calculus) that if the original expression has a normal form then the normal order strategy will find it, whereas the applicative order may not. However, there are terms which can never be reduced to a normal form no matter what order of evaluation is chosen. Choosing the “leftmost outermost” redex corresponds to evaluating a function application without first evaluating the arguments, whilst “leftmost innermost” is the opposite. Thus the normal order is non-strict and the applicative order is strict.

Figure 2.2 shows the reduction of the vector magnitude program using normal order and applicative order strategies. In this case neither strategy is better than the other in terms of the number of reduction steps. What is interesting is the difference in the sequence of reductions. Each new line in the sequence is derived

Normal order (non strict)

```

start
mag (1 : 2 : [])
sqrt (sum (map sq (1 : 2 : [])))
sqrt (sum (sq 1 : map sq (2 : [])))
sqrt (sq 1 + sum (map sq (2 : [])))
sqrt (1 * 1 + sum (map sq (2 : [])))
sqrt (1 + sum (map sq (2 : [])))
sqrt (1 + sum (sq 2 : map sq []))
sqrt (1 + (sq 2 + sum (map sq [])))
sqrt (1 + (2 * 2 + sum (map sq [])))
sqrt (1 + (4 + sum (map sq [])))
sqrt (1 + (4 + sum []))
sqrt (1 + (4 + 0))
sqrt (1 + 4)
sqrt 5
2.236

```

Applicative order (strict)

```

start
mag (1 : 2 : [])
sqrt (sum (map sq (1 : 2 : [])))
sqrt (sum (sq 1 : map sq (2 : [])))
sqrt (sum (1 * 1 : map sq (2 : [])))
sqrt (sum (1 : map sq (2 : [])))
sqrt (sum (1 : sq 2 : map sq []))
sqrt (sum (1 : 2 * 2 : map sq []))
sqrt (sum (1 : 4 : map sq []))
sqrt (sum (1 : 4 : []))
sqrt (1 + sum (4 : []))
sqrt (1 + 4 + sum [])
sqrt (1 + 4 + 0)
sqrt (1 + 4)
sqrt 5
2.236

```

Figure 2.2: Comparing normal order and applicative order term reduction sequences.

by reducing the previous one using the redex indicated by the underline. Both strategies begin in the same way and perform the same reductions when there is only one redex to choose from. However, when there are multiple redexes, they do different things. Perhaps the most salient point, in terms of debugging programs, is that normal order reduction is more difficult to reconcile with the structure of the code than applicative order. This is particularly obvious with the recursive calls in `sum`. In most cases, the programmer's intuition about how the program is evaluated follows the structure of the source code. Statically, `sum` recursively calls itself. In the applicative order, each the the reductions of `sum` occur consecutively, and its argument is always a list in normal form. In the normal order the reductions of `sum` are interspersed with those of `sq`, `*` and `map`, and its argument is not always a list in normal form (in the first two reductions it is a complex expression). The problem is that standard debugging techniques that trace the execution step by step are much less useful for a non-strict language because it is hard for the programmer to relate the order of reductions with their mental model of the program. Also, in the non-strict setting, the arguments and results of function applications will often be complex expressions. Understanding a function's actual behaviour relies on inspecting the result it produced for its given arguments, however this can be more difficult when those values are only partially reduced.

A key feature of *lazy evaluation* is the sharing of argument expressions, though this aspect is missing from the normal order term reduction discussed above. Consider the simple program below:

```
double x = x + x
start = double (3 * 2)
```

Normal order reduction of `start` proceeds as follows:

```
start
double (3 * 2)
(3 * 2) + (3 * 2)
6 + (3 * 2)
6 + 6
12
```

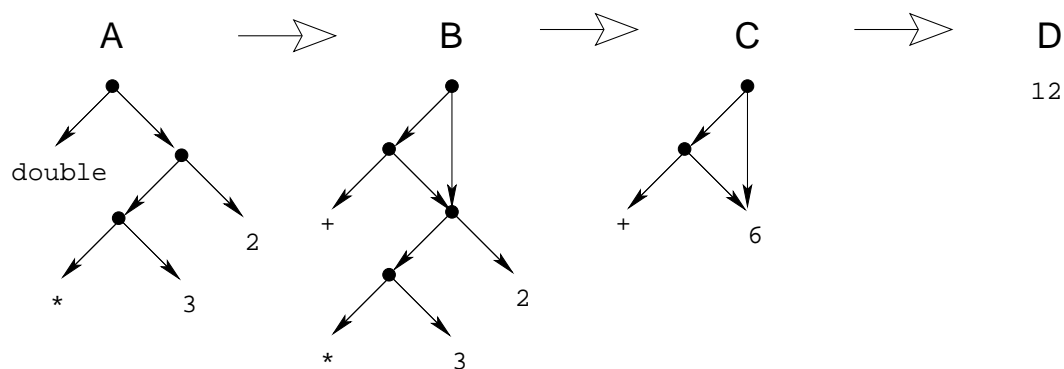


Figure 2.3: Graph reduction of `double (3 * 2)`.

Notice that `double` duplicates its parameter in its body. This causes repeated evaluation of the expression `3 * 2`. Lazy evaluation avoids this redundancy. Most implementations of lazy evaluation are based on *graph reduction* because graphs provide the necessary identity for terms which enables sharing.

2.3.2 Graph reduction

Graph reduction proceeds along the same lines as term reduction. Initially, the program is a complex graph, and redexes are sub-graphs that can be simplified. Figure 2.3 depicts the graph reduction of the example program. Vertices in the graph represent function applications, which are connected to their argument graphs by directed edges, and terminals represent variables or constants. All application nodes are binary due to currying. Notice that the two arguments of `+` share the same graph representation of `3 * 2`. This saves one reduction step over the term rewriting evaluation because the redundant re-evaluation of `3 * 2` is avoided.

Sharing and cyclic structures

Certain infinite values can be represented very compactly by taking advantage of the potential for self-sharing, or cycles, within a graph. The classic example is the infinite list of ones:

```
ones = 1 : ones
```

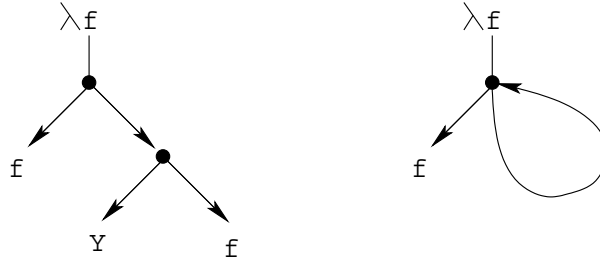


Figure 2.4: Two candidate graph implementations of the fixed point operator.

A non-strict language enables us to write functions which can operate on a finite prefix of this list without causing non-termination. Whether or not `ones` is represented with a cyclic graph depends on how recursion is implemented, and Haskell does not make any specific requirements in this regard. The textbook approach to implementing recursion is to introduce a new function called `Y`, which computes the fixed point of its argument:

$$Y\ f = f\ (Y\ f)$$

Each recursive equation in the program can be turned into non-recursive one by the use of `Y`. The result is that `Y` is the only recursive part of the program, which can be implemented as some kind of primitive operation. For example, `ones` can be made non-recursive in the following way with the help of `Y`:

```
ones' = \r -> 1 : r
ones  = Y ones'
```

A few reductions of `ones` shows the effect of `Y`:

```
ones
Y ones'
ones' (Y ones')
(\r -> 1 : r) (Y ones')
1 : (Y ones')
1 : ones' (Y ones')
1 : ((\r -> 1 : r) (Y ones'))
1 : (1 : (Y ones'))
...
```

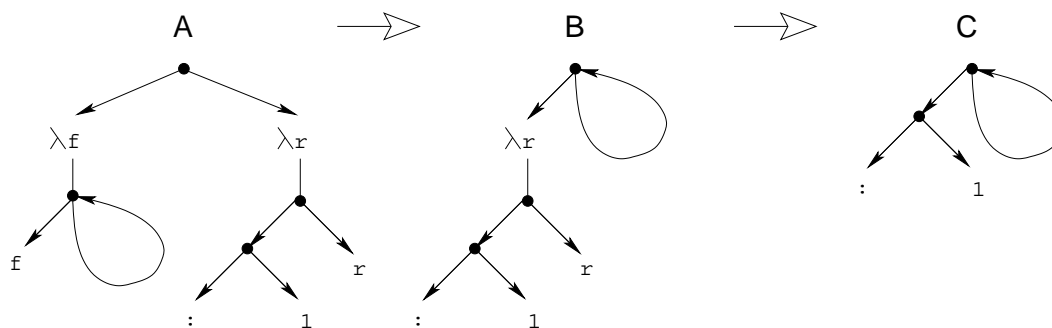



Figure 2.5: Graph reduction resulting in a cyclic data-structure.

How might `Y` be implemented? Figure 2.4 shows two candidate graph encodings. Lambda abstractions are encoded with a normal graph representing the function body extended with a edge connecting the body to the variable bound in the function head. The first representation of `Y` is a very direct translation of the function definition into graph notation, and the second uses cycles in a clever way, giving a more succinct representation of the same function.

Figure 2.5 shows the graph reduction of ‘`Y ones`’’, using the cyclic representation of `Y`. The end result is a cyclic structure. The benefit of the cyclic representation is that the list consumes only a constant amount of space.

The graph reduction of ‘`double (3 * 2)`’ shows that the sharing of graphs can reduce the amount of work needed to evaluate an expression. In that example, the time saved was modest, but sharing can have a dramatic effect on the complexity of a computation. Consider the code below, which computes the infinite list of Fibonacci numbers, called `fibs`:

```
fibs :: [Integer]
fibs = 1 : 1 : zipPlus fibs (tail fibs)

zipPlus (x:xs) (y:ys) = x + y : zipPlus xs ys
zipPlus xs ys         = []

tail (x:xs) = xs
```

Figure 2.6 illustrates the initial graph representation of the body of `fibs`. Notice

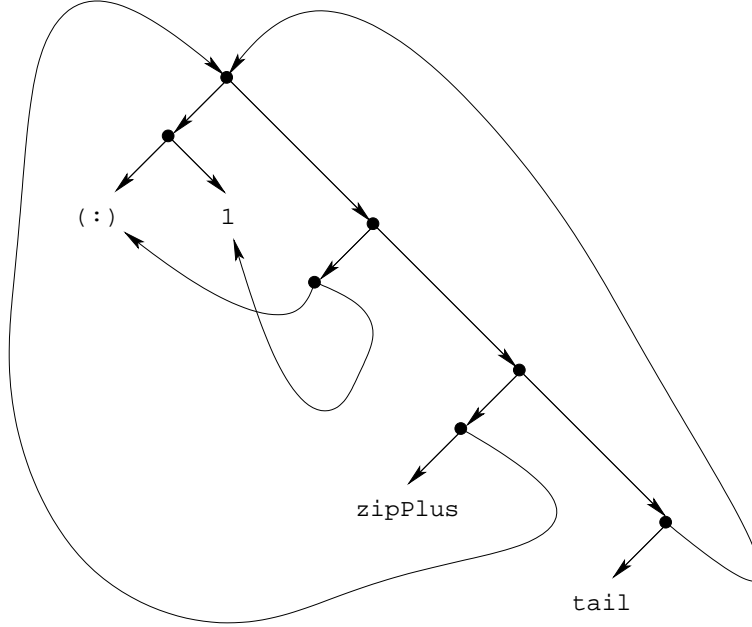


Figure 2.6: A graph illustrating the sharing in the definition of `fibs`.

that the recursive references to `fibs` in the body of the function are represented as edges to the top node of the graph, creating a compact cyclic form. Computing the n th element of that list has time complexity proportional to n , because recursive references to `fibs` are shared and thus not re-computed upon every call. If the recursive references to `fibs` were not shared, the computation of the n th Fibonacci number would be exponential in n , because each new reference to `fibs` produces two more references. We must be careful in the construction of the debugger to preserve sharing in the underlying program, lest we risk a severe performance penalty in some cases.

Weak head normal form

Previously it was stated that reduction continues until the expression is a normal form. Under lazy evaluation this is not quite true; reduction continues until the outermost expression reaches a weaker kind of normal form called *weak head normal form* (WHNF). An expression is in WHNF if it is a manifest function (a lambda

abstraction or let-bound function), a partial application of a manifest function, or if it is an application of a data constructor to zero or more arguments. The body of the lambda abstraction and the arguments of the applications do not themselves have to be normal forms (weak head or otherwise), they can be arbitrary expressions. The effect is that, at the end of evaluating a program, some redexes might remain unevaluated. For debugging, this means that when values from the program are printed, there is a good chance that some of them will be only partially computed, even if printing is delayed until after the program is finished its normal execution. This requires some way of showing the unevaluated parts that the user can understand. This issue does not arise for strict languages, because they usually evaluate to normal form terms, thus upon termination of the program there are no redexes left.

2.4 Monads and I/O

Our biggest mistake: Using the scary term “monad” rather than “warm fuzzy thing”.

Wearing the hair shirt: A retrospective on Haskell

[Peyton Jones, 2003]

Haskell’s standard library provides an abstract type ‘`IO t`’ which describes a computation that produces a value of type `t` and *may* cause side-effects. Side-effects are characterised by an in-place modification to the state of the world, where the world is made available to the program via an operating system, or some such environment. Typical side-effects are reads/writes on a stateful device such as a disk drive, or memory buffer. For side-effects to be predictable (and thus useful for a programmer) their relative ordering must be manifest in the source code. The catch is that in a non-strict language the order of evaluation is not easily correlated with the structure of the program. What is needed is a means for introducing determinism in the order that side-effects are performed, without adversely compromising the non-strict semantics of the purely functional part of a program.

The `IO` type on its own does not guarantee the correct semantic properties of I/O in Haskell. The role of the type is to denote an expression that possibly performs a side-effect, however it says nothing about when the effect will be performed. Four additional ingredients are required:

1. Primitive effects (such as reading from and writing to a file).
2. An effect sequencer.
3. A method for injecting pure (non side-effecting) values into the `IO` type.
4. A means for making `IO` computations happen.

Primitive effects are provided by the runtime environment. Sequencing is done by:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

which is commonly pronounced *bind*. The first argument to `(>>=)` is an `IO` computation producing a value of type ‘`a`’, the second argument is a function which consumes the output from the first computation producing a new `IO` computation as its result. Nested applications of `(>>=)` can be used to ensure that sequences of `IO` computations occur in the order that they are syntactically specified. Pure computations are inserted into the sequence by ‘`return :: a -> IO a`’. By convention, each Haskell program must include a top-level identifier called `main` with type ‘`IO t`’. The runtime environment drives the sequence of `IO` computations which are bound to `main`.

The standard library does not provide the programmer with a method to “run” `IO` computations on their own; in short there is no function of type ‘`IO a -> a`’.⁵ The only way to manipulate an `IO` value is with `(>>=)`, whose type requires that a new `IO` value is produced as its result. Thus the type system ensures that there is a well defined order for all side-effects produced by the program.

⁵Actually, Haskell does provide a “back door” to the `IO` type called `unsafePerformIO`, which we describe in Section 2.5.

Perhaps one of the most surprising aspects of Haskell is that the machinery introduced for I/O can be generalised to other kinds of computational features. The generalisation is called a *monad*, which consists of a type constructor ‘**t**’, and versions of (**>>=**) and **return** parameterised over **t**. An abstract interface to monads is provided by a type class:

```
class Monad t where
  return :: a -> t a
  (>>=)  :: t a -> (a -> t b) -> t b
```

Individual monads are simply instances of this class, where the parameter **t** is replaced by some type constructor, such as **IO**.⁶

A simple example is the failure monad, which represents computations that can either fail, or succeed with exactly one result. The two possible outcomes are encoded by the **Maybe** type:

```
data Maybe a = Nothing | Just a
```

Sequencing works as follows:

```
Nothing >>= x = Nothing
Just x   >>= f = f x
```

Failure propagates upwards, whilst the values of successful computations are passed from left to right. Computations which (trivially) succeed are constructed like so:

```
return x = Just x
```

One of the advantages of the monad abstraction is that it allows us to write functions which are parameterised by the type of monad, for instance:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f [] = return []
mapM f (x:xs)
  = f x      >>= \y ->
    mapM f xs >>= \ys ->
    return (y:ys)
```

⁶Formally, to qualify as a monad, the implementations of **>>=** and **return** must satisfy three laws, though we do not dwell on them here since they are of no great consequence for the rest of the thesis.

This is a generalisation of the list-map function, whose semantics depends, in part, on the particular monad which is used.

Haskell provides some syntactic sugar for monads, called *do-notation*, which resembles the sequential statement notation of imperative languages. Using this notation the recursive equation of `mapM` can be written as follows:

```
mapM f (x:xs) = do
  y  <- f x
  ys <- mapM f xs
  return (y:ys)
```

Do-notation is desugared like so (somewhat simplified):

```
do { e } ⇒ e
do { p <- e; stmts } ⇒ e >>= \p -> do { stmts }
```

The use of (`>>=`) in the desugaring means that do-notation works for any type of monad.

Other common uses for monads include: state threading, parsing, exceptions, backtracking, and continuations. Wadler [1993] provides a survey of many interesting examples. An operational semantics for the I/O monad (and various extensions) is given in Peyton Jones [2001].

2.5 Pragmatic features

Haskell includes two primitives which are helpful for pragmatic reasons:

```
seq :: a -> b -> a
unsafePerformIO :: IO a -> a
```

`seq` introduces strict evaluation into the language, and `unsafePerformIO` allows possibly side-effecting expressions to be treated as if they were pure expressions.

The Haskell Report gives a denotational semantics for `seq` as follows:

$$\begin{aligned} \text{seq } \perp b &= \perp \\ \text{seq } a b &= b, \text{ if } a \neq \perp \end{aligned}$$

The main use for `seq` is to force the evaluation of its first argument in situations where delaying that evaluation may have unwanted consequences; typically to avoid space leaks. The lack of a formal operational semantics for Haskell means that the relative order of evaluation of the arguments to `seq` is unspecified. Despite this, it is often assumed that — as the name suggests — the first argument is evaluated before the second argument, and this is the semantics that most compilers provide. `seq` is also used to implement a strict function application operator as follows:

```
($!) :: (a -> b) -> a -> b
f $! x = seq x (f x)
```

We use `seq` and `$!` in the implementation of `buddha`, and we assume the operational semantics described earlier.

`unsafePerformIO` allows an IO computation to be “run” in an arbitrary context. As its name suggests, the function *can be* unsafe to use. For instance, in conjunction with other IO primitives, it can be used to cause a program to crash. Nonetheless, there are legitimate uses for `unsafePerformIO`. For example, Haskell supports a foreign function interface (FFI) [Chakravarty, 2002], which allows Haskell programs to interface with code written in other languages. It is assumed that foreign procedures may perform side-effects, so the FFI requires that foreign calls return their results in the IO type. Some foreign procedures behave like pure functions. Wrapping such calls in `unsafePerformIO` allows them to be treated as pure functions from within Haskell.

Another use for `unsafePerformIO` is to observe the behaviour of programs, for the purpose of implementing program monitors and debuggers. As Reinke [2001] notes, `unsafePerformIO` allows us to attach hooks to the underlying evaluation mechanism. A simple example of this practice is demonstrated below, where `seq` and `unsafePerformIO` are used in conjunction to provide a very primitive tracing facility:

```
trace :: String -> Bool
trace str = seq (unsafePerformIO (putStrLn str)) False
```

`trace` takes a string argument, prints it to the standard output device, and returns `False`. It is intended to be used in conjunction with Haskell’s guarded equation notation. Recall the `mag` function from Figure 2.1. Suppose that we want to print a debugging message each time `mag` is called, which shows the value of its argument, without changing the value that `mag` computes. We can do this by adding an additional equation to the start of the function like so:

```
mag xs | trace ("mag " ++ show xs) = undefined
mag xs = ... -- the original definition of mag

undefined :: a
undefined = undefined
```

In a multi-equation function definition, if all the guards in the first equation fail, execution “falls through” to the following equation (if one exists), and so-on until a successful match is found, or all the equations are exhausted. When `mag` is called, the first equation will be tried. `trace` always returns `False`, which causes the guard in the first equation to fail (so its body is not evaluated). This causes the second equation to be tried, leading to the normal evaluation of `mag`. However, a consequence of the call to `trace` is a side-effect which prints the desired debugging message to the standard output device.

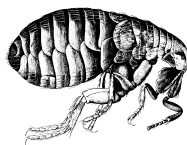
In `buddha`, we use `unsafePerformIO` to attach observation hooks to parts of the program, to build a detailed record of its evaluation history.

2.6 Final remarks

A detailed specification of the static semantics of Haskell is provided in Faxén [2002], whilst Jones [1999] formalises a large part of the type system as a Haskell program. A thorough discussion of type systems, especially the Hindley Milner variety (and extensions), is in Pierce [2002].

Unfortunately the dynamic semantics of Haskell is not fully defined in the Language Report, though the omission is probably intended to make the language flexible with respect to evaluation order. An early draft is given in Hammond and Hall

[1992], but it is somewhat out of date, especially with regards to I/O. However, many facets of candidate semantics can be found in the literature. Launchbury [1993] provides a semantics for lazy evaluation which is very helpful for understanding the dynamic behaviour of lazy languages at a fairly high level of abstraction. Harrison et al. [2002] describe some of the finer points of Haskell's semantics, particularly with reference to pattern matching, and cases where Haskell is strict, using an interpreter for a subset of Haskell, written in Haskell. Various abstract machines are also described in great detail, including the G Machine [Johnsson, 1984] and STG Machine [Peyton Jones, 1992], which show how the high-level notions of lazy evaluation and graph reduction can be mapped onto the low-level aspects of real computers. Finally, Peyton Jones [1986] gives a very thorough treatment of high- and low-level semantics of lazy languages, and Plasmeijer and van Eekelen [1993] discuss graph reduction in detail.



Chapter 3

Declarative Debugging

... programmers who write debugging systems wrestle with the problem of providing a proper vantage point.

Reflection and Semantics in Lisp

[Cantwell Smith, 1984]

3.1 Introduction

DEBUGGING involves a comparison of the actual and intended behaviours of a program, with the aim of constructing an explanation for any disparity between them. For obvious reasons the diagnosis of a bug must be in terms of the source code, and it is preferable to localise the description of a bug to a small section of code, to make it easier to fix. To achieve this end, it is necessary for the debugger to show the behaviour of the program at a suitably fine granularity. In compiled languages a program may pass through several intermediate states in its transformation from source code to machine code. Various bits of information are lost in the transition from one state to the next, such as types, identifier names and program structure. Part of the process of building a debugger is to undo this loss of information. That begs the question: what information should be kept, and furthermore, how should it be presented?

It is a long established principle that declarative languages, such as Prolog and Haskell, emphasise the *what* of programming rather than the *how*. Or to put it another way, declarative thinking focuses on describing logical relationships between elements of a problem rather than a procedure for ordering actions to produce a solution. One benefit of the declarative view is that it allows for a more abstract mode of programming, which can often lead to more concise and “obvious” programs. Another benefit is that the meaning of programs can be described very simply, without recourse to the complexities of control flow and program state.

A problem with this style of programming is that when an execution of a program produces the wrong result for its given arguments it can be very difficult for the programmer to understand why, especially if they are forced to think in terms of its operational behaviour. Declarative debugging was proposed by Shapiro [1983]¹ to overcome this problem by focusing on the declarative semantics of the program, rather than its evaluation order. In other words, a suitable vantage point for debugging logical errors in declarative languages is their declarative semantics. Shapiro’s main contribution was to show that, given a description of the declarative semantics of a program as a computation tree, it is possible to automate much of the labour which is normally involved in debugging.

Shapiro’s work was couched in terms of Prolog and logic programming, but since then it has been transferred to other programming paradigms, such as procedural languages [Fritzson et al., 1992], object oriented languages [Naish, 1997], purely functional languages [Naish and Barbour, 1996, Nilsson, 1998, Sparud, 1999], logic-functional languages [Caballero and Rodríguez-Artalejo, 2002], and type error debugging [Chitil, 2001, Stuckey et al., 2003].

This chapter considers the basic principles of declarative debugging in the context of Haskell.

¹He called it Algorithmic Debugging.

3.1.1 Outline of this chapter

The rest of this chapter proceeds as follows. In Section 3.2 we discuss debugging Haskell in very broad terms, providing some background and motivation for the rest of the chapter. In Section 3.3 we introduce the evaluation dependence tree (EDT), which resembles a dynamic call graph, and we define a debugging algorithm which operates on that tree. In Section 3.4 we illustrate the behaviour of the algorithm in a small debugging example. In Section 3.5 we discuss two different ways of showing higher-order functions, and relate each way to the structure of the EDT. This leads to a more general concept of evaluation dependency than previous definitions in the literature. In Section 3.6 we show that named constant declarations can introduce cyclic paths in the EDT, and consider the implications for the debugging algorithm. In Section 3.7 we discuss some ways which can improve the efficiency of the debugging algorithm.

3.2 Background

A natural way to decompose computations in functional languages is in terms of reductions. We assume a single-step reduction relation, called \rightarrow , which is defined over pairs of terms, using the rules defined in the program equations, and the semantics of Haskell (*i.e.* the rules for variable substitution, pattern matching and so forth). If $t_1 \rightarrow t_2$, then t_1 can be reduced to t_2 , by one reduction step. The normal soundness condition on \rightarrow is assumed, namely:

$$\text{if } t_1 \rightarrow t_2 \text{ then } t_1 = t_2$$

That is, if one term can be reduced to another, then those terms are equal (have the same meaning). Of course, if there is a bug in the program, it may be the case that the equality implied by reduction does not hold in our intended interpretation of the program.

Suppose that t_1 can be reduced to t_2 by application of the program rule p . We

can associate an individual reduction step with the program rule from which it was derived, using an annotation like so:

$$t_1 \xrightarrow[p]{} t_2$$

If t_1 is not equal to t_2 in the intended interpretation of the program, then p is to blame for the error.

Not all redexes involve program equations. Some are what we call *system redexes*, which involve insignificant “internal” evaluations. Examples are *case* and *let* expressions. Conversely, *program redexes*, are those redexes which arise from program equations:

- $f\ e_1 \dots e_n$, where f is the name of a let-bound function of arity n
- f , where f is the name of a let-bound constant (a pattern binding)

It is reasonable to limit our attention to program redexes, since they represent the invocation of programmer defined abstractions. System redexes are, by their very nature, always correct.

To find buggy program rules we could search through all the annotated reduction steps from a program evaluation, and identify those steps which violate our expectations about term equality. Of course, for non-trivial program runs, a search through all the reduction steps in sequential order is unlikely to be feasible because the number of steps will be prohibitively large.

A much better approach is to employ a multi-step reduction relation, so that we can consider the correctness of many single steps at one time. We define a multi-step reduction relation over pairs of terms, called \rightarrow^* , as the reflexive, transitive closure of \rightarrow :

$$\begin{aligned} & t \rightarrow^* t, \text{ for all } t \\ & \text{if } t_1 \rightarrow t_2 \text{ then } t_1 \rightarrow^* t_2 \\ & \text{if } t_1 \rightarrow t_2 \text{ and } t_2 \rightarrow t_3 \text{ then } t_1 \rightarrow^* t_3 \end{aligned}$$

The benefit of multi-step reductions is that, if $t_1 \rightarrow^* t_n$, and t_1 is equal to t_n in the intended interpretation of the program, there is no need for us to consider the correctness of any of the individual reduction steps in between t_1 and t_n (which could be a large number of steps). It might be the case that one or more of those steps was incorrect, however, none of those errors can be said to be to blame for any bugs which are observed for the program run as a whole. If we do find an incorrect multi-step reduction, we only need to consider the correctness of the single steps in between the initial and final terms of that reduction. These too can be partitioned into multi-step reductions, and so on, until we arrive at reductions which require only one step.

Conventionally, computations are regarded as sequential structures. An important idea in declarative debugging is that computations can also be regarded as trees. The use of a multi-step reduction relation leads naturally to a tree structure, which we call an *evaluation dependency tree* (EDT).

In the next section we define the properties of the EDT, and give a recursive error diagnosis algorithm which automates the search for bugs.

3.3 The EDT and wrong answer diagnosis

3.3.1 Properties of the EDT

An EDT has the following properties:

1. Nodes in the EDT have:
 - A multi-step reduction.
 - A reference to a program equation.
 - Zero or more children nodes.
2. Reductions in the nodes have the form $\mathcal{L} \rightarrow^* \mathcal{R}$ where \mathcal{L} and \mathcal{R} are different terms. \mathcal{L} is a program redex, and \mathcal{L} is reduced as one of the steps from \mathcal{L} to \mathcal{R} . The node refers to the program equation whose left-hand-side matches \mathcal{L} .

3. If a node contains a reduction $\mathcal{L} \rightarrow^* \mathcal{R}$, and that reduction does not involve any program redexes, then the node has no children. Otherwise, the node has one or more children. Let $\mathcal{L} \rightarrow \mathcal{R}_0$ be the single-step reduction of \mathcal{L} . The children of the node are any set of sub-trees constructed from the reductions: $\mathcal{L}_1 \rightarrow^* \mathcal{R}_1, \dots, \mathcal{L}_k \rightarrow^* \mathcal{R}_k$, such that the following entailment holds: $\mathcal{L} = \mathcal{R}_0, \mathcal{L}_1 = \mathcal{R}_1, \dots, \mathcal{L}_k = \mathcal{R}_k \vdash_{\mathcal{H}} \mathcal{L} = \mathcal{R}$

The entailment operator, $\vdash_{\mathcal{H}}$, is specific to the “theory” of Haskell computations (hence the \mathcal{H} annotation). It means that the term equalities on the right-hand-side can be deduced from the equalities on the left-hand-side, plus any equalities arising from system redexes. Hence, we avoid the need to mention the system redexes explicitly.

In addition to the above properties, it is useful to require that the EDT represents the complete evaluation of some initial program term. The simplest way to do this is to define a special root node like so: if t_0 is the initial program term, and its final value is t_f , then we can require that the EDT contains a node representing the reduction $t_0 \rightarrow^* t_f$.

The second property of the EDT ensures that there is exactly one outermost redex in \mathcal{L} which is reduced in the reduction. This tends to simplify the task of judging the correctness of reductions, and it also means that each node in the EDT refers to just one program equation. For example, it rules out reductions such as this:

$$\text{f (g 3, h 4)} \rightarrow^* \text{f (5, 6)}$$

If this reduction is incorrect, it could be because ‘ $\text{g 3} \rightarrow^* 5$ ’ is incorrect, or because ‘ $\text{h 4} \rightarrow^* 6$ ’ is incorrect, or because both are incorrect. It is much simpler if the EDT stores each reduction in a separate node. Doing so does not lose any precision in the diagnosis.

The above definition of the EDT allows for many different concrete trees for an initial program term. The reasons are twofold. First, we do not specify the reduction relation. This is necessary because Haskell does not have a formal operational

semantics. Indeed, Haskell specifically allows different evaluation strategies. Different evaluation strategies can lead to different reduction steps, which in turn can lead to different nodes in the EDT. Second, we allow the children of a node to be organised in different ways. This is because we view the EDT as a proof tree. A sub-tree containing a reduction $\mathcal{L} \rightarrow^* \mathcal{R}$ is a proof that $\mathcal{L} = \mathcal{R}$, according to the program equations and the semantics of the language. By using entailment to relate a parent node with its children, we do not prescribe an order in which the steps in the proof must be made. This means we are free to restructure the EDT, so long as the entailment is preserved. A small issue with entailment is that it allows a node to have children which are not actually related to its reduction. We could add an additional requirement that the entailment is somehow “minimal”. We believe that this detail is not important because the addition of spurious children does not change the soundness of the bug diagnosis (providing those children nodes satisfy all the requirements of normal EDT nodes), and an implementation of the EDT (such as ours) will avoid the addition of such nodes in practice.

For aesthetic reasons, we employ the symbol \Rightarrow to indicate the reduction relation when we show nodes in the EDT. We use `=>` for the same symbol in typewriter font.

3.3.2 Identification of buggy equations

Given an EDT we can say which nodes correspond to buggy equations in the program. We adopt the terminology of Naish [1997]. We assume that there is an intended interpretation of the program which defines the *expected* meaning of terms which appear in the EDT. A node containing $\mathcal{L} \Rightarrow \mathcal{R}$ is *erroneous* if (and only if) \mathcal{L} and \mathcal{R} do not have the same meaning in the intended interpretation. Conversely, a node is *correct* if (and only if) \mathcal{L} and \mathcal{R} *do* have the same meaning in the intended interpretation. There is a third case, which arises when \mathcal{L} or \mathcal{R} (or both) do not have any meaning in the intended interpretation, but for simplicity we do not cover that case here; we will return to this issue in Chapter 4, when the process of judging nodes for correctness is considered in more detail.

A node is *buggy* if it is erroneous but has no erroneous children. A buggy node refers to an incorrect equation in the program. We can show that this is true by considering two cases. The first case is when a node has no children. \mathcal{L} can be reduced to \mathcal{R} by the application of the program equation whose head matches \mathcal{L} (the equation referred to by the node) and the evaluation of zero or more system redexes. If the node is erroneous, it must be the equation referred to by the node which is to blame for the error. The second case is when a node has one or more children $\mathcal{L}_1 \Rightarrow \mathcal{R}_1, \dots, \mathcal{L}_k \Rightarrow \mathcal{R}_k$, and all the children are correct. Let $\mathcal{L} \rightarrow \mathcal{R}_0$ be the reduction of \mathcal{L} by one step. From the definition of the EDT we have:

$$\mathcal{L} = \mathcal{R}_0, \mathcal{L}_1 = \mathcal{R}_1, \dots, \mathcal{L}_k = \mathcal{R}_k \vdash_{\mathcal{H}} \mathcal{L} = \mathcal{R}$$

If the right-hand-side of the entailment is erroneous then it must be the case that one or more of the premises on the left-hand-side is to blame. If all the children are correct then the only equation to blame for the mistake is the one whose head matches \mathcal{L} , therefore that is a buggy equation.

Declarative debugging is a search through the EDT for buggy nodes. Shortly we will present a simple algorithm which automates this search.

3.3.3 Example EDTs

Recall the small program introduced in Section 2.3.1:

```
double x = x + x
start = double (3 * 2)
```

Figure 3.1 depicts three of the many possible EDTs for the evaluation of **start**. Each tree represents a proof that **start** can be reduced to 12. The difference between them is the order in which the sub-proofs are structured. The top tree is labeled “small-step” because each node contains only a single small-step reduction. The bottom tree is labeled “big-step” because each node contains reductions which show their results in their final state of evaluation. The middle tree is labeled “multi-step”, depicting the possibility of reduction steps which are somewhere in between

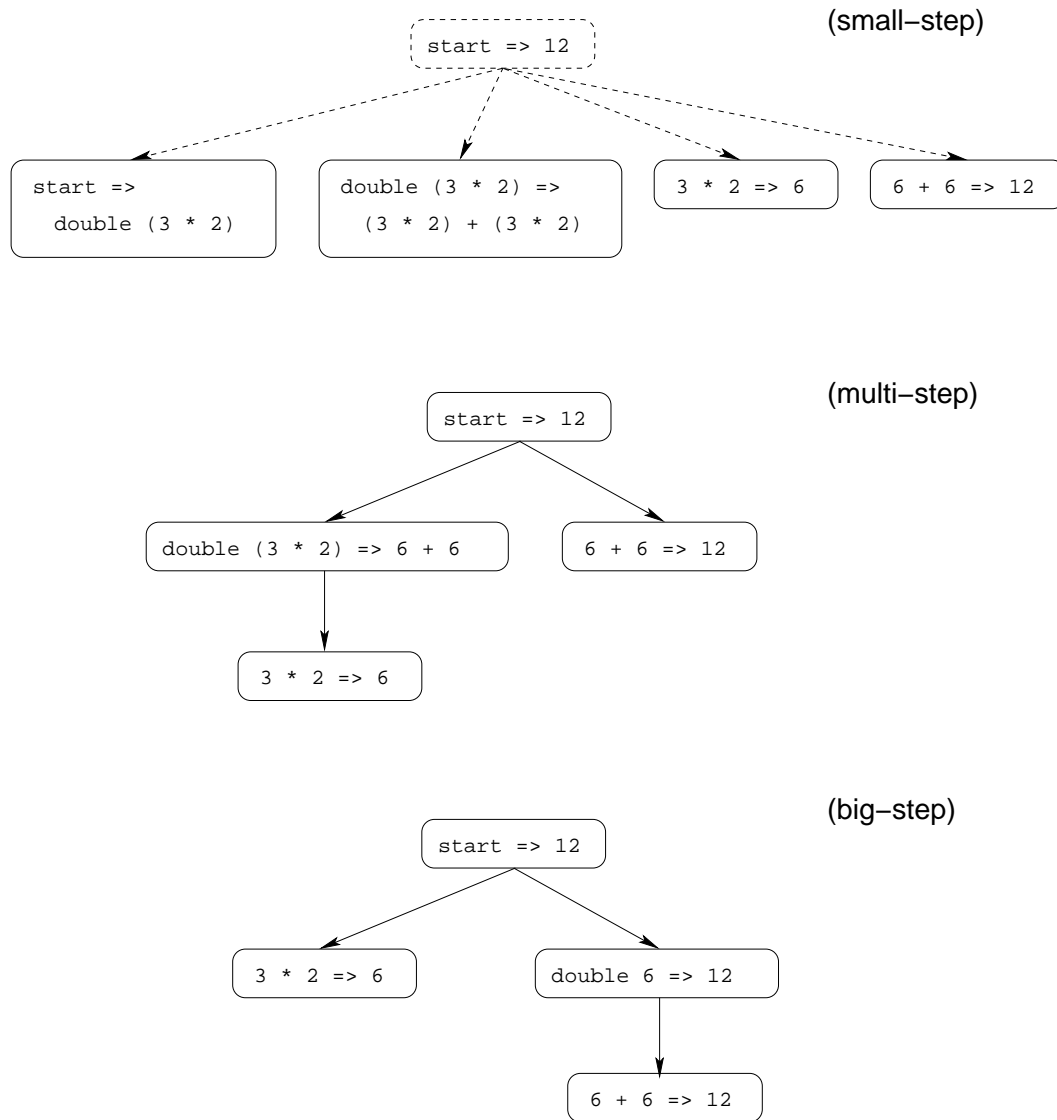


Figure 3.1: Three example EDTs for the same computation, exhibiting various reduction step sizes in their nodes.

small and big steps. The small-step tree is shown with an extra “virtual” node at its root – hence the use of dashed lines – which collects all the individual reduction steps under a common parent. Without this contrivance the small-step EDT would not be a tree at all, but simply a collection of nodes.

Whilst the structures of the trees are different, each tree is suitable for declarative debugging.

3.3.4 Big-step EDTs

In *buddha*, as in all previous declarative debuggers for functional languages, we construct a big-step EDT. An EDT is a *big-step EDT* if, in each node, the sub-terms in \mathcal{L} , and the whole of the result \mathcal{R} , are shown in their final state of evaluation. A term is in its final state of evaluation just prior to the point where it is no longer needed by the program (*i.e.* just before it would be garbage collected).

Big-step EDTs have a couple of advantages over the other structural variants:

1. It is (usually) easier to understand a reduction if the components are final values, rather than arbitrary intermediate expressions.
2. A big-step tree suggests an “order of evaluation” which reflects the static dependencies between function calls in the source code.

However, different step sizes may have their own benefits in special circumstances, and we plan to investigate more flexible tree structures in future work.

Each node $\mathcal{L} \Rightarrow \mathcal{R}$ in a big-step EDT has the following two properties:

1. Any redexes which appear in \mathcal{L} , except \mathcal{L} itself, were never evaluated in the execution of the program.
2. \mathcal{R} is not a redex, and any redexes which appear in \mathcal{R} were never evaluated in the execution of the program.

In other words, the sub-terms of \mathcal{L} and the whole of \mathcal{R} are shown in their final state of evaluation. Let $\mathcal{L} \rightarrow \mathcal{R}_0$ be the single step reduction of \mathcal{L} . The two

Let $f\ x_1 \dots x_m$ be a redex for some function f (of arity m) with arguments x_i , $1 \leq i \leq m$. Suppose

$$f\ x_1 \dots x_m \Rightarrow \dots (g\ y_1 \dots y_n) \dots$$

where $g\ y_1 \dots y_n$ is an instance of an application occurring in f 's body and furthermore a redex for the function g (of arity n) with arguments y_i , $1 \leq i \leq n$. Should the g redex ever become reduced, then the reduction of the f redex is *direct evaluation dependent* on the reduction of the g redex.

Figure 3.2: Nilsson's definition of direct evaluation dependency.

properties above imply that the children nodes of $\mathcal{L} \Rightarrow \mathcal{R}$ correspond to *all* and *only* those redexes which were created by $\mathcal{L} \rightarrow \mathcal{R}_0$, and which were eventually reduced in the execution of the program. Based on this property, Nilsson [1998, Chapter 4] defines the relationship between nodes in a big-step EDT according to the rule for *direct evaluation dependence* in Figure 3.2. What this means is that we can determine the dependencies between nodes in the EDT based on a syntactic property of the program. Indeed, this is one of the reasons why a big-step EDT is desirable, because it reflects the dependencies of symbols in the source code. A central part of the program transformation employed by **buddha** is to encode this notion of direct evaluation dependency into the program.

We argue for the soundness of this rule in Section 5.8.2.

3.3.5 An interface to the EDT

For the purposes of this chapter an abstract interface to the EDT is sufficient, as illustrated in Figure 3.3. We consider a concrete implementation in Chapter 5.

The interface provides two operations on EDT nodes:

1. **reduction**, to extract the reduction from a node.
2. **children**, to get the children of a node.

```

module EDT where

-- name and source coordinates of an identifier
type Identifier    = (FileName, IdentStr, Line, Column)
type FileName      = String
type IdentStr      = String
type Line          = Int
type Column        = Int

-- note the explicit quantifier in this type
data Value = forall a . V a

data Reduction
  = Reduction
    { name      :: Identifier
    , args      :: [Value]
    , result    :: Value
    }

data EDT = ...    -- abstract

reduction :: EDT -> Reduction
reduction node = ...

children :: EDT -> [EDT]
children node = ...

```

Figure 3.3: An abstract interface to the EDT in Haskell.

Each reduction contains three components:

1. The name of the function that was applied.
2. The arguments of the application.
3. The result.

All values stored in the EDT are injected into a universal type called `Value`, by way of the constructor function:

```
V :: a -> Value
```

Note that the type of *V*'s argument is not exposed in its result. This allows the EDT to store values of arbitrary types. For this we need to use an explicit quantifier in the definition of *V*. This kind of quantification is not allowed in Haskell 98, however it is a widely supported extension. In Chapter 6 we show how to turn *Values* into printable strings.

3.3.6 Wrong answer diagnosis

```
data Judgement = Correct | Erroneous
data Diagnosis = NoBugs | Buggy Reduction

wrongAnswer :: Diagnosis -> [EDT] -> IO Diagnosis
wrongAnswer diagnosis [] = return diagnosis
wrongAnswer diagnosis (node:siblings) = do
  let thisReduction = reduction node
  judgement <- askOracle thisReduction
  case judgement of
    Correct -> wrongAnswer diagnosis siblings
    Erroneous
      -> wrongAnswer (Buggy thisReduction) (children node)

askOracle :: Reduction -> IO Judgement
askOracle reduction = ... -- abstract

-- the top level of the debugger
debug :: IO ()
debug = do
  roots <- get the root(s) of the EDT
  diagnosis <- wrongAnswer NoBugs roots
  case diagnosis of
    NoBugs -> output: no bugs found
    Buggy reduction -> output: this reduction is buggy
```

Figure 3.4: The top-down left-to-right wrong answer diagnosis algorithm.

A simple way to find a buggy node is to search the EDT in a top-down, left-to-right fashion. Figure 3.4 contains Haskell code which implements this kind of search.

A diagnosis is obtained from the `wrongAnswer` procedure, which returns either

NoBugs if no bugs were found, or ‘**Buggy** r ’, where r is the reduction from a buggy node. **wrongAnswer** takes two arguments: the current diagnosis, and a list of sibling nodes. Initially, the current diagnosis is **NoBugs**, and the list of sibling nodes just contains the root node(s) of the EDT. If the list of sibling nodes is empty, the current diagnosis is returned, otherwise the nodes are considered from left to right. If the head of the list is **Erroneous**, it could be a buggy node, so a new bug diagnosis is constructed containing the reduction from the node, and its children are recursively considered. Otherwise, the tail of the list is recursively considered with the current diagnosis. Thus the debugger moves down the EDT when it finds an erroneous node, and left-to-right when it finds a correct node.

Judgement is treated as an abstract process which is performed by an oracle (via **askOracle**). The oracle knows the intended meaning of each function defined in the program. In practice the oracle simply passes reductions on to the user of the debugger for judgement, but much of its behaviour can be automated. For instance, the oracle in **buddha** remembers previous answers given by the user to avoid repeated questions.

If all the paths in the EDT are finite, and the root node is erroneous, the algorithm will eventually find a buggy node. If the root node of the EDT corresponds to the first function called in the program (*i.e.* **main**), then the diagnosis returns what Naish [1997] calls a *topmost buggy node*. Such a node is buggy and all its ancestors up until **main** are erroneous. This is distinct from an arbitrary buggy node which might be the descendent of a correct node. Only topmost buggy nodes can be considered as potential causes of externally observable bugs.

There are a couple important points to note about this algorithm:

- There might be multiple buggy nodes in the EDT. The algorithm only finds one at a time (in particular, it finds the leftmost one).
- The diagnosis of a bug in some child node is not necessarily to blame for the erroneous result in its parent for two reasons. First, there might be multiple erroneous children nodes. Second, the parent might be buggy independently

of its children. Therefore new buggy nodes might appear after the causes of other buggy nodes have been fixed.

3.3.7 Constructing the EDT

There are two basic approaches to constructing the EDT described in the literature:

1. Modification of the runtime environment to produce it as a side-effect of reduction.
2. Source-to-source program transformation which extends a program to compute the EDT as part of its final value.

In the first approach, a special runtime environment is employed which builds EDT nodes and links them together as part of the process of graph reduction. A modified compiler is needed to produce object code which takes advantage of the additional features of the runtime environment.

In the second approach, the debuggee is transformed into a new Haskell program which computes both the value of the debuggee, plus an EDT describing that computation. The new program is compiled to produce a debugging executable.

We employ the second approach because it simplifies the implementation of the debugger and enhances its portability. A more detailed discussion of the pros and cons of each approach appears in Section 8.3.

3.4 An example debugging session

In this section we consider an example debugging session using `buddha`. Figure 3.5 contains a small program for converting numbers written in base ten notation to other bases. It reads two numbers from the user: the number to convert, and the desired base of the output. It prints out the number written in the new base. The intended algorithm goes as follows:

1. Prompt the user to enter a number and a base. Read each as a string, and convert them to integers using the library function `read` (which assumes its argument is in base ten).
2. Compute a list of “prefixes” of the number in the desired base. For example, if the number is 1976, and the base is 10, the prefixes are ‘[1976, 197, 19, 1]’. This is the job of `prefixes`.
3. For each number in the above list, obtain the last digit in the desired base. For example if the list is ‘[1976, 197, 19, 1]’, the output should be ‘[6, 7, 9, 1]’. This is the job of `lastDigits`.
4. Reverse the above list to give the digits in the desired order.
5. Convert each (numerical) digit into a character. Following tradition, numbers above 9 are mapped to a letter in the alphabet. For example, 10 becomes ‘a’, 11 becomes ‘b’ and so on. This is the job of `toDigit`.

The program refers to several functions which are imported from the Haskell Prelude. Definitions of those functions, and examples of their use can be found in [Pope, 2001]. The implementation of the Prelude and standard libraries are *trusted* to be correct, and assumed to be well understood by Haskell programmers. *Buddha* can take advantage of this and avoid mentioning these functions, which tends to make debugging sessions shorter and simpler. More information about trusted functions can be found in Section 7.3.

Higher-order functions require special consideration. To highlight this aspect of debugging we have included a version of `map`, called `mymap`, in the definition of the program. Calls to the Prelude defined `map` would be trusted by *buddha*, and thus would be invisible in the debugging example. There is no way to turn off the trusting of Prelude functions in the current version of *buddha*, but this limitation can be overcome by re-defining the function within the program, where it will no longer be automatically trusted.

```

1      module Main where

      main = do putStrLn "Enter a number"
                num <- getLine
5         putStrLn "Enter base"
                base <- getLine
                putStrLn (convert (read base) (read num))

      convert :: Int -> Int -> String
10     convert base number
        = mymap toDigit
          (reverse
           (lastDigits base
            (prefixes base number)))
15
      toDigit :: Int -> Char
      toDigit i = (['0'..'9'] ++ ['a' .. 'z']) !! i

      prefixes :: Int -> Int -> [Int]
20     prefixes base n
        | n <= 0 = []
        | otherwise = n : prefixes base (n `div` base)

      lastDigits :: Int -> [Int] -> [Int]
25     lastDigits base xs = mymap (\x -> mod base x) xs

      mymap :: (a -> b) -> [a] -> [b]
      mymap f [] = []
      mymap f (x:xs) = f x : mymap f xs

```

Figure 3.5: A (buggy) program for converting numbers in base 10 notation to other bases.

There are numerous bugs in the program, which exhibit several faults:

1. The conversion for positive numbers and positive bases is wrong. For example, converting 1976 to base 10, produces 0aaa as output. The expected output is, of course, 1976.
2. It terminates, but produces no output when the number to convert is less than or equal to zero.

3. It fails with an exception if a conversion requires a digit larger than 'z'.
4. It appears to enter an infinite loop when the base is 1, producing no output.
5. It fails with a “divide by zero” exception if the base is zero.
6. It produces a numeric answer for negative bases, when it probably should report an error message.
7. It fails with an exception if the input strings cannot be parsed as base ten numbers.

The rest of this example shows how to use **buddha** to find the first of the above bugs.

Debugging with **buddha** takes place in five steps:

1. Program transformation. To make a debugging executable, the source code of the original program (the *debuggee*) is transformed into a new Haskell program. The transformed code is compiled and linked with a declarative debugging library, resulting in a program called **debug**.
2. Program execution. The **debug** program is executed.
3. Declarative debugging. Once the debuggee has terminated, the user can begin declarative debugging, which takes the form of a dialogue between the debugger and the user, following the algorithm in Figure 3.4.
4. Diagnosis. The debugging dialogue continues until either the user terminates the session or the debugger makes a diagnosis.
5. Retry. For a given set of input values there might be more than one cause of an erroneous program execution. To find all the causes the user must repeat the above steps until no more bugs are found (after modifying the source code to repair each defect).

Each step is outlined below. Boxed text simulates user interaction on an operating system terminal. Italicised text indicates user-typed input, the rest is output. The operating system prompt is indicated like so: `▷`.

Program transformation

Suppose that the program resides in a file called `Main.hs`. The first thing to do is transform the program source code (and compile it *etcetera*). A program called `buddha-trans` is provided for this task:

```
▷ buddha-trans -t extends Main.hs
buddha-trans 1.2.1: initialising
buddha-trans 1.2.1: transforming: Main.hs
buddha-trans 1.2.1: compiling
Chasing modules from: Main.hs
Compiling Main_B          ( ./Main_B.hs, ./Main_B.o )
Compiling Main            ( Main.hs, Main.o )
Linking ...
buddha-trans 1.2.1: done
```

Appendix A contains the Haskell code which results from this step.

Buddha allows higher-order values to be printed in two different styles: the intensional style, which is based on the term representation of the function; and the extensional style, which shows the function as a finite map (we call this a *minimal function graph*, borrowing the terminology from Jones and Mycroft [1986]). For the sake of demonstration we use the extensional style in this example; we tell `buddha-trans` to use this style by default with the ‘`-t extends`’ command line switch. For comparison, Appendix B illustrates an alternative debugging session for the same program using the intensional style instead. We discuss the ramifications of each style in more detail in Section 3.5.

For each module `X` in the program, `buddha-trans` transforms the code in that module and stores the result in a file called `X_B.hs`. To avoid cluttering the working directory with the new files, `buddha-trans` does all of its work in a sub-directory called `Buddha`, which is created during its initialisation phase. Compilation is done

by the Glasgow Haskell Compiler (GHC)².

Program execution

The compilation of the transformed program results in an executable file called `debug`. When this program is executed it first behaves like the original program, then it starts a debugging session.

```
▷ ./Buddha/debug
```

In this example, the program prompts for two input numbers, and prints the result:

```
Enter a number
1976
Enter base
10
0aaa
```

The debugger begins at the point where the original program would have terminated.

```
Welcome to buddha, version 1.2.1
A declarative debugger for Haskell
Copyright (C) 2004 - 2006 Bernie Pope
http://www.cs.mu.oz.au/~bjpop/buddha

Type h for help, q to quit
```

Declarative debugging

After the welcome message, we see a reduction for `main` and a prompt:

```
[0] <Main.hs:3:1> main
    result = { 0 -> (8,Right ()) }

buddha:
```

²www.haskell.org/ghc. At present this is the only compiler that `buddha` supports.

The first line indicates that this reduction involves `main` whose definition begins on column 1 of line 3 in the file `Main.hs`. Each reduction is uniquely numbered, and the number appears within square brackets at the start of the first line. Since `main` is the first thing evaluated by the program its reduction is always numbered 0. The second line shows the result of `main`, which is an I/O value. We explain the meaning of this representation in Section 7.2. As we shall see, it is safe to ignore it for the purposes of this example.

Before we proceed with debugging we might like to see what the top of the EDT looks like. We can ask for an outline of the top few nodes in the EDT using the `draw` command:

```
buddha: draw edt
```

The output is saved into a file called `buddha.dot` by default, using the Dot graph language [Gansner et al., 2002]. The graph can be viewed with a tool such as `dotty`:³

```
▷ dotty buddha.dot
```

Figure 3.6 illustrates the output of the `draw` command for this particular debugging session. Note that nodes in the diagram only show reduction numbers and identifier names; the arguments and results in reductions are not shown. Figure 3.7 provides a more detailed illustration of the top part of the EDT, where reductions are shown within the nodes. Dashed lines in the diagram indicate parts of the tree that have been truncated for brevity.

As already mentioned, `buddha` does not display nodes for trusted functions. This explains why there are no nodes in the diagram for Prelude functions.

It is worth pointing out that the numbering of reductions in Figure 3.6 corresponds to the order in which function applications are reduced. Tracing through the nodes in order reveals an intricate interleaving of reductions, which occurs because of lazy evaluation. For example, consider the subtrees under `lastDigits` and

³www.research.att.com/sw/tools/graphviz

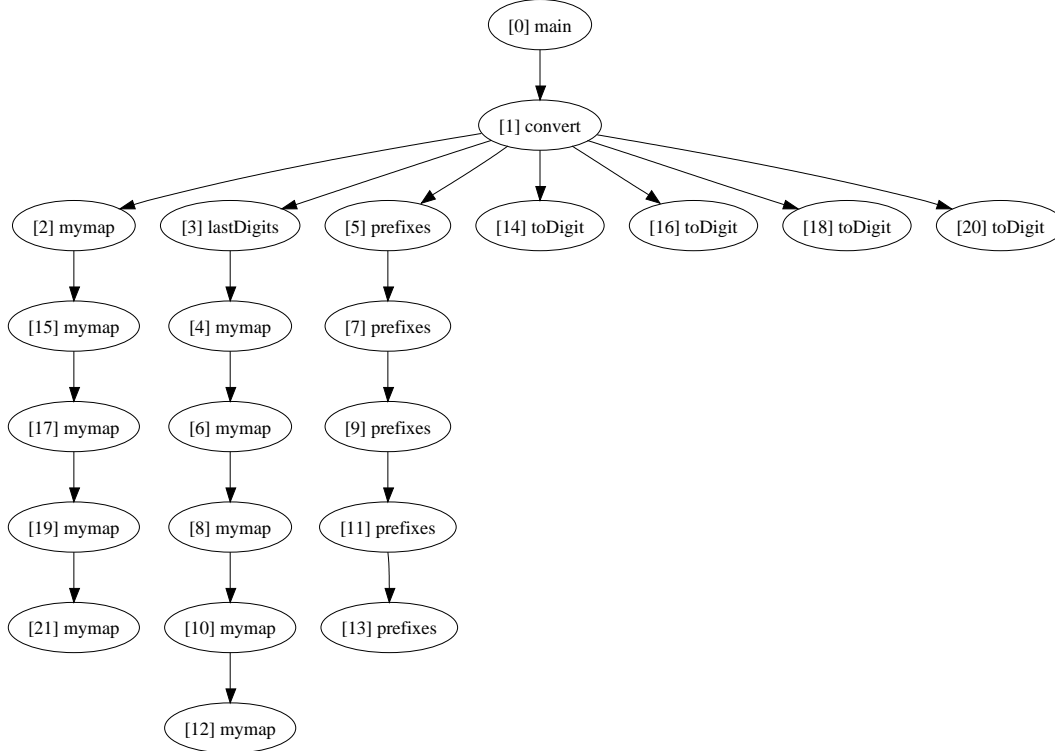


Figure 3.6: An example EDT diagram produced by the ‘draw edt’ command.

prefixes. Note how the reductions of **mymap** in the first subtree and interspersed with the reductions of **prefixes**. This kind of “demand driven” evaluation can be very difficult to follow, which motivates structuring the EDT according to logical dependencies rather than reduction order.

Now, back to debugging. We are faced with a reduction for **main**. At this point we can choose between three basic courses of action:⁴

1. Judge the correctness of the reduction.
2. Explore the EDT.
3. Quit the debugger.

The first option is ruled out because we do not, as yet, know how to interpret

⁴Actually, there are many more things the user can do, such as ask for help, print diagrams of values, change settings in the debugger and so on. However, the three actions mentioned here are the most fundamental of them all.

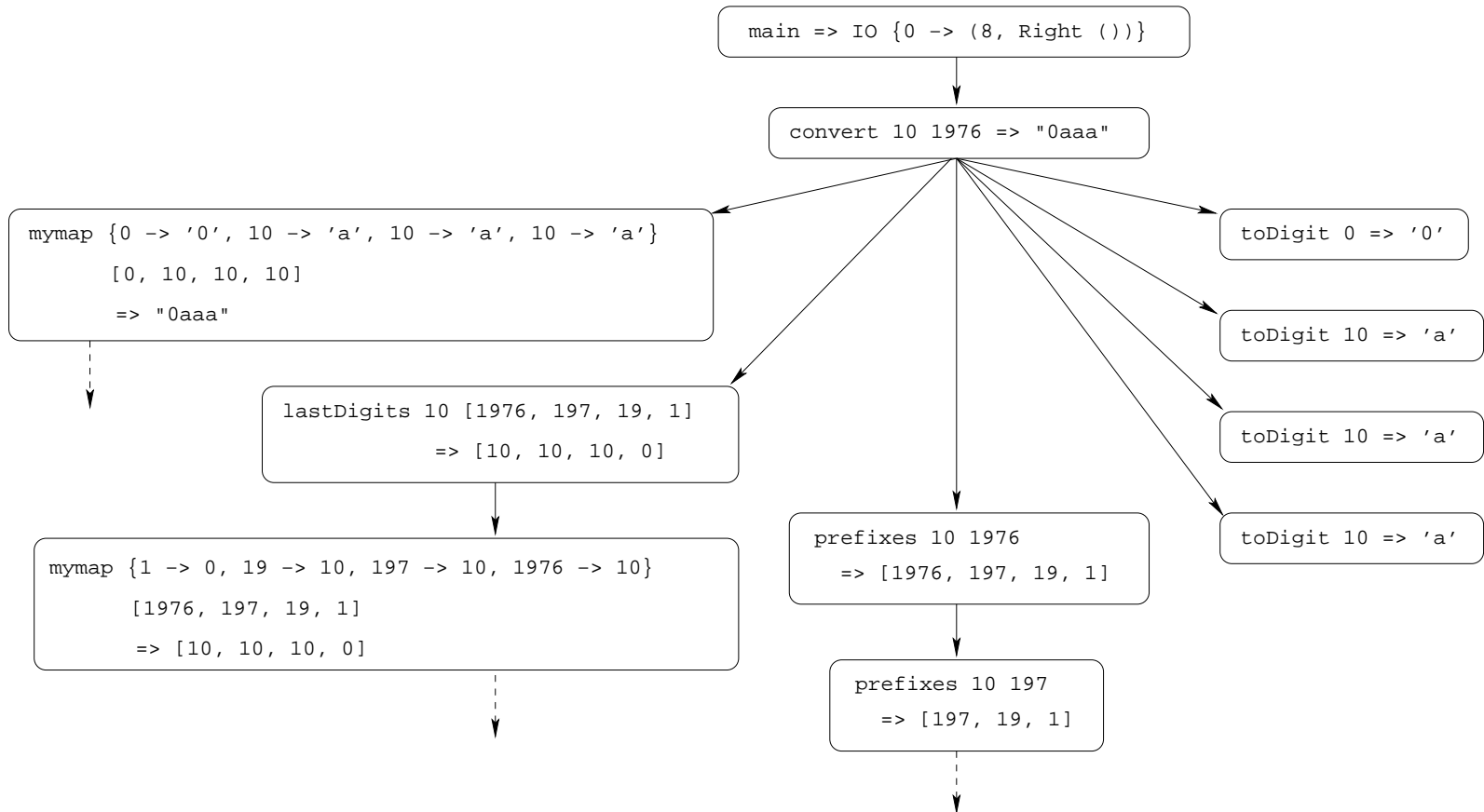


Figure 3.7: An EDT for the program in Figure 3.5.

I/O values. There is no point quitting, so we must explore the EDT. We can do this by jumping from `main` to some other node. For instance we can jump from `main` to one of its children. The children of a node can be viewed with the `kids` command:

```
buddha: kids
```

Buddha responds as follows:

```
Children of node 0:  
  
[1] <Main.hs:10:1> convert  
    arg 1  = 10  
    arg 2  = 1976  
    result = ['0','a','a','a']
```

There is only one child, which accords with the diagram of the EDT in Figure 3.6. We can jump to this node like so:

```
buddha: jump 1
```

Clearly the reduction for `convert` is wrong, because the output is expected to be "1976". We can declare this to the debugger judging the reduction to be *erroneous*:

```
buddha: erroneous
```

When we make a judgement the debugger automatically chooses which node to visit next. Since `convert` is erroneous the debugger moves to the first of its seven children, which is displayed on the terminal:

```
[2] <Main.hs:28:1> mymap  
    arg 1  = { 10 -> 'a', 10 -> 'a', 10 -> 'a', 0 -> '0' }  
    arg 2  = [0,10,10,10]  
    result = ['0','a','a','a']
```

The first argument of `mymap` is a function, which is displayed as a minimal function graph:

```
{ app1, app2 ... appn }
```

Each ‘ app_i ’ represents an individual element of the function graph, of the form: ‘ $\text{argument} \rightarrow \text{result}$ ’. It is minimal in the sense that it only shows those instances of the function that were needed in the execution of the program.

The intended meaning of `mymap` is the same as the Prelude function `map`, and it should be clear, at least intuitively, that this reduction is correct. The user can declare this using the `correct` judgement:

```
buddha: correct
```

Since the first child of `convert` is correct, the debugger automatically moves on to the next child, which is an application of `lastDigits`:

```
[3] <Main.hs:25:1> lastDigits
    arg 1  = 10
    arg 2  = [1976,197,19,1]
    result = [10,10,10,0]
```

The expected output is [6, 7, 9, 1], so the application is erroneous:

```
buddha: erroneous
```

Since `convert` is erroneous the debugger moves to its first (and only) child:

```
[4] <Main.hs:28:1> mymap
    arg 1  = { 1976 -> 10, 197 -> 10, 19 -> 10, 1 -> 0 }
    arg 2  = [1976,197,19,1]
    result = [10,10,10,0]
```

Careful inspection of this application reveals that it is correct:

```
buddha: correct
```

Diagnosis

```
Found a bug:
[3] <Main.hs:25:1> lastDigits
    arg 1  = 10
    arg 2  = [1976,197,19,1]
    result = [10,10,10,0]
```

The debugger concludes that this application of `lastDigits` is buggy, because it is erroneous and its only child is correct.

Here is the definition of `lastDigits`:

```
lastDigits base xs = mymap (\x -> mod base x) xs
```

The error is due to an incorrect use of `mod`. The intention is to obtain the last digit of the variable `x` in some base. However, the arguments to `mod` are in the wrong order (an easy mistake to make); it should be `'\x -> mod x base'`.

Retry

Having repaired the code to fix the defect, we may be tempted to dust our hands, congratulate ourselves, thank `buddha` and move on to something else. But our celebrations may be premature. `Buddha` only finds one buggy node at a time, however there may be more lurking in the same tree. A diligent bug finder will re-run the program on the same inputs that caused the previous bug, to see whether it has been resolved, or whether there is more debugging to be done. Of course it is prudent to test programs on a large number and wide variety of inputs as well. If we make any modifications to the program we will have to run the program transformation again, otherwise we can skip that step.

```
start = map (plus 1) [1,2]
map f [] = []
map f (x:xs) = f x : map f xs
plus x y = x - y
```

Figure 3.8: A small buggy program with higher-order functions.

3.5 Higher-order functions

Consider the buggy program in Figure 3.8. Obviously `plus` is incorrectly defined. In a conventional declarative debugger the first reduction of `map` would be presented as follows:

```
map (plus 1) [1,2] => [0,-1]
```

Note that the partial application of `plus` is a function, and it is printed as a Haskell term. We call this the *intensional* representation of the function. It is also possible to print the function using an *extensional* representation, like so:

```
map { 1 -> 0, 2 -> -1 } [1,2] => [0,-1]
```

The debugging example from Section 3.4 used this style for printing functions. It is worth noting that we could render the extensional function using a Haskell term, for instance, the argument to `map` could be printed as:

```
\ x -> case x of { 1 -> 0; 2 -> -1; y -> plus 1 y }
```

The last part, ‘`y -> plus 1 y`’, is redundant, because it represents all the instances of the function which were not needed in the execution of the program. In Chapter 4 we show that such unneeded parts can be elided. Thus, the set notation is more succinct.

The way that a function is printed affects how we determine its meaning. In the first case ‘`plus 1`’ is understood as the increment function because we (must) read function names as if they carry their intended meaning. So the first reduction above is judged to be erroneous. In the second case ‘`{ 1 -> 0, 2 -> -1 }`’ is just an anonymous (partial) function which we read at face value. So the second reduction above is judged to be correct.

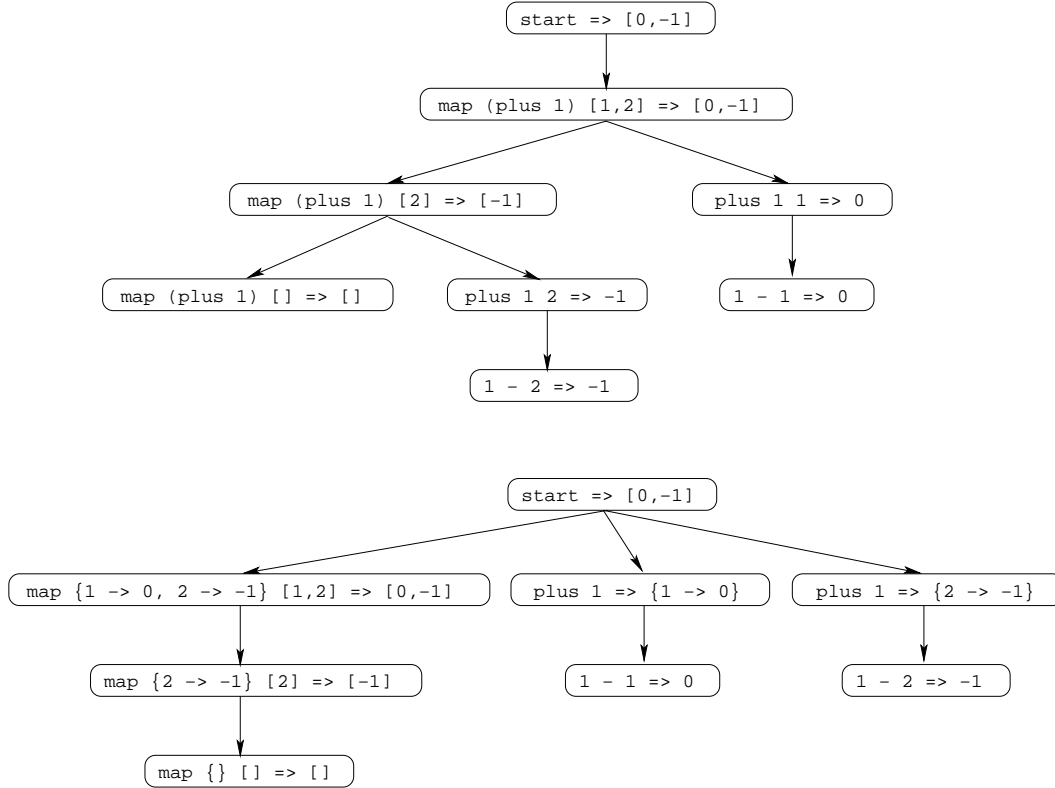


Figure 3.9: Two EDTs for the same computation, illustrating the different ways that functional values can be displayed.

It follows then that the way functional values are printed affects the shape of the EDT, otherwise we should get different bug diagnoses for the example program.

Figure 3.9 shows the two different EDTs for the example program resulting from the different ways functional values can be displayed. The top tree uses the intensional style, and the bottom tree uses the extensional style. Both are suitable for debugging.

The intensional representation follows the conventional view that manifest functions (*i.e.* partial applications and lambda abstractions) are WHNF values. They do not undergo reduction. So, just like constants, we do not need to record nodes for them in the EDT. Conversely, under the extensional representation, manifest functions are treated as if they are redexes. For instance, we pretend that ‘plus 1’ can be “reduced” to ‘{ 1 -> 0, 2 -> -1 }’. In this light, the final representa-

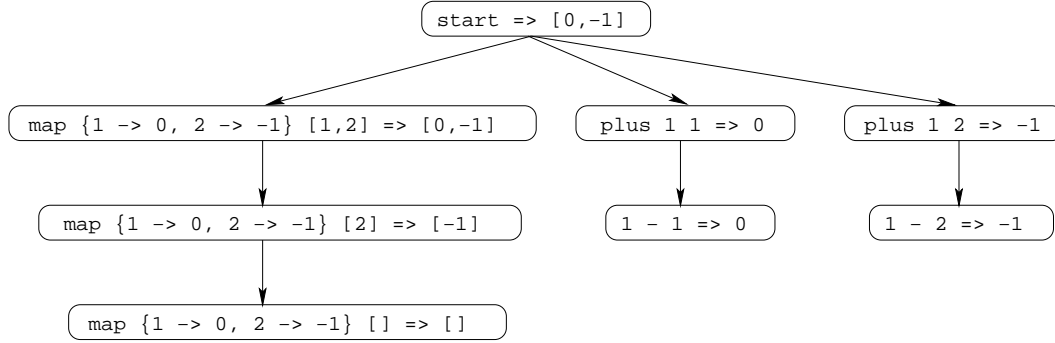


Figure 3.10: An EDT with functions printed in extensional style.

tion of a function is not a term, but a set. We can piece together the “reduction” of this term by collecting all of its application instances from a sub-tree in the EDT. For example, we can regard the reduction ‘`plus 1 1 => 0`’ to be equivalent to ‘`plus 1 => { 1 -> 0 }`’. Indeed, the extensional EDT is just a “bigger-step” variant of the intensional EDT.

It must be pointed out that `buddha` produces a slightly different EDT to the bottom one in Figure 3.9 when the extensional style is used. A more accurate depiction of `buddha`’s tree is given in Figure 3.10. There are two main differences between the trees. First, in `buddha`’s tree, the nodes for `plus` contain reductions which have the form ‘`plus X Y => Z`’. In the earlier tree, those same reductions have the form ‘`plus X => { Y -> Z }`’. This is only a minor presentational issue, and it is straightforward to convert between them. Second, in `buddha`’s tree, the first argument in each node for `map` is always displayed as the set ‘`{ 1 -> 0, 2 -> -1 }`’. In the earlier tree, the set only contains those applications of ‘`plus 1`’ which are found in the sub-tree underneath the particular node for `map`. For example, in the earlier tree we have the reduction:

`map { 2 -> -1 } [2] => [-1]`

which, in `buddha`’s tree, is printed as:

`map { 1 -> 0, 2 -> -1 } [2] => [-1]`

Buddha's representation contains more information than necessary. This is because we modify the representation of functional values so that they record the arguments and results of their own applications in a private data structure. The one representation of a function can be shared at many different application sites, and the private data structure will bear witness to each of those application instances. When the data structure is printed, it might contain application instances which are not strictly relevant to a particular sub-tree of the EDT. However, this does not affect the outcome of debugging. We discuss this issue in more detail in Chapter 4.

While the extensional view might seem strange at first, the set representation of the function is analogous to an ordinary lazy data structure. For instance, we could have written the program like this:

```
start = map plus_1 [1,2]
map f [] = []
map f (x:xs) = apply f x : map f xs
plus_1 = [(a, 1 - a) | a <- [1..]]
apply ((a,b):rest) x
  | a == x = b
  | otherwise = apply rest x
```

That is, we represent the increment function as a list of pairs, and we replace function application with `apply`, which just performs a list lookup. Ignoring `apply` (which could be considered trusted), the EDT for the code above is essentially the same as the one in Figure 3.10.

An interesting consequence of the extensional style is that the links between nodes in the EDT resemble the structure of the code more closely than the intensional style. This is because the extensional style treats partial applications and lambda abstractions as redexes. Therefore the rule for direct evaluation dependency is very simple. A function application determines its parent based on where the function name appears in the source code — even if the name appears as part of a partial application. For instance, `plus` is a child `start` in the above example simply because `plus` appears literally in the body of `start`'s definition. Conversely, `plus` is *not* a child of `map`, because `plus` does not appear in the body of `map`'s definition.

Compare this to the intensional style which can produce rather subtle dependencies because it relates saturated function applications, and functions can become saturated in contexts which are far removed from the place where they are first mentioned. Curiously, the extensional style produces a big-step EDT, but the evaluation steps are even bigger in the case of higher-order functions. This is because the final state of evaluation for functions is further reduced than the corresponding term representation. Therefore, Nilsson’s rule for evaluation dependency still applies, but we must adopt an unorthodox definition of redex.

It is not clear that one particular style of printing functions is always superior to the other. Sometimes the function undergoes many applications, and it can be quite daunting to see them collected together all at once. At other times the term representation of a function can grow to be quite large and difficult to understand, even though that function is applied a small number of times. **Buddha** allows both styles to be used, so that the user can choose which is most appropriate for their particular circumstances. This feature is discussed in more detail in Section 5.6.

3.6 Pattern bindings

Haskell allows named constants to be defined using the same equational notation as functions, for example:

```
pi = 3.142
```

These are called *pattern bindings*. Pattern bindings which do not refer to any lambda-bound variables in their body are sometimes called *constant applicative forms* (CAFs).⁵

Pattern bindings with the CAF property are usually compiled in such a way that their representation is shared by all references to the value, which means that their body is evaluated at most once. Nested pattern bindings which are not CAFs can

⁵Technically a CAF is a term which is not a lambda abstraction and which does not contain free lambda-bound variables.

also be shared, but only within their local scope. Sharing can have a big impact on the performance of a program, as noted in Section 2.3.2.

Sharing is also important for declarative debugging because pattern bindings can be recursive which can lead to cyclic paths in the EDT. This can be a problem for declarative debugging, because the wrong answer diagnosis algorithm from Figure 3.4 can enter an infinite loop if there is cyclic path in the EDT and each node in the path is erroneous. For example, consider this program:

```
ones = 1 : unos
unos = 2 : ones
```

Suppose that both `ones` and `unos` are supposed to equal the infinite list of ones. If pattern bindings are shared we will get one node in the EDT for each value, with a cyclic dependency between them. Suppose the debugger visits the node for `ones` first. We get a reduction like so:

```
ones => 1 : 2 : 1 : 2 : 1 ...
```

The right-hand-side must be truncated at some point because the list is infinite. Nonetheless, it is easy to see that it is erroneous. Therefore we move on to `unos`, which has this reduction:

```
unos => 2 : 1 : 2 : 1 : 2 ...
```

This is also erroneous, which takes us back to `ones` again, and so on forever. Starting with `unos` first also results in an infinite loop.

The bug is in `unos`, but the debugger cannot decide which of the two equations is buggy because of the cycle. In `buddha` we break the cycle at an arbitrary point by deleting one of the edges in the cyclic path. This means that all paths in the EDT have finite length, which in turn means that the top-down wrong answer diagnosis algorithm is guaranteed to terminate. Unfortunately it also means that we can get the wrong diagnosis in some cases.⁶ For example, suppose that we delete the edge

⁶It seems that Nilsson's debugger Freya also has this problem [Nilsson, 1998, Section 6.1]. Regarding this issue he says: "The user has to take the diagnosis of the debugger with a pinch of salt when debugging mutually recursive CAFs, but we do not think this is a large problem in practice."

from **ones** to **unos**. If the debugger ever visits **ones** it will be diagnosed as buggy because it is erroneous but it has no erroneous children.

A better solution would be for the debugger to report all the nodes in an erroneous cycle as *potential* causes of the bug in the program, and let the user decide which ones are true bugs. One way to achieve this is to change the definition of the EDT so that individual nodes can refer to more than one equation. Rather than have one node per evaluated pattern binding, we could treat a set of mutually recursive pattern bindings as a single unit, and generate only one node in the EDT for the whole group. This would eliminate the cyclic dependency from the EDT, but it would require the user to judge more than one reduction at a time.

3.7 Final remarks

The top-down left-to-right algorithm is to be admired for its simplicity, however it is not always the best method for identifying buggy nodes.

Sometimes it is preferable to start the diagnosis somewhere deeper in the EDT than the root node. **Buddha** provides commands which allow the user to explore the EDT in addition to debugging. Even though this feature is useful, we must find our node of interest by navigating to it from **main**, which can be frustrating when that node is very deep in the EDT. In Section 9.2.3 we consider a different interface design for **buddha** which will allow the user to start debugging from an arbitrary expression.

An important factor in the effectiveness of the debugger is how many reductions have to be judged by the user in order for a diagnosis to be made, and also the relative difficulty of reductions that must be considered. Shapiro [1983] calls this the *query complexity* of the debugging algorithm. The worst case behaviour of the top-down left-to-right algorithm is equal to the number of nodes in the EDT.

A number of more advanced search strategies have been proposed in the literature to reduce the query complexity of the diagnosis algorithm. Shapiro [1983] proposes a *divide and query* approach which is motivated by the classic *divide and conquer*

algorithm pattern. Each subtree is assigned a weight, which is some measure of the complexity of debugging the tree. The standard metric is to count the number of nodes in a subtree. The debugger chooses an initial node which divides the EDT as closely as possible into two equally weighted parts. If the chosen node is correct, the entire subtree rooted at that node is pruned from the EDT. If the node is erroneous the subtree rooted at that node is kept and the rest of the EDT is pruned. New weights are calculated for all the subtrees that remain after pruning, and the process repeats until a buggy node is found. The main benefit of the algorithm is that each judgement from the oracle causes the effective search space to be divided by two in terms of weight. If the original EDT has weight n , a buggy node will be found with only $O(\log n)$ nodes considered by the oracle.

The implementation of a practical divide-and-query algorithm in the context of the Mercury logic programming language is discussed by MacLarty, Somogyi, and Brown [2005]. To save space the Mercury debugger does not always keep the entire EDT in memory. Instead, only a partial tree is kept and subtrees are pruned away after a certain depth. Pruned subtrees can be regenerated on demand by re-executing the computation represented by the reduction in their root node. The fact that some parts of the tree may be missing makes the computation of weights more difficult. Therefore the debugger is forced to make an approximation of the weights of some subtrees. The authors also discuss another search strategy called *sub-term dependency tracking* which allows users to mark the sub-parts of values in a reduction that cause the output to be different than what was expected. The debugger will then focus on reductions which produce those sub-parts as their result. This can give a very big improvement in the query complexity of the debugger if the wrong sub-part is only a small fraction of the overall value in which it appears, as is often the case, because the debugger is able to skip over the many reductions that produced the otherwise correct parts of the same value. The idea of sub-term dependency tracking is based on the earlier work of Pereira [1986] in the context of Prolog, under the moniker of *rational debugging*.

One big concern is how well declarative debugging scales with respect to the complexity and size of the program being debugged. The main limiting factor is the space required by the EDT on top of what is needed to execute the debuggee. Long running computations undergo many reduction steps, and each reduction step gives rise to a node in the EDT. On a modern machine, **buddha** can allocate at least 80,000 nodes per second.⁷ Furthermore, the reductions in the EDT retain references to argument and result values which precludes their garbage collection. This means that the space consumption of a debugged program is proportional to the duration of its execution, even if the debuggee needed only constant space. Without some way of reducing the size of the EDT, declarative debugging is limited to only very short computations. Chapter 7 considers various techniques for keeping the memory requirements of the EDT within feasible limits.

The class of bugs detectable by declarative debugging is limited by what information is expressed in the EDT. This rules out any aspects of the program that are not visible from the values it computes, limiting the kind of bugs found to those dealing with the logical consequences of the program. Debugging of performance related issues, such as space leaks or excessive execution times, is a very important part of program development and maintenance, however they are not diagnosed by the declarative debugger, and thus not dealt with in this thesis. Conventional wisdom suggests that profiling tools are the most suitable debugging aids for these kinds of problems, and such facilities are available for the main Haskell implementations, see for example Runciman and Wakeling [1993], Runciman and Rojemo [1996], Samson and Peyton Jones [1997].



⁷This is only a very rough figure. The number of nodes allocated per second varies between applications, because not every reduction takes the same amount of time. Other factors, like garbage collection, can also greatly influence the rate of node allocation.

Chapter 4

Judgement

Computers are good at following instructions, but not at reading your mind.

The T_EXbook

[Knuth, 1984]

4.1 Introduction



JUDGING reductions for correctness can be a difficult task, especially when the values contained in them are large. In non-strict languages the task is even harder because not all values are necessarily reduced to normal forms at the end of program execution. Therefore, the oracle must decide on the correctness of reductions which contain unevaluated function applications (thunks). This chapter formalises judgement in the presence of partial values.

Under non-strict semantics it is possible that a function application is made but never reduced to a normal form. Consider this code:

```
const x y = x
loop n = loop (n+1)
start = const True (loop 0)
```

The evaluation of `start` will eventually produce the value `True`.

According to the program, the value of `‘const True (loop 0)’` is independent of the value of `‘loop 0’`. Under lazy evaluation `‘loop 0’` will never be reduced. In a “less lazy” (but still non-strict) implementation of Haskell, such as optimistic evaluation, it might be the case that `‘loop 0’` undergoes some finite number of reductions during the reduction of `start`. In this context, function applications that remain at the end of the program execution *may* have been subject to some reduction. And some of those reductions could have been erroneous. However, terms that do not reach weak head normal forms cannot be causes of externally observable bugs in the program. This is because Haskell’s pattern matching rules can only distinguish between weak head normal forms. Pattern matching is the only way to affect which equation of a function is used in a given reduction step. Thus an expression which is not a weak head normal form cannot influence which reductions are made in the rest of the program.

In Chapter 3 we showed that the big-step EDT prints argument and results of reductions in their final state of evaluation. Therefore we might expect to see a reduction for the application of `const` printed in this way:

```
const True (loop 0) => True
```

If this is correct, then it must be correct independently of the intended meaning of `‘loop 0’`. It should be possible to replace `‘loop 0’` with any other expression and still get the same answer, so the representation of the first argument is irrelevant to the question of the correctness of the reduction. For this reason `buddha` replaces all unevaluated terms with question marks:

```
const True ? => True
```

Other declarative debuggers for non-strict languages also show unevaluated terms with questions marks (or some kind of special symbol), but the issue of judgement in the presence of these terms has not been given much attention in the literature.

4.1.1 Outline of this chapter

The rest of the chapter proceeds as follows. In Section 4.2 we introduce the notation and terminology used in the rest of the chapter. In Section 4.3 we show how partial values in reductions can be related to the intended interpretation by the use of quantifiers. In Section 4.4 we introduce the notion of inadmissibility to allow for partial functions in the intended interpretation. In Section 4.5 we consider higher-order functions, with specific emphasis on the extensional representation. In Section 4.6 we conclude with some final remarks.

4.2 Preliminaries

Ignoring partial values for the moment, the basic process of judgement works as follows. The oracle is presented with a reduction of the form: $\mathcal{L} \Rightarrow \mathcal{R}$, where \mathcal{L} and \mathcal{R} are closed Haskell terms. If the intended meanings of both sides are equal then the reduction is correct, otherwise it is erroneous.¹

The intended interpretation is elaborated by a *semantic function* which maps closed terms to values in the semantic domain.

$$\mathcal{V} : \text{Closed Term} \rightarrow \text{Value}$$

The semantic domain contains the values that we *think* the program computes, so it is necessarily conceptual. Given \mathcal{V} , a reduction is judged correct if and only if:

$$\mathcal{V}(\mathcal{L}) = \mathcal{V}(\mathcal{R})$$

We assume that objects in the semantic domain can always be compared for equality.

A term is considered closed (in this context) if it does not contain any free lambda-bound variables. However, let-bound variables are always free in the term. This does not cause any trouble for deducing the meaning of the enclosing term

¹In this chapter we will introduce three new judgement types called *inadmissible*, *don't know* and *defer*.

because all let-bound variables are assumed to have an intended meaning which is known to the oracle.

4.3 Partial values

We call function applications which remain unevaluated at the end of program execution *residual thunks*. Residual thunks have no causal connection with bugs which are externally observable, therefore it is possible to debug the program without knowing their value.

Consider the following implementation of natural numbers:

```
data Nat = Z | S Nat
plus :: Nat -> Nat -> Nat
plus x y = ...
```

`Z` represents zero, `S` is a function that maps any natural to its successor, and `plus` is supposed to implement addition.

Suppose that `plus` is buggy, and that the bug leads to this reduction:

```
plus (S ?) Z => S (S ?)
```

Residual thunks have a slightly different connotation depending on which side of a reduction they appear. Those in \mathcal{L} indicate values that were not needed in the determination of \mathcal{R} , whereas those in \mathcal{R} indicate values that were not needed to produce the final result of the program. It is possible to consider the correctness of a reduction by substituting terms for each residual thunk and comparing the result with the intended interpretation. No matter how you instantiate \mathcal{L} it should always be possible to find an instantiation for \mathcal{R} such that they produce the same value according to the semantic function. Under this reasoning the reduction for `plus` is erroneous because there is a counter example. It is intended that ‘`plus (S Z) Z`’ should return one, but all instances of ‘`S (S ?)`’ have value two or more.

In the context of reductions with residual thunks, *correct* really means that a computation is a safe approximation of the intended semantics. In other words, the

result is valid in all the places where it was computed, given how much information was known about the argument values.

The definition of correctness can be extended to support partial reductions by the use of quantifiers. Each residual thunk is regarded as a distinct variable which ranges over the set of closed Haskell terms. Those in \mathcal{L} are universally quantified and those in \mathcal{R} are existentially quantified.² The notation $\mathcal{L}[\alpha_1 \dots \alpha_m]$ represents any left-hand-side term with variables $\alpha_1 \dots \alpha_m$, and similarly $\mathcal{R}[\beta_1 \dots \beta_n]$ for the right-hand-side, with $m, n \geq 0$. A reduction is correct if and only if the following statement is true:

$$\forall \alpha_1 \dots \alpha_m \exists \beta_1 \dots \beta_n : \mathcal{V}(\mathcal{L}[\alpha_1 \dots \alpha_m]) = \mathcal{V}(\mathcal{R}[\beta_1 \dots \beta_n])$$

where $\alpha_1 \dots \alpha_m \beta_1 \dots \beta_n \in \text{Closed Terms}$

That is to say: for all instances of \mathcal{L} there is some instance of \mathcal{R} such that the two instances are equal in the intended interpretation.

It must be emphasised that this statement only describes what it means for a reduction to be correct; it does not provide a feasible algorithm for checking correctness. To actually decide whether a reduction is correct it is necessary for the oracle to find a counter example (and thus show it is erroneous), or *prove* that the statement of correctness always holds.

A problem with the above formulation of correctness is that it allows terms which are not in the intended interpretation, such as ‘`plus (S True) Z`’. Clearly, any term which is the result of a substitution should be closed and well typed. This issue is resolved in the next section.

4.4 Inadmissibility

What should the oracle expect to happen when a function is applied outside of its intended domain?

²Our use of quantified variables is inspired by Naish and Barbour [1995].

Consider `merge`, which takes two sorted lists as inputs and returns a sorted list which contains all and only the elements from the input lists. It is quite useful to allow the intended interpretation of `merge` to be unspecified on unsorted arguments. This allows the programmer to change the actual implementation of a function without having to change its intended meaning. A typical implementation of `merge` will assume that its arguments are sorted, but it will not check this condition for efficiency reasons. In a buggy program this precondition might not be satisfied.

Suppose a call to `merge` occurs in the following context:

```
f xs ys = merge (g xs) ys
```

If '`g xs`' returns an unsorted list then a bug is in at least one of these places:

1. In `g`, or something called by `g`, because '`g xs`' failed to produce a sorted list when it should have.
2. In `f`. Perhaps because `f` should not have called `g` in this instance, or because something is missing from `f`'s body to convert '`g xs`' to a sorted list.
3. In some function which helped produce `xs` (maybe `f` and/or `g` expects that `xs` will have some property which ensures that '`g xs`' is sorted).

It is undesirable for the oracle to judge `merge` to be erroneous on unsorted lists because the fault is elsewhere in the program. In this instance we want the diagnosis to point to the function that caused the unsorted list to be supplied as an argument.

For these situations `buddha` offers a third judgement value called *inadmissible*. In terms of the final diagnosis, *inadmissible* has the same effect as *correct*, but it has a much different meaning. *Correct* means that the intended meaning has been preserved, whereas *inadmissible* means that the application on the left-hand-side should never have happened because some expected property of the intended interpretation has been violated.

Inadmissible can also be used to rule out consideration of unintended terms when quantifying over residual thunks. This includes ill-typed terms which can be

viewed as a special case. As an aside, there is a connection between admissibility and types. If not for Haskell’s type system we would anticipate many more bugs which are caused by inadmissible function applications. Taking this to an extreme, it is possible to think of *inadmissible* applications as type errors, for some sufficiently powerful notion of types. The “sorted inputs” precondition of `merge` can be regarded as a type concept, though one which is not expressible in Haskell.

`Buddha` provides *inadmissible* as a judgement value, but it is also possible to work it into the previous definition of correctness. The idea is to introduce a predicate *admissible* which takes a term and returns true if the term is defined in the intended interpretation, and false otherwise:

$$\begin{aligned} \forall \alpha_1 \dots \alpha_m \exists \beta_1 \dots \beta_n : \\ \text{admissible}(\mathcal{L}[\alpha_1 \dots \alpha_m]) \Rightarrow (\mathcal{V}(\mathcal{L}[\alpha_1 \dots \alpha_m]) = \mathcal{V}(\mathcal{R}[\beta_1 \dots \beta_n])) \\ \text{where } \alpha_1 \dots \alpha_m \beta_1 \dots \beta_n \in \text{Closed Terms} \end{aligned}$$

That is to say: for all admissible instances of the \mathcal{L} there is some instance \mathcal{R} such that the two instances are equal in intended interpretation.

The idea of inadmissible calls comes from Pereira’s work on *rational* debugging for logic programming [Pereira, 1986]. However, its application in `buddha` is more directly influenced by the *three valued declarative debugging scheme* of Naish [2000] (and the corresponding three valued semantics for logic programs [Naish, 2006]), which shows how *inadmissible* can be used to diagnose type and mode errors in Prolog programs. The example of `merge` on unsorted lists is taken from that paper.

4.5 Higher-order functions

Higher-order functions deserve special mention because `buddha` can print functional values in the extensional style. The most immediate effect is that it requires the domain of \mathcal{V} be extended to incorporate this new syntax. A deeper concern is how this representation of functions should be attributed meaning.

The extensional style of printing makes functions resemble lookup tables. In this sense the table is “computed” by evaluating the function at various argument values. Most functions are only computed on a subset of the points in their domain, therefore they exhibit the same kind of partiality as lazy data structures. The analogy with data structures can be quite useful since it allows the oracle to use the same reasoning tactics that were available in the setting of first-order reductions.

Nonetheless, higher-order code can give rise to some surprising reductions because bugs can affect the way functional values are applied, which in turn affects how they are printed and understood. This is illustrated in the next example.

Following on from the theme of functions as data, consider the use of functions to implement lists. The idea is that a list is just a function from index positions to values:

```
type List a = Integer -> Maybe a
```

Looking up an element is achieved by function application. The `Maybe` type indicates that some indices are out of bounds. For this example let us assume that in-bounds indices are always contiguous, and that the first element of the list is always found at index zero. An empty list is just a function where every index is out of bounds:

```
empty :: List a
empty = \x -> Nothing
```

An empty list can be detected by checking if index zero maps to `Nothing`:

```
isEmpty :: List a -> Bool
isEmpty list
  = case list 0 of
      Nothing -> True
      _       -> False
```

Prefixing an element onto the front of a list is achieved by shifting all indices in the list along to the right by one step, and re-assigning index zero to the new element:

```
cons :: a -> List a -> List a
cons val tail
  = \ index -> if index == 0
                then Just val
                else tail (index - 1)
```

Haskell's standard list type can easily be converted to the new `List` type using a right fold:

```
mkList :: [a] -> List a
mkList = foldr cons empty
```

Finally a small program which constructs a `List` from "abcd" and prints its length:

```
main = print (len (mkList "abcd"))

tail :: List a -> List a
tail list = \ index -> list (index - 1)

len :: List a -> Integer
len list
  = case isEmpty list of
      True  -> 0
      False -> 1 + len (tail list)
```

Unfortunately, there is a bug somewhere in the code that causes it to print the answer 1 instead of 4.

Debugging the program with `buddha` leads to this reduction:

```
len { 0 -> Just 'a', -1 -> Nothing } => 1
```

At first it is hard to know what to make of `len`'s argument. It looks like a badly defined list, especially since it has an entry for an index below zero! To make a judgement in this case requires clear thinking. It should not be possible to calculate the length of a list without knowing at which index the last element occurs. However, in this reduction the end of the list has not been determined. What the reduction actually says is: all lists which have the character 'a' as their first element have length one. This is patently false. Some lists do have this property, but not all lists, therefore the reduction is erroneous. The fact that the list gave `Nothing` at index '-1' is a red herring: all well-defined lists have this property.

Upon receiving an *erroneous* judgement from the oracle, `buddha` considers `len`'s two children nodes in turn:

1. `len { 0 -> Nothing } => 0`
2. `tail { 0 -> Just 'a', -1 -> Nothing } => { 0 -> Nothing }`

The first child is correct, because its argument is the empty list. However, the second child is erroneous. It says that all lists which have the character 'a' as their first element have a tail which is the empty list. Again, this is not true for all such lists. Since `tail` has no children it is diagnosed as a buggy node.

Re-considering the implementation of `tail` we can see that this is the right diagnosis. Consider what happens by applying each side to some index value i :

```
(tail list) i = list (i - 1)
```

That means each element in the tail of the list is shifted along one position to the left in the original list. For instance, element at index 0 in the tail is equal to element at index '-1' in the original, when it ought to be at index 1. Thus the correct definition of `tail` is:

```
tail list = \ index -> list (index + 1)
```

Re-running the program in `buddha` with the new definition of `tail` gives a correct reduction for the original call to `len`:

```
len { 0 -> Just 'a', 1 -> Just 'b', 2 -> Just 'c'
      , 3 -> Just 'd', 4 -> Nothing }
=> 4
```

The rules of quantification follow from regular data structures: universal quantification for \mathcal{L} and existential quantification for \mathcal{R} . The same applies for residual thunks which appear as part of an individual entry in the function's representation. For instance, this reduction is correct:

```
len { 0 -> Just ?, 1 -> Just ?, 2 -> Nothing } => 2
```

because the length of the list is always two, irrespective of what the residual thunks are instantiated to. However, this reduction is erroneous:

```
len { 0 -> Just ?, 1 -> ?, 2 -> Nothing } => 2
```

because index 1 could map to `Nothing` making the length of the list one.

4.6 Final remarks

While the use of question marks in reductions allows us to abstract away residual thunks, it does have its drawbacks. Perhaps the biggest problem is that it forces the oracle to reason “in the large”. Universal quantification often leads to questions about infinitely many different instances of a single reduction. Therefore a proof of correctness is required.

In cases where proofs are difficult to construct, a useful heuristic is to search for a counter example. If such an example is found then correctness is ruled out immediately. If a counter example is not found after a reasonable amount of time then the oracle can try to look elsewhere in the EDT for buggy nodes. **Buddha** provides two ways to do this. First, the oracle can issue a *don't know* judgement. In terms of the debugging search, this has the same effect as judging the reduction to be correct. However, any diagnosis reached after this point is tagged with a reminder that the correctness of this reduction was not known. Second, the oracle can issue a *defer* judgement. This tells the debugger to suspend consideration of the current reduction, but perhaps come back to it later if need be.

A simple deferral mechanism is employed in **buddha** where the set of children nodes is treated as a circular queue. Deferring a node simply places it at the end of the queue. One of two scenarios will follow. Either an erroneous node is found amongst the remaining siblings, or all the remaining siblings are either correct or deferred. In the first case the debugging search enters the subtree rooted at that node and the deferred nodes are never re-visited. In the second case the first deferred node will be re-visited when it reaches the front of the queue. Eventually the oracle must make a judgement about the node's correctness, or say that they don't know. A fairly obvious limitation of this style of deferral is that it only operates over the children of a given node, which of course makes it useless when there is only one child. One can imagine more elaborate schemes, such as one that propagates the deferral back up the EDT, so that alternative erroneous paths can be explored. However, the need for such schemes is diminished because **buddha** allows the oracle to jump to

4.6 *Final remarks*

any other node at any time in debugging. All nodes jumped from are remembered on a stack, which when popped, allows debugging to resume as if the jump was never made. In essence the jump mechanism is itself a very flexible mechanism for deferral.



Chapter 5


EDT Construction

... debugging can be a messy problem, if one creates a messy environment in which it has to be solved, but it is not inherently so. The theory of algorithmic debugging suggests that if the programming language is simple enough, then programs in it can be debugged semi-automatically on the basis of several simple principles.

Algorithmic Program Debugging

[Shapiro, 1983]

5.1 Introduction

 CHAPTER 3 established the EDT as the basis for the declarative debugging algorithm. In this chapter we consider the construction of that tree, by way of a source-to-source program transformation. The source code of the input program is transformed into an output program which is suitable for debugging. The output program computes the same answer as the input program, and it also constructs an EDT. The output program is linked with a library containing the debugging code and the total package forms the debugger.

The transformation rules presented in this chapter build a complete EDT — every reduction of a program redex builds a node in the tree. Storing such a tree in

main memory is infeasible for all but the shortest program runs. Chapter 7 considers various modifications to make the system more practical.

5.1.1 Outline of this chapter

The rest of this chapter proceeds as follows. Section 5.2 introduces the central ideas of the transformation, which are illustrated with a small example. Section 5.3 provides an implementation of the EDT using Haskell types. Section 5.4 shows how function bindings are transformed. Section 5.5 covers pattern bindings, which require special treatment to preserve the sharing of their values. Section 5.6 deals with higher-order functions, in particular the transformation of lambda abstractions, and partial applications. Section 5.7 formalises the transformation rules over a core abstract syntax for Haskell. Section 5.8 considers the correctness of the rules. Section 5.9 examines the runtime performance of transformed programs. Section 5.10 concludes with pointers to the the next two chapters, which extend the transformation in various ways.

5.2 The general scheme

Constructing the EDT involves two interrelated tasks:

1. The creation of individual nodes.
2. Linking the nodes together, according to their evaluation dependency.

The EDT is built as a side-effect of program evaluation. Each reduction of a program redex constructs a new EDT node, which is inserted into a list of its siblings by destructive update. Access to the list of siblings is provided by a mutable reference, which is passed to the applied function via a new argument.

5.2.1 An example

The process of constructing the EDT is illustrated in the following example, using code in Figure 5.1 for computing the area of a circle. Reduction steps are shown

```

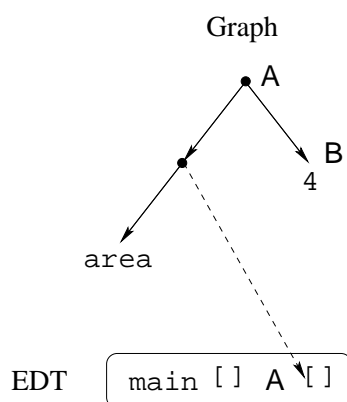
main = area 4
area r = pi * (square r)
square x = x * x
pi = 3.142

```

Figure 5.1: A program for computing the area of a circle.

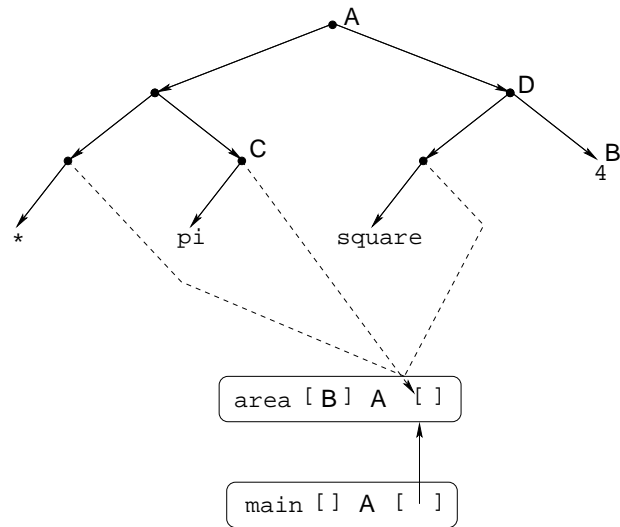
one at a time. Each step is accompanied by a diagram which shows the state of the EDT and the program graph, just after the reduction has taken place.

Step one: ‘main => area 4’.



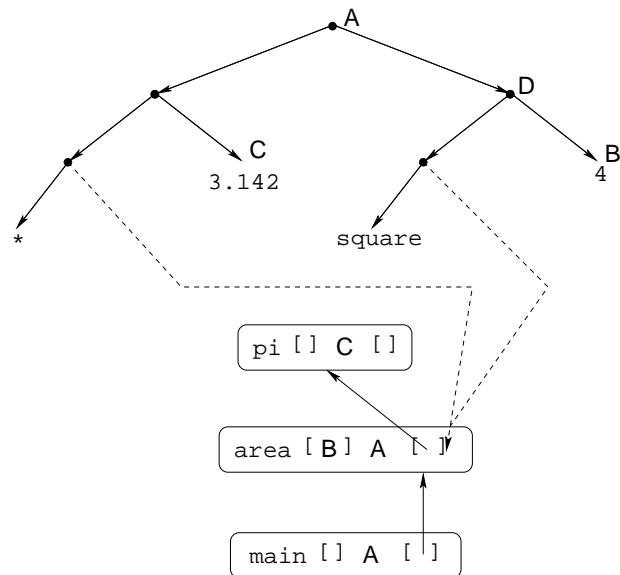
At the top of the diagram is the program graph, and at the bottom is the EDT. Each node has four components: the name of a function, a list of references to its arguments, a reference to its result, and a list of its children nodes. Constants such as `main` have an empty list of arguments. To simplify the diagram, argument and result references are marked by alphabetic labels, such as **A**, though in practice the references are pointers. Function applications in the graph have an additional argument, which occurs in the first position. This extra argument is a mutable reference to the parent of the application (specifically the list of children nodes contained in the parent node), indicated by a dashed edge. These references encode the direct evaluation dependency between nodes. For instance, `main` directly depends on the evaluation of ‘`area 4`’.

Step two: 'area 4 => pi * (square 4)'.



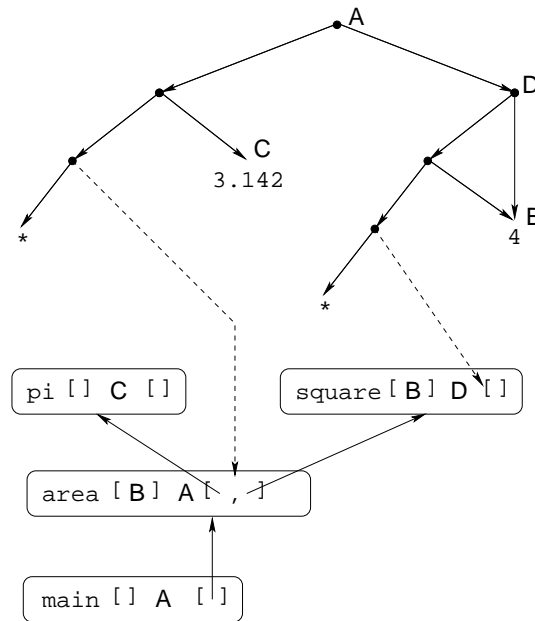
A node for 'area 4' is constructed and inserted as the child of **main**. The body of **area** introduces two new saturated function applications, and a reference to the constant **pi**. Each of these is passed mutable reference to the list of children in the node for 'area 4'. If any of those redexes are reduced, their nodes will be inserted into the correct place in the EDT.

Step three: 'pi => 3.142'.



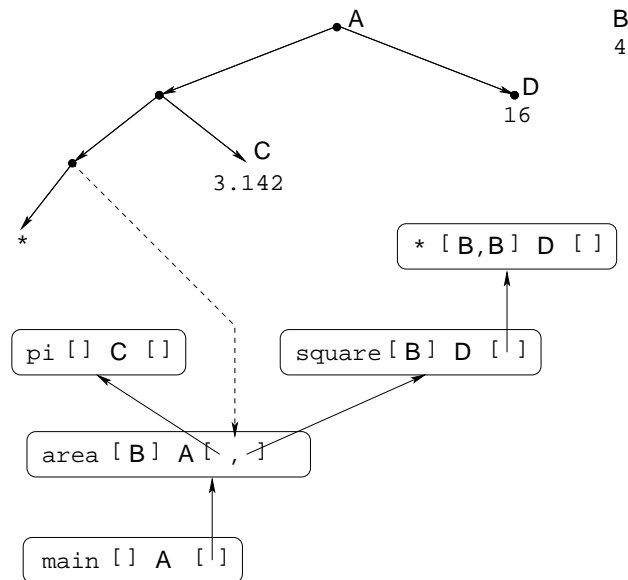
A new node for **pi** is created, and inserted as the first child of ‘**area 4**’. There are no function applications in the body of **pi**, so there are no references from the graph to its list of children.

Step four: ‘**square 4** => **4 * 4**’.



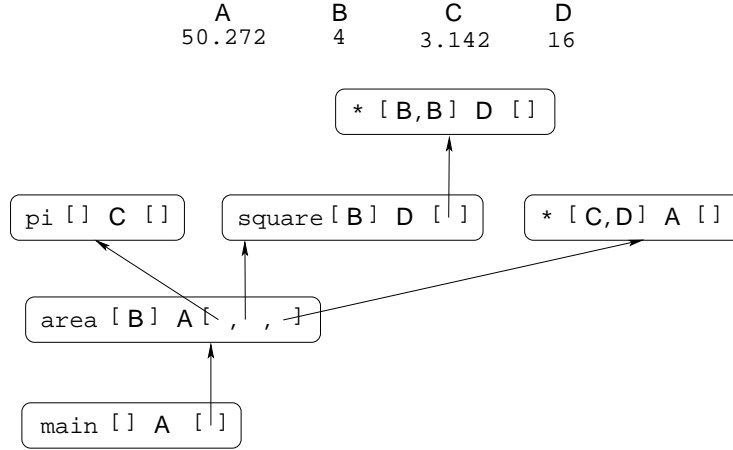
A new node for ‘**square 4**’ is created, and inserted as the second child of ‘**area 4**’.

Step five: ‘**4 * 4** => **16**’.



A new node for ‘4 * 4’ is created, and inserted as the child of ‘square 4’. Note that value B has become disconnected from the main graph. In the normal execution of the program this would allow B to be garbage collected. In the debugging execution B is retained in the heap because it is referred to by nodes in the EDT.

Step six: ‘3.142 * 16 => 50.272’.



A new node for ‘3.142 * 16’ is created, and inserted as the child of ‘area 4’. At this point the evaluation of the original program is complete, and all program values are in their final state of evaluation. The EDT is built and ready for traversal by the diagnosis algorithm, starting with the node for `main`.

5.3 Implementing the EDT

The EDT is implemented using the following type:

```

data EDT
  = EDT
  { nodeName    :: Identifier    -- function name
  , nodeArgs    :: [Value]      -- argument values
  , nodeResult  :: Value        -- result value
  , nodeChildren :: IORef [EDT] -- children nodes
  , nodeID      :: NodeID       -- unique identity
  }

```

Each node contains a function name, a list of argument values, a result value, a list of children nodes, and a unique identity.

Recall from Section 3.3 the use of `Value` as a universal type, which allows each node to refer to a heterogeneous collection of argument and result types.

`IORef` provides a mutable reference type, accessible in the `IO` monad, with the following interface:

```
data IORef a = ... -- abstract
newIORef    :: a -> IO (IORef a)
modifyIORef :: IORef a -> (a -> a) -> IO ()
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO ()
```

`IORefs` are not standard, but are supported by all the main Haskell implementations.

Side effecting operations on `IORefs` are attached to pure computations by the use of ‘`unsafePerformIO :: IO a -> a`’. This is our way of attaching a hook onto the evaluation of redexes.

`NodeID` is an unsigned integer which encodes the unique identifier of each node.

A fresh supply of identifiers is provided by a global counter:

```
counter :: IORef NodeID
counter = unsafePerformIO (newIORef 0)

nextCounter :: IO NodeID
nextCounter = do
  old <- readIORef counter
  writeIORef counter $! (old + 1)
  return old
```

(Strict function application is provided by the `$!` operator, to keep the counter in normal form, and thus avoid a space leak).

The root of the EDT is a global variable containing a mutable list of EDT nodes, which is initially empty:

```
root :: IORef [EDT]
root = unsafePerformIO (newIORef [])
```

5.4 Transforming function bindings

Each function in the program is transformed so that it computes its normal result *and* an EDT node, which it inserts into the EDT as a side-effect. To facilitate this, each let-bound function is given an additional parameter through which it receives a mutable reference to its list of sibling nodes.

The following example demonstrates the transformation of `square` from Figure 5.1. Underlined pieces of code indicate things that have been added or changed.

First, the function needs a new parameter to receive a reference to its siblings:

```
square :: Context -> Double -> Double  
square context x = x * x
```

The extra parameter is called `context`, because it represents the calling context of the function. At present the context is simply a mutable reference to a list of sibling nodes:

```
type Context = IORef [EDT]
```

However, the context can be used to convey more information. In Chapter 7 more detailed context information, such as the depth of the node, is exploited for the purpose of reducing the size of the EDT by computing only parts of it on demand.

There is a design decision to be made in regards to the position of the context parameter. Should it be inserted before or after the existing parameters? In a curried function this decision is significant because the function can be partially applied, thus receiving its arguments in different evaluation contexts. This issue is deferred until Section 5.6 where higher-order functions are dealt with specifically.

Second, a node must be constructed for each call to the function:

```
square context x = call context (x * x)
```

This is achieved by wrapping the body with `call`, which constructs a new EDT node, adds it onto the front of the list of sibling nodes pointed to by `context`, and returns the original value of the body. A full implementation of `call` appears at the end of this section.

Third, the call to `square` must pass its own context information to each of its children:

```
square context x = call context (\i -> (*) i x x)
```

This is done by transforming the original body into a lambda abstraction. The abstraction introduces a new variable, called `i`, which is supplied as the first argument to every function application in the body. A new EDT node is allocated by `call`, and the appropriate context information is derived from it (a pointer to its list of children). The transformed body is then applied to the context information, thus binding it to `i`, forming the link between the parent node and its potential children.

The fourth and last step is to pass the name of the function and a list of its arguments to `call`:

```
square context x
  = call context "square" [V x] (\i -> (*) i x x)
```

Figure 5.2 contains the code for `call` which carries out six steps:

1. Allocate a new mutable list of children nodes.
2. Pass a reference to that list into the transformed body of the function, and save the result.
3. Allocate a new node identifier.
4. Construct a new EDT node.
5. Insert the EDT node into its list of siblings.
6. Return the value saved in step 2.

In step 4 all the components of the EDT node come together. The function's name and argument references are given to `call` as arguments. The result of the function is obtained in step 2. The list of children nodes is created in step 1, and is initially empty. It will be updated if and when any function applications in the body are reduced. The identity of the node is obtained in step 3.

```
call :: Context -> Identifier -> [Value] -> (Context -> a) -> a
call context name args body
  = unsafePerformIO $ do
    children <- newIORef []           -- step 1
    let result = body children       -- step 2
    identity <- nextCounter          -- step 3
    let node =
      EDT                             -- step 4
      { nodeName      = name
      , nodeArgs      = args
      , nodeResult     = V result
      , nodeChildren  = children
      , nodeID        = identity
      }
    updateSiblings context node      -- step 5
    return result                   -- step 6

updateSiblings :: Context -> EDT -> IO ()
updateSiblings context node
  = modifyIORef context (node :)
```

Figure 5.2: Code for constructing an EDT node.

5.5 Transforming pattern bindings

Transforming pattern bindings with the same rules as function bindings has undesirable consequences for sharing. As noted in Section 3.6, multiple references to pattern-bound values are normally shared (relative to their scope) for efficiency reasons.¹ It is desirable for the transformation to preserve this property.

Applying the basic transformation scheme described above to `pi` produces this result:

```
pi context = call context "pi" [] (\i -> 3.142)
```

References to `pi` from distinct contexts will cause it to be re-computed. For something cheap like `pi` this is unlikely to be a problem, but it can have severe consequences for pattern bindings which are expensive to compute, and/or those which

¹The degree of sharing is not defined by the Language Report, therefore it is difficult to make statements which apply equally to all Haskell implementations.

are recursively defined.

In Section 2.3.2 an efficient Fibonacci sequence generator was defined using a recursive pattern binding. Transforming `fibs` as if it were a function binding produces this definition:

```
fibs :: Context -> [Integer]    -- version 1: no sharing
fibs context
  = call context "fibs" []
    ( \i -> 0 : 1 : zipPlus i (fibs i) (tail i (fibs i)))
```

Recursive calls are no longer shared, which means computing the *n*th element of the output list now has exponential time complexity (assuming the compiler does not do common sub-expression elimination on ‘`fibs i`’).

To preserve the sharing of pattern bindings, it is necessary to transform them differently to function bindings.

One possible solution is to follow the design of Freya [Nilsson, 1998, Section 6.1], where each pattern binding produces a root EDT node if and when its body is evaluated to WHNF. This avoids the need for context arguments, thus the declarations remain as constants, and sharing is retained.

This approach is quite easy to implement. We introduce a new function called `constantRoot`, illustrated in Figure 5.3. It plays a similar role to `call`, in that it builds new EDT nodes. However, it inserts its nodes into the root of the EDT, instead of into some parent node. Therefore, `constantRoot` does not need to be given a context argument.

Now, `fibs` can be transformed in a way that preserves its sharing:

```
fibs :: [Integer]    -- version 2: simulating Freya's solution
fibs = constantRoot "fibs"
      ( \i -> 0 : 1 : zipPlus i fibs (tail i fibs) )
```

Note that the recursive calls to `fibs` do not receive context arguments. We assume that the body of `fibs` is evaluated at most once, and that its result is shared amongst all its references. If this is true, then `constantRoot` will also only be called once, and thus `fibs` will get only one node in the EDT.

```
constantRoot :: Identifier -> (Context -> a) -> a
constantRoot name body
  = unsafePerformIO $ do
    children <- newIORef []
    let result = body children
    identity <- nextCounter
    let node =
      EDT
      { nodeName      = name
      , nodeArgs      = []
      , nodeResult     = V result
      , nodeChildren  = children
      , nodeID        = identity
      }
    updateSiblings root node
    return result
```

Figure 5.3: A method for constructing EDT nodes for pattern bindings, which simulates the method used in Freya.

The downside of this approach is that a program with more than one pattern binding will produce a forest of EDTs. Nilsson proposes that the forest of nodes be topologically sorted according to the dependencies between all of the top-level pattern bindings, producing a list of EDTs, which can be debugged one at a time using the standard algorithm.

Debugging over of forest of EDTs is troublesome for three reasons:²

1. Pattern bindings can be defined in different modules. It is difficult to sort them in a system which supports separate compilation. Freya solves this problem within the linker, where global information about a program can be computed. Nilsson says that this is straightforward in Freya because this kind of ordering is already needed to support proper garbage collection of such bindings. However, the task is much more difficult in a debugger based on program transformation, because compilation and linking are delegated to an independent Haskell compiler, and are thus outside of the debugger’s control.

²These problems are also mentioned in [Brehm, 2001, Section 4.5].

2. Pattern bindings can be mutually recursive, therefore there may not be a total order on their dependencies.
3. The user may need to consider many correct EDTs before reaching an erroneous one. However, if there was just one EDT rooted at `main`, the user will not need to consider any nodes for pattern bindings which are descendents of correct nodes.

Hence it is preferable to have a single EDT rooted at the node for `main`. This requires two issues to be solved:

1. Nodes for pattern bindings can have multiple parents.
2. The body of a pattern binding should not be evaluated more often than it would in the normal execution of the program.

In `buddha` we transform pattern bindings so that (in a sufficiently lazy implementation of Haskell) the value of the binding and its corresponding EDT node are computed at most once (within their scope). Subsequent references to the pattern binding share its result and its node is shared by all of its parents in the EDT. We do this by transforming pattern bindings into functions. If the original pattern binding computes some value of type `T`, the transformed version computes a function of type `'Context -> T'`. The first reference to the pattern binding causes its original value to be computed, along with an EDT node. The value and the node are bundled into a function closure, and that closure is then bound to the pattern identifier. The function inserts the EDT node into its context argument, before returning the original value as its result. References to pattern bindings are transformed into function applications, which apply the pattern identifier to the context that is in scope at that point.

We define a new function, called `constant`, which is based on `constantRoot`, but is split into three parts, as shown in Figure 5.4. The first part, called `valueAndNode`, computes the value of the original pattern binding and an EDT node, and returns them in a pair. The second part, called `ref`, takes a value-node pair and a context

```
valueAndNode :: Identifier -> (Context -> a) -> (a, EDT)
valueAndNode name body
  = unsafePerformIO $
    do children <- newIORef []
       let result = body children
       identity <- nextCounter
       let node = EDT
           { nodeName      = name
           , nodeArgs      = []
           , nodeResult    = V result
           , nodeChildren  = children
           , nodeID        = identity
           }
       return (result, node)

ref :: (a, EDT) -> Context -> a
ref (val, node) context
  = unsafePerformIO $
    do updateSiblings context node
       return val

constant :: Identifier -> (Context -> a) -> (Context -> a)
constant name body = ref (valueAndNode name body)
```

Figure 5.4: Code for constructing EDT nodes for pattern bindings.

as arguments and inserts the node into the context before returning the value as its result. The third part, called `constant`, joins the first two parts together. In particular, `constant` constructs a new function by partially applying `ref` to the pair produced by `valueAndNode`. Under lazy evaluation, all instances of that function share the same value-node pair, which means that the pair is only computed once.

Here is `fibs` with the new version of `constant`:

```
fibs :: Context -> [Integer]    -- version 3: with sharing
fibs = constant "fibs"
      ( \i -> 0 : 1 : zipPlus i (fibs i) (tail i (fibs i)))
```

There are two points that deserve special mention. First, note that the body of `fibs` remains a constant expression, but it computes a function as its result. Since it is a constant expression it can be computed once and shared. Yet its value is a

function, so it is possible to pass it different context arguments, and have its node inserted under different parents in the EDT. Second, note that references to `fibs` become function applications, with `i`, a context, as their arguments. In this case it means that the node for `fibs` will be inserted twice into its own list of children, making two cycles in the EDT. Such cycles are deleted by `buddha` as discussed in Section 3.6.

5.6 Transforming higher-order code

Supporting higher-order functions in the transformation raises two issues which are considered in this section:

1. How should lambda abstractions be transformed?
2. What is the right context for a function which is partially applied?

5.6.1 Lambda abstractions

Normally it is assumed that the intended meaning of a function is signified by its name. This assumption breaks down in the case of lambda abstractions because they are anonymous.

Suppose a program contains this declaration:

```
g xs = map (\x -> x + 1) xs
```

If the intended meaning of `g` is to *subtract* one from each element in a list of numbers there is a bug in the code. Is the bug in `g`, or is it in the lambda abstraction? One could argue either way. It boils down to how you describe the intended meaning of the lambda abstraction: it could be as it is written, or it could be as it is supposed to be written. This can be confusing. For instance, suppose this reduction was encountered during debugging:

```
(\x -> x + 1) 2 => 3
```

Taking the meaning of the function as it is written (increment), suggests that the reduction is correct. However, taking the meaning as it is supposed to be written (decrement), suggests that it is erroneous.

We decided that applications of lambda abstractions should not produce reductions of their own, for three reasons:

1. Reductions for lambda abstractions are potentially ambiguous.
2. Lambda functions tend to be used for functions of only secondary significance.
3. It is easy for the user to translate lambda abstractions into let-bound ones if they want a finer precision in the diagnosis.

To simplify the rules of the transformation, lambda abstractions are automatically turned into let-bound functions with unique identifiers. For instance, just prior to the normal transformation, the definition of `gs` is re-written to:

```
g xs = map (let f_uniq x = x + 1 in f_uniq) xs
```

where `f_uniq` is a new unique identifier in the program. Of course it would be confusing for the user to be asked to judge reductions for `f_uniq`, since it is not part of the original program. This problem is avoided by making it a trusted function, which means that when the EDT is traversed, the debugger effectively “steps over” a call to the function and proceeds directly to its children (Section 7.3 covers trusting in more detail).

5.6.2 Partial applications

A partially applied multi-parameter function can receive each of its arguments in disparate evaluation contexts. Which of those many contexts constitutes the proper parent of the eventual call to the function? That depends on how higher-order functions are printed, as discussed in Section 3.5. *Buddha* allows functions to be printed in two different ways: the intensional and extensional styles. Each style has an effect on the direct evaluation dependency which determines the links between

parent nodes and their children. First we consider the extensional style, since that gives the simplest evaluation dependencies, then the intensional style, and finally a generalisation that incorporates both.

Extensional printing of functions

For the extensional style, the parent of a partially applied function is obtained in the context where the function is first mentioned by name. This is easy to implement because this context is a static property of the program.

Consider the transformation of the following example:

```
gs xs = map increment xs
map f list
  = case list of
      [] -> []
      (x:xs) -> f x : map f xs
increment x = x + 1
```

In the body of `gs` two functions are mentioned by name, `map` and `increment`. The eventual reductions of saturated applications of these functions will form the children of the node for `gs`, therefore, each function receives a context argument at its call site:

```
gs context xs
  = call context "gs" [V xs]
    (\i -> map i (increment i) xs)
```

A consequence of the transformation is that `gs` will have at most one child for an application of `map`, because `map` is saturated in the body of `gs`, but `gs` might have any number of children corresponding to applications of `increment`. The actual number is determined by how many times this particular instance of `increment` becomes fully saturated and reduced, which in turn depends on how many elements are demanded in the list returned by `gs`.

Now the transformation of `map`:

```
map context f list
= call context "map" [V f, V list]
  (\i -> case list of
    [] -> []
    (x:xs) -> f x : map i f xs)
```

The application of `f` does not get a context argument. This is because `f` is lambda-bound: whatever function it is bound to dynamically will have its context determined at the point where it is originally mentioned by name.

One of the main advantages of the extensional style is that it leads to a parent-child relationship which closely reflects the relationship between symbols in the source code. In the above example, `gs` is the parent of calls to `map` and `increment`, precisely because `map` and `increment` are mentioned by name in its body. This simplifies the transformation because there is no need to distinguish between partial and saturated applications of let-bound functions. In contrast, the intensional style places nodes in the context where the function is saturated, which can be quite removed from where the function was first mentioned. Also, the point of saturation is, in general, a dynamic property of the program.

One of the main disadvantages of the extensional style is that nodes for partially applied functions form clumps under their parent. Suppose that the list produced by `gs` is evaluated to one thousand elements long. This would give `gs` one thousand children nodes corresponding to calls to `increment` (and only one node for the call to `map`). In the intensional style, the maximum number of children nodes is bounded by the number of saturated function applications that appear in the body. For almost all programs this number is only in the order of tens of nodes.

Intensional printing of functions

For the intensional style, the parent of a call to a function is determined at the point where the function is saturated. It is not always possible to decide statically where function applications are saturated. This makes the transformation more

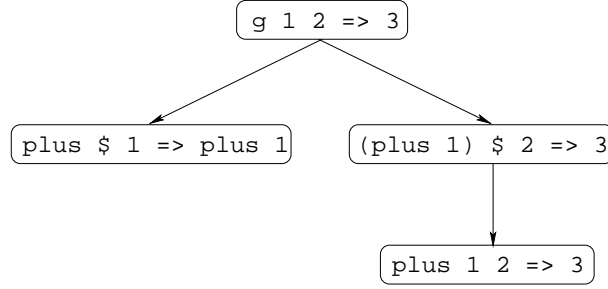


Figure 5.5: An EDT with functions printed in intensional style.

complex, because it must encode sufficient information into the program so that, where necessary, saturated function applications can be recognised dynamically.

Consider this example:

```

($) :: (a -> b) -> a -> b
f $ x = f x
g :: Int -> Int -> Int
g a b = (plus $ a) $ b

```

The first equation defines the infix operator called `$`, which implements function application. The second, somewhat contrived equation defines some function `g`, which implements addition, using `plus` which is assumed to be a function of type ‘`Int -> Int -> Int`’.

What is interesting about this example is that `plus` is applied to its arguments in two steps, and each step occurs inside an independent call to `$`. Figure 5.5 illustrates the EDT that results from evaluating the expression ‘`g 1 2`’, using the intensional printing of functional values. Observe that only the right node for `$` has a child for `plus`, specifically because that represents the context where `plus` is saturated.

The body of `$` contains a function application: ‘`f x`’. When `f` is bound to a function of arity one, the application is saturated, making the node for `$` its parent. Yet, when `f` is bound to a function of arity two or more, the application is partial, which means the node for `$` is not its parent. This creates a problem for the transformation. Sometimes context information should flow from the node created for `$` to the function applied in its body, and sometimes not. Which case applies

depends on how `$` is called, in particular the arity of its first argument. The problem is that functions are (ideally) transformed in a fashion which is independent of their calling contexts. If not, one function definition in the original program might need to be mapped to multiple function definitions in the transformed program, to cater for all the different arities of its arguments. This would complicate the transformation and it could cause an explosion in the code size of the resulting program. Higher-order functions, like `$`, must be transformed in a way which does not depend on the arities of their functional arguments.

As the example above demonstrates, any binary application of a lambda-bound function could potentially be a saturation site. Therefore each binary application is accompanied by a context argument, just in case it is needed. To maintain type correctness, each let-bound function must be transformed to take an additional context argument for each of its usual arguments, but it should only keep the last one, since that represents the point where the function is definitely saturated.

Here are the transformations of `$`, and `g`:

```

($) :: (a -> Context -> b) -> Context -> a -> Context -> b
($) f _ignore x context
    = call context "$" [V f, V x] (\i -> f x i)

g :: Int -> Context -> Int -> Context -> Int
g a _ignore b context
    = call context "g" [V a, V b]
      (\i -> ($) (($) plus i a i) i b i)

```

Note that in `$`, the application of `f` must take a context argument, regardless of the arity of the function that it is bound to.

A typing problem

Inserting context arguments in between ordinary arguments presents a minor typing problem. Intuitively, the transformation rule for function types is something like this:

$$\text{trans}(t1 \rightarrow t2) \Rightarrow \text{trans}(t1) \rightarrow \text{Context} \rightarrow \text{trans}(t2)$$

The problem is that Haskell allows higher-order type application, which means that not all uses of `->` are fully applied.

Here is an example using data types:³

```
data T a = MkT (a Int Int)

h :: T (->) -> Int -> Int
h (MkT g) x = g x
```

The type `T` has a type parameter `a` with kind $\star \rightarrow \star \rightarrow \star$, which means that `a` must be bound to a type constructor of arity two. In the type of `h`, `a` is bound to `->`, but because this is a partial application of the type constructor, there is no way to insert the necessary context argument.

One might hope that type synonyms could be used to provide a sort of lambda abstraction at the type level:

```
type Fun a b = a -> Context -> b
```

and thus transform the type of `h` as follows:

```
h :: T Fun -> Int -> Int
```

Unfortunately, the Haskell Report forbids the partial application of type synonyms [Peyton Jones, 2002, Section 4.2.2].

A workaround is to introduce a new data type to “encode” the function arrow:

```
newtype F a b = MkF (a -> Context -> b)
```

This introduces a new distinct type, which can be partially applied, thus solving the original problem. The downside is that it requires the introduction of the `MkF` data constructor everywhere a function is created. To help with this, a family of encoding functions are introduced, of the form:

```
fun1 :: (a -> Context -> b) -> F a b
fun1 g = MkF g
fun2 :: (a -> Context -> b -> Context -> c) -> F a (F b c)
fun2 g = MkF (\x c -> fun1 (g x c))
...
```

³The same problem can occur wherever higher-kinded type variables are allowed, for instance in type constructor class declarations.

where `funn` encodes functions of arity n . Also it requires a special function application to convert between the “encoded form” and the underlying function:

```
apply :: F a b -> (a -> Context -> b)
apply (MkF f) = f
```

Here is the transformation of `h` without encoded function types, using the intensional style of printing functions:

```
h :: T (->) -> Context -> Int -> Context -> Int
h y@(T g) _c1 x c2
  = call c2 "h" [V y, V x]
    (\i -> g x i)
```

It is ill-typed because the application ‘`g x i`’ in the body requires that `g` have type ‘`Int -> Context -> Int`’. However, because `g` is an argument of the `T` constructor, it must have type ‘`Int -> Int`’.

Here is the transformation of `h` with encoded function types, which fixes the typing problem:

```
h :: F (T F) (F Int Int)
h = fun2 (\y@(T g) _c1 x c2 ->
          call c2 "h" [V y, V x]
            (\i -> apply g x i))
```

Note that all arrow type constructors are transformed into `F`s.

The encoding of function types introduces extra clutter into the transformation, by way of the `funn` encodings, and the `apply` decodings. For the moment, the purpose of `F` is really just to work around a limitation of Haskell’s type system, and a good optimising compiler should be able to remove them completely. Having said that, `F` plays another more substantial role in the transformation, with regards to observing printable representations of functional values. That topic is covered in Chapter 6, specifically in Section 6.5.

Combining the extensional and intensional styles

It is possible to transform the program so that both evaluation dependency rules for higher-order function applications are supported. Each function takes a context

argument at the point where it is mentioned by name, and also a context argument for each of its normal arguments. Only one of the initial context and the final context is retained. The first context is suitable for the extensional style of printing, and the final context is suitable for the intentional style. All intermediate ones are ignored. Currently the user determines which context to use for each function (via a configuration file or command line argument), but it could also be dynamic. It is even possible insert the node into both contexts to allow different views of higher-order code in the same debugging session; the utility of this last option remains an open research question.

5.7 The transformation rules

In this section the formal transformation rules are defined over a typical core abstract syntax for Haskell.

5.7.1 Abstract syntax

Figure 5.6 contains the abstract syntax definition of core Haskell, which is divided into three categories: declarations, expressions and types. Terms appearing in **Typewriter** font are literal fragments of code. Subscripts represent multiple distinct instances of an entity. Ellipses represent variable length sequences.

The main feature missing from the core language is type constructor classes (including class declarations, instance declarations, and type constraints). Of course *buddha* supports these features in practice, but the transformation rules for type classes are fairly mundane — basically type classes just add more declarations to the program — so we have left them out. The overall structure of type classes remains unchanged, and the usual rules of overloading apply. This means that the constraints in type signatures are left unchanged by the transformation rules.

Declarations

$$\begin{array}{ll}
D & \in \quad x :: T \\
& \mid \quad x \, y_1 \, \dots \, y_n = E \qquad (n > 0) \\
& \mid \quad x = E \\
& \mid \quad \text{data } f \, a_1 \, \dots \, a_n = K_1 \mid \dots \mid K_m \qquad (n \geq 0, m > 0) \\
K & \in \quad k \, a_1 \, \dots \, a_n \qquad (n \geq 0)
\end{array}$$

Expressions

$$\begin{array}{ll}
E & \in \quad x \mid k \mid E_1 \, E_2 \\
& \mid \quad \backslash y_1 \, \dots \, y_n \rightarrow E \qquad (n > 0) \\
& \mid \quad \text{let } D_1 \, \dots \, D_n \text{ in } E \qquad (n > 0) \\
& \mid \quad \text{case } E \text{ of } A_1 \, \dots \, A_n \qquad (n > 0) \\
A & \in \quad P \rightarrow E \\
P & \in \quad x \mid k \, P_1 \, \dots \, P_n \qquad (n \geq 0)
\end{array}$$

Types

$$T \in a \mid f \mid T_1 \, T_2$$

Atoms	Syntactic Classes
f type constructors	D declarations
a type variables	K data constructor declarations
x, y term variables	E expressions
k data constructors	A alternatives
	P patterns
	T types

Figure 5.6: Abstract syntax for core Haskell.

$$\begin{array}{ll}
\mathcal{D} \llbracket x :: T \rrbracket \Rightarrow x :: \text{Context} \rightarrow \mathcal{T} \llbracket T \rrbracket & (\text{TySig}) \\
\\
\mathcal{D} \llbracket x \ y_1 \ \dots \ y_n = E \rrbracket \Rightarrow & \\
\quad x \ c_0 = \text{fun}_n (\backslash y_1 \ c_1 \ \dots \ y_n \ c_n \rightarrow & \\
\quad \quad \text{call } c_{0/n} \ "x" \ [\text{V } y_1, \ \dots, \ \text{V } y_n] & (\text{FunBind}) \\
\quad \quad (\backslash i \rightarrow \mathcal{E} \llbracket E \rrbracket_i)) & \\
\\
\mathcal{D} \llbracket x = E \rrbracket \Rightarrow x = \text{constant } "x" (\backslash i \rightarrow \mathcal{E} \llbracket E \rrbracket_i) & (\text{PatBind}) \\
\\
\mathcal{D} \llbracket \text{data } f \ a_1 \ \dots \ a_n = K_1 \mid \dots \mid K_m \rrbracket \Rightarrow & \\
\quad \text{data } f \ a_1 \ \dots \ a_n = \mathcal{K} \llbracket K_1 \rrbracket \mid \dots \mid \mathcal{K} \llbracket K_m \rrbracket & (\text{Data}) \\
\\
\mathcal{K} \llbracket k \ T_1 \ \dots \ T_n \rrbracket \Rightarrow k \ \mathcal{T} \llbracket T_1 \rrbracket \ \dots \ \mathcal{T} \llbracket T_m \rrbracket & (\text{ConDecl})
\end{array}$$

Figure 5.7: Transformation of declarations.

5.7.2 Notation used in the transformation rules

Each rule is named by an uppercase calligraphic letter. Double square brackets ‘ $\llbracket \ \rrbracket$ ’ enclose arguments which denote a syntactic entity, such as an expression, or a declaration, and so on. Most of the transformation rules have just one argument, but the rules for expressions and alternatives have two, and are written in this style: $\mathcal{E} \llbracket E \rrbracket_i$. The subscript i is the second argument, and denotes the name of the variable which contains the current context value. Each equation is also given a mnemonic name, written to the right in Sans Serif font. Variables are sometimes annotated with salient attributes: x^{let} denotes a let-bound variable, x^λ denotes a lambda-bound variable, k^n denotes a data constructor with arity n . Two new term variables are introduced, called c and i ; these are always bound to values of type **Context**.

5.7.3 Declarations

Figure 5.7 defines the rules for the transformation of declarations.

TySig transforms type annotations. In the core language type annotations are only given to let-bound identifiers. Each let-bound entity receives a new first parameter, which represents the context where the entity is first mentioned by name. The original type of the identifier is preceded by ‘**Context** ->’ to account for this extra parameter. The original type of the identifier, T , is transformed by \mathcal{T} (Figure 5.9), replacing all occurrences of -> with **F**.

FunBind transforms function bindings, and is the most complex of all the rules. It is best described from the inside out. The original body of the function, E , is wrapped in a lambda abstraction, which binds the context argument i . Each function in E must be applied to i , so E is transformed by \mathcal{E} (Figure 5.8), with i as an argument. The transformed body is wrapped in an application of **call** to construct an EDT node. In addition to the body, **call** takes three other arguments. The first argument is the context value, either c_0 or c_n , depending on whether the function is printed in the extensional style or the intensional style. The second argument represents the name of the function encoded as a string. The third argument is a list of the function’s arguments, encoded in the **Value** type. For each argument of the original function, y_n , a new context argument, c_n , is introduced. The function is encoded in the **F** type by wrapping it in a call to **fun_n**, where n is its arity. The initial calling context of the function, c_0 , is not included in the lambda abstraction passed to **fun_n**, instead it appears as a separate argument bound in the function’s head. This is because c_0 is the only context which is guaranteed to be the same as where the function is referred to by its let-bound name — simply because it represents the place where the function is applied to zero arguments. All the remaining contexts represent places where the function has been applied to at least one argument, which could be different to c_0 , and thus must be part of the encoded representation of the function.

As an aside, the finer details of Haskell’s type system make it desirable to avoid

transforming function bindings into pattern bindings. The reason is that Haskell’s type system forbids pattern bindings with overloaded types, *unless* the binding is accompanied by an appropriate type annotation. The name for this restriction is the (dreaded) *Monomorphism Restriction* [Peyton Jones, 2002, Section 4.5.5]. Function bindings do not suffer the same restriction. Therefore it is possible that the original program contains an overloaded function binding without a type annotation. Transforming such a function to a pattern binding would invalidate the Monomorphism Restriction. This is important in `buddha` because the transformation is source-to-source, which means that the output must be a type-correct Haskell program. The problem could be solved if the types of all let-bindings are known by the transformation, thus allowing type annotations to be introduced where necessary. Type checking Haskell, especially with all the popular language extensions in place, is by no means a trivial matter, and it is an advantage to build the program transformation without having to check types first.

`PatBind` transforms pattern bindings, following the scheme from Section 5.5.

`Data` and `ConDecl` transform data type declarations and data constructor declarations respectively. The structure of the declarations remains the same, but the types that they refer to are transformed by \mathcal{T} .

5.7.4 Expressions

Figure 5.8 defines the rules for the transformation of expressions and alternatives. The first five rules, which deal with variables, data constructors, and function applications are the most important.

`ExpVarLam` and `ExpVarLet` transform variables. Lambda-bound variables are unchanged. Let-bound variables are applied to i , the current context value. `ExpConZero` and `ExpConN` transform data constructors. Constructors of arity zero are constants, and remain unchanged. Constructors of non-zero arity are functions, and they must be encoded. Encoding is performed by a family of functions `conn`, reminiscent of

$\mathcal{E} \llbracket x^\lambda \rrbracket_i \Rightarrow x$	(ExpVarLam)
$\mathcal{E} \llbracket x^{let} \rrbracket_i \Rightarrow x \ i$	(ExpVarLet)
$\mathcal{E} \llbracket k^0 \rrbracket_i \Rightarrow k$	(ExpConZero)
$\mathcal{E} \llbracket k^n \rrbracket_i \Rightarrow \text{con}_n \ k$	(ExpConN)
$\mathcal{E} \llbracket E_1 \ E_2 \rrbracket_i \Rightarrow \text{apply } \mathcal{E} \llbracket E_1 \rrbracket_i \ \mathcal{E} \llbracket E_2 \rrbracket_i \ i$	(ExpApp)
$\mathcal{E} \llbracket \text{let } D_1 \ \dots \ D_n \ \text{in } E \rrbracket_i \Rightarrow$ $\text{let } \mathcal{D} \llbracket D_1 \rrbracket \ \dots \ \mathcal{D} \llbracket D_n \rrbracket \ \text{in } \mathcal{E} \llbracket e \rrbracket_i$	(ExpLet)
$\mathcal{E} \llbracket \text{case } E \ \text{of } A_1 \ \dots \ A_n \rrbracket_i \Rightarrow$ $\text{case } \mathcal{E} \llbracket E \rrbracket_i \ \text{of } \mathcal{A} \llbracket A_1 \rrbracket_i \ \dots \ \mathcal{A} \llbracket A_n \rrbracket_i$	(ExpCase)
$\mathcal{E} \llbracket \backslash y_1 \ \dots \ y_n \rightarrow E \rrbracket_i \Rightarrow$ $\mathcal{E} \llbracket \text{let } x \ y_1 \ \dots \ y_n = E \ \text{in } x \rrbracket_i$	(ExpLambda)
$\mathcal{A} \llbracket P \rightarrow E \rrbracket_i \Rightarrow P \rightarrow \mathcal{E} \llbracket E \rrbracket_i$	(Alt)

Figure 5.8: Transformation of expressions and alternatives.

the fun_n family, where n is the arity of the constructor:

```

con1 :: (a -> b) -> F a b
con1 f = MkF (\x _ -> f x)

con2 :: (a -> b -> c) -> F a (F b c)
con2 f = MkF (\x _ -> con1 (f x))

...

```

The only difference between con_n and fun_n is that applications of data constructors do not produce EDT nodes, so they simply ignore any context arguments which are passed to them (hence the underscores in each lambda underneath **MkF**).

ExpApp transforms function applications, with E_1 the function, and E_2 the ar-

$$\begin{array}{ll}
\mathcal{T} \llbracket a \rrbracket \Rightarrow a & (\text{TypeVar}) \\
\mathcal{T} \llbracket T_1 \ T_2 \rrbracket \Rightarrow \mathcal{T} \llbracket T_1 \rrbracket \ \mathcal{T} \llbracket T_2 \rrbracket & (\text{TypeApp}) \\
\mathcal{T} \llbracket -> \rrbracket \Rightarrow F & (\text{TypeConArrow}) \\
\mathcal{T} \llbracket f \rrbracket \Rightarrow f \quad (f \neq ->) & (\text{TypeCon})
\end{array}$$

Figure 5.9: Transformation of types.

gument. Both E_1 and E_2 are recursively transformed by \mathcal{E} . $\mathcal{E} \llbracket E_1 \rrbracket_i$ is an encoded function, so it must be decoded by **apply**, before it can receive its argument. Each application also receives the current context value i , just in case the application is saturated (which explains why i is passed everywhere through \mathcal{E}). In Section 6.6 we introduce additional rules to optimise the transformation of statically saturated function applications, eliminating redundant function encodings in cases where the function is immediately applied.

The remaining rules transform let expressions, lambda expressions, case expressions and alternatives. Lambda expressions are translated into equivalent let form before transformation, as discussed in Section 5.6.1. The rest are simply boilerplate traversals of the abstract syntax tree.

5.7.5 Types

Figure 5.9 defines the rules for the transformation of types. The function type constructor $->$ is replaced everywhere with F , by **TypeConArrow**, accommodating the encoding of functions in **FunBind** and **ExpConN**. Otherwise the types remain the same.

5.8 Correctness

What does it mean for the transformation to be correct? In broad terms it means that it does not change the externally observable behaviour of the debuggee in any significant way (other than increasing its space and time consumption), and that it causes the debuggee to build an EDT which is suitable for declarative debugging. We now briefly consider each of these issues in turn.

5.8.1 Semantics

When we talk of the meaning of programs we normally refer to formal semantics. However, there is no standard definition of Haskell's semantics, in either of the denotational, or operational styles [Peyton Jones et al., 2006]. Whilst this is unfortunate, it is not without reason:

- A complete denotational semantics must capture the meaning of entire Haskell programs, which requires a proper treatment of I/O, and thus interactive programs. So far this goal has been out of reach.
- The operational semantics of Haskell is left open to allow for some flexibility in the underlying implementation.

Even if we were to create our own semantics for the core language in Figure 5.6, we would face difficulty because:

- Our transformation makes extensive use of `unsafePerformIO`, which is particularly difficult to model without recourse to a specific evaluation order. Thus, a simple denotational semantics will not be sufficient.
- We would like to show that our transformation works correctly on top of *any* Haskell implementation, which could not be done by choosing a specific operational semantics. Also, we are unlikely to be able to prove that any particular compiler supports our semantics precisely.

For these reasons we believe a less formal argument is necessary, and more economical.

In general, the transformation preserves the meaning of the original program because:

1. Reduction steps are annotated by side-effecting operations which construct the EDT as an external data structure, however that structure is not depended upon by any value computed by the rest of the program. We can erase the construction of the EDT without changing the meaning of the transformed program.
2. Outside the construction of the EDT, the only change that the transformation makes to the program is to add extra parameters to functions which are matched by supplying extra (context) arguments to function applications. The values of the context arguments are not scrutinised outside of the code which constructs the EDT node. Otherwise the control flow of the program remains exactly the same as before.

Consider the transformation of function declarations (`FunBind`). The call to `funn` provides a compile time type tag on functions, but it serves no important task at runtime, so it can safely be inlined and eliminated, giving us a simplified transformation rule:

$$\begin{aligned}
 \mathcal{D} \llbracket x \ y_1 \ \dots \ y_n = E \rrbracket &\Rightarrow \\
 x \ c_0 = \backslash y_1 \ c_1 \ \dots \ y_n \ c_n \rightarrow \text{call } c_{0/n} \ "x" \ [\text{V } y_1, \ \dots, \ \text{V } y_n] & \\
 (\backslash i \rightarrow \mathcal{E} \llbracket E \rrbracket_i) &
 \end{aligned}$$

A minor improvement is to shift all the argument patterns to the left-hand-side of the equation:

$$\begin{aligned} \mathcal{D} \llbracket x \ y_1 \ \dots \ y_n = E \rrbracket &\Rightarrow \\ x \ c_0 \ y_1 \ c_1 \ \dots \ y_n \ c_n &= \text{call } c_{0/n} \ "x" \ [V \ y_1, \ \dots, \ V \ y_n] \\ &(\backslash i \rightarrow \mathcal{E} \llbracket E \rrbracket_i) \end{aligned}$$

We are only interested in the behaviour of the debuggee, and not the value of the EDT, so we can eliminate all the parts of `call` that deal with the EDT in any detail. To begin with we can remove its first three arguments:

$$\mathcal{D} \llbracket x \ y_1 \ \dots \ y_n = E \rrbracket \Rightarrow x \ c_0 \ y_1 \ c_1 \ \dots \ y_n \ c_n = \text{call } (\backslash i \rightarrow \mathcal{E} \llbracket E \rrbracket_i)$$

The representation of `Context` values is irrelevant outside of `call`, so we can model them as constants:

```
data Context = K
```

This allows us to treat `call` as if it were a pure function:

```
call :: (Context -> a) -> a
call body = body K
```

Now we can eliminate `call` altogether by inlining the above definition:

$$\mathcal{D} \llbracket x \ y_1 \ \dots \ y_n = E \rrbracket \Rightarrow x \ c_0 \ y_1 \ c_1 \ \dots \ y_n \ c_n = \mathcal{E} \llbracket E \rrbracket_K$$

To emphasise that the value of the additional context arguments are now irrelevant, we can replace them with “wildcard” patterns:

$$\mathcal{D} \llbracket x \ y_1 \ \dots \ y_n = E \rrbracket \Rightarrow x \ _ \ y_1 \ _ \ \dots \ y_n \ _ = \mathcal{E} \llbracket E \rrbracket_K$$

We can perform a similar kind of reduction to the rule for pattern bindings (`PatBind`), which simplifies to:

$$\mathcal{D} \llbracket x = E \rrbracket \Rightarrow x \ _ = \mathcal{E} \llbracket E \rrbracket_K$$

Now it is just a question of what the transformation does to expressions. We observe that the context argument is only used in two places: `ExpVarLet` and `ExpApp`. `ExpVarLet` says that every let-bound variable receives a context value as its first argument. Looking at the above simplified rules for function and pattern bindings we can see that all let-bindings have at least one argument, and that first argument does not affect the result of the definition. `ExpApp` says that every function application receives a context value as its second argument. We can eliminate the use of `apply` since at runtime it is equivalent to the identity function. We observe that every function receives an extra argument for each of its normal arguments, and those extra arguments do not affect the result of the function (non-nullary constructors get extra arguments, by way of `conn`, which can be inlined in a similar fashion to `funn`). The remaining rules just propagate the context arguments throughout the expression, but otherwise the structure of the expression remains the same.

In summary, if we ignore the side-effects which are used to construct the EDT, the program transformation simply adds extra redundant parameters to functions, and extra constant arguments to function applications. The two changes effectively cancel each other out, and so the meaning of the program is not changed in any significant way.

5.8.2 EDT correctness

The debugger is correct if it is sound and complete. In general terms, soundness means that every diagnosis returned by the debugger identifies a truly buggy equation in the program (in terms of the oracle's intended interpretation), and completeness means that if we have a program execution which exhibits a bug symptom, then the debugger will return a diagnosis of some bug. (Note we only return one bug at a time).

There are three key ingredients that determine the diagnosis of the debugger: the wrong answer diagnosis algorithm, the intended interpretation of the oracle, and the EDT. We do not get to choose the intended interpretation; it is simply a parameter

of the wrong answer diagnosis algorithm. Therefore, the correctness criteria must hold for all possible intended interpretations. It is trivial to show that, given an EDT in which all paths are finite, the wrong answer diagnosis algorithm will return some diagnosis in a finite number of steps (it could either find a buggy node, or not find any bugs). The question as to whether it is a correct diagnosis is therefore determined by the relationship between the EDT and the program execution from which it is derived. So soundness and completeness can be cast in terms of the structure and content of the EDT.

EDT soundness

The EDT is sound if every sub-tree constitutes a valid proof of the reduction in its root node. If this holds then the wrong answer diagnosis algorithm always identifies buggy equations, following from the argument presented in Section 3.3.2. We build a big-step EDT according to Nilsson's rule for direct evaluation dependency in Figure 3.2. Therefore, we must show that this rule is sound.

Consider the general form of a function declaration: $f\ x_1 \dots x_n = e$. Given some argument terms $a_1 \dots a_n$, a single step reduction of the application $f\ a_1 \dots a_n$ produces the term $e[a_1/x_1 \dots a_n/x_n]$. That is, we get an instantiation of the body of the function, e , such that each of the parameters of f are replaced by their corresponding argument terms. We construct a node in the EDT for the reduction of this application like so:

$$f\ a_1\Downarrow \dots a_n\Downarrow \Rightarrow (e[a_1/x_1 \dots a_n/x_n])\Downarrow$$

where \Downarrow returns the most evaluated representation of a value in a given program run. We observe that, because Haskell is purely functional, it does not matter whether we evaluate the argument terms before or after we evaluate the body of the function:

$$e[a_1/x_1 \dots a_n/x_n] = e[a_1\Downarrow/x_1 \dots a_n\Downarrow/x_n]$$

There is no risk of non-termination because we assume that the whole program run is terminating, so the final state of all intermediate terms will also be terminating.

Let $\mathcal{L}_1 \Rightarrow \mathcal{R}_1 \dots \mathcal{L}_k \Rightarrow \mathcal{R}_k$ be the reductions in the children nodes. The EDT node is sound if:

$$\mathcal{L}_1 = \mathcal{R}_1, \dots, \mathcal{L}_k = \mathcal{R}_k \vdash_{\mathcal{H}} e[a_1 \Downarrow/x_1 \dots a_n \Downarrow/x_n] = (e[a_1 \Downarrow/x_1 \dots a_n \Downarrow/x_n]) \Downarrow$$

In other words, we must be able to show that the initial and final instances of the function body are equal, using only the equalities derived from the reductions in the children nodes, plus any number of reductions of system redexes.

Soundness is established (informally) as follows. We assume a complete record of the reduction history from some initial start term t_0 . A *needed redex* is any instance of a redex which was eventually reduced in the evaluation of t_0 . We allow “instances of a redexes” because we are intentionally reordering the evaluation of terms. The arguments of a needed redex may be in different states of evaluation than when the reduction originally took place. An *innermost needed redex* is a needed redex which does not contain any other needed redexes in its sub-terms. Let us use the names e_{init} for $e[a_1 \Downarrow/x_1 \dots a_n \Downarrow/x_n]$, and e_{final} for $(e[a_1 \Downarrow/x_1 \dots a_n \Downarrow/x_n]) \Downarrow$. We can show that e_{init} can be reduced to e_{final} using some finite number of small-step reductions of system redexes and some finite number of big-step reductions for program redexes.

The general idea is to repeatedly reduce innermost needed redexes until there are none left. If a term has no innermost needed redexes, it is in its most evaluated state.⁴ We start with e_{init} , and select an innermost needed redex (there may be zero, one, or more). If there are no innermost needed redexes, then e_{init} is in its final state of evaluation, and soundness is trivial. If there is an innermost needed redex in e_{init} , it could either be a system redex or a program redex. If an innermost needed redex is a system redex, then we can reduce it by one step, and update e_{init} correspondingly. The evaluation of a system redex by one step is trusted to

⁴If there are no innermost needed redexes, then there can be no outer needed redexes, thus there are no needed redexes at all.

be correct. We do not construct any children nodes for such cases. If an innermost needed redex is a program redex, then we can reduce it to its final state of evaluation by a big-step reduction, and update e_{init} correspondingly. If this innermost needed redex corresponds to an instance of a function application appearing in e_{init} , then, following the rule for direct evaluation dependency, we construct a child node for this reduction. By definition, the argument terms of innermost needed redexes are in their final state of evaluation (they would not be *innermost* otherwise). Therefore, the argument and result terms in the new child node have the required big-step property. We can build the sub-tree for this child by applying the same technique recursively. After reducing the selected innermost needed redex, we obtain a new, more evaluated, instance of e_{init} . We select an innermost needed redex from the new term and continue until we arrive at a term which has no innermost needed redexes. If the evaluation of t_0 is terminating, there must be only finitely many times we can apply this process.

If e_{init} is an instance of the body of some function f , then the only innermost needed redexes in e_{init} must be instances of function applications which appear in the body of f . This is because all the parameters of f are replaced by terms in their most evaluated state, therefore the argument terms do not contain any innermost needed redexes. The reduction steps that we apply repeatedly to e_{init} and its successors do not introduce any new applications into the resulting term which later become innermost needed redexes. Therefore, we can reduce e_{init} to e_{final} using the process outlined above, *and* the only innermost needed redexes that we encounter along the way are those which are instances of applications appearing in the body of f . Those are precisely the reductions for which we construct children nodes. This means that the reductions in the children nodes, plus zero or more reductions of system redexes, are sufficient to show that e_{init} can be reduced to e_{final} .

The extensional style of printing functions requires us to adopt an unorthodox definition of redex, as noted in Section 3.5. Nonetheless, the soundness argument remains the same. Recall the example buggy program in Figure 3.8. We regard the

partial application ‘`plus 1`’ as a redex, which is reduced to ‘`{ 1 -> 0, 2 -> -1 }`’.

At some point we will have a node in the EDT corresponding to this reduction:

$$\text{map } \{ 1 \rightarrow 0, 2 \rightarrow -1 \} [1,2] \Rightarrow [0,-1]$$

Reducing the left-hand-side of the reduction by one step using the definition of `map` gives:

$$\{ 1 \rightarrow 0, 2 \rightarrow -1 \} 1 : \text{map } \{ 1 \rightarrow 0, 2 \rightarrow -1 \} [2]$$

There are two innermost needed redexes in this term. The leftmost one corresponds to the application of an “extensional” function to an argument:

$$\{ 1 \rightarrow 0, 2 \rightarrow -1 \} 1$$

Whilst this is not a function application in the traditional Haskell sense, it is nonetheless obvious that it “reduces” to 0. Since this reduction does not involve any let-bound function names, it is not a program redex, so we treat it as a new type of system redex, which is always correct. Hence, there is no need to create a child node for it. The rightmost innermost needed redex corresponds to the recursive application of `map`, which is a program redex, so it does generate a child node, as usual. A big-step reduction of this redex produces `[-1]`. Combining the single-step reduction of the system redex and the big-step reduction of the program redex, we can show that the left-hand-side term can be reduced to the right-hand-side term.

EDT completeness

The EDT is complete if and only if the following two conditions are met:

1. An externally observable bug symptom implies that the root node is erroneous.
2. All paths in the EDT are finite.

The top node of the EDT corresponds to the evaluation of `main`, which is the entry point of every Haskell program. The value of `main` is, in effect, the value of the whole program. If there is a bug in the program it will be observable in the value that `main` reduces to. In practice this issue is complicated by the fact that Haskell

programs can have side-effects. The value returned by `main` has type `IO`, which is an abstract data type. Therefore, printing the value of `main` is more difficult than an ordinary concrete data type. This is a long-standing problem for declarative debuggers for purely functional languages. We provide a solution in Section 7.2.1, which allows the side-effects associated with `IO` values to be printed in a convenient manner.

The only place where cyclic dependencies can arise in the EDT is between mutually recursive pattern bindings. Following the discussion in Section 3.6, we break those cycles by arbitrarily deleting one of the edges in a loop. Therefore, all paths in the EDT are finite. As we noted in that section, this can result in an unsound diagnosis. Such cycles are rare in practice, nonetheless the problem can be avoided by allowing the diagnosis returned by the debugger to contain more than one reduction. That is, we effectively judge the correctness of all reductions in a mutually recursive set of pattern bindings together. We avoid the cycle by merging the nodes together. Thus, we trade precision in the diagnosis for soundness. We plan to incorporate this solution in future versions of `buddha`.

5.9 Performance

Transformed programs will always run slower and use more space than their original versions. But by how much? Space usage is the proverbial Achilles' Heel of declarative debugging. EDT nodes store references to the intermediate values computed by the program. Building the whole tree means keeping all its intermediate values, thus rendering garbage collection ineffectual. The space complexity of a debugger which materialises the whole EDT at once is proportional to the running time of the debuggee. In practice, for most programs, the debugger would run out of memory in a matter of seconds on a typical desktop computer. High space consumption can also have an adverse affect on runtime performance. For instance, greater heap retention of data can increase garbage collection times, because more memory must

be traversed at each collection.⁵ In the worst case, if the space requirements exceed the real memory of the computer, the debugger is likely to induce thrashing in the virtual memory subsystem, which can lead to several orders of magnitude degradation in runtime performance.

It is interesting to investigate how much overhead the transformation introduces *besides* the EDT itself. In order to gauge this cost, the definitions of `call` and `constant` were modified to build an empty EDT. Nodes were constructed for each reduction, but not inserted into their parents, thus allowing them to be garbage collected immediately. Five moderately complex programs were transformed and executed, and measurements were taken to see the effect on space and time. The following programs were tested:

- Raytrace: a simple grey-scale ray-tracer, applied to a scene containing seven spheres and two light sources. The output image size is 250×250 pixels.
- Prolog: Mark Jones' Prolog interpreter, applied to a query which reverses a list of 360 numbers.
- Pancito: version one of Andrew Cooke's image generation program⁶, producing a colour image of 50×50 pixels.
- Mate: Colin Runciman's solver for chess end-games, applied to the Wurzburg input case.
- Cacheprof: Julian Seward's cache profiling tool, applied to a 13708 line assembly program.

Prolog, Mate and Cacheprof are part of the Nofib benchmark suite [Partain, 1993], which is distributed with GHC.

All testing was performed on a Intel Pentium 4 CPU clocked at 1.60 GHz, using the Linux operating system with kernel version 2.6.3. GHC version 6.2.1 was used

⁵A generational garbage collector can reduce this cost, by traversing long-lived data less frequently than newly constructed data.

⁶<http://www.acooke.org/jara/pancito>

<i>program</i>	<i>nodes</i> (million)	<i>growth (trans / orig)</i>		
		<i>compile time</i>	<i>max. mem</i>	<i>run time</i>
Raytrace	156.6	2.1	1.5	10.0
Prolog	29.3	2.8	1.5	4.0
Pancito	62.9	2.5	2.3	13.7
Mate	10.9	2.4	3.5	12.1
Cacheprof	73.8	2.1	1.6	22.5
<i>average</i>		2.4	2.1	12.46

Figure 5.10: Relative performance of the transformed program compared to the original, when an empty EDT is constructed.

to compile the programs with optimisation turned on.⁷

The results of the tests are shown in Figure 5.10. The first column lists the name of the program. The second column lists the number of EDT nodes (in millions) that would be allocated by the debugger, had the whole tree been kept. The remaining columns list the ratio of the transformed program over the original program in terms of compile time, maximum memory usage, and run time. All functions in the program were transformed by the rules shown in this chapter, including all the functions in the standard libraries.

An obvious question is how much overhead is acceptable? Absolute figures are hard to estimate without extensive user testing. Nonetheless, there are a couple of observations that can be drawn from the data.

First, compile time and maximum memory usage are both very close to a factor of two larger than the original. This is probably because every function application is doubled in size, including applications of data constructors. More importantly, the data suggests that the transformation (without the EDT) preserves the asymptotic space complexity of the debuggee — it introduces no space leaks into the program. This means that the biggest wins in optimising the space behaviour of the debugger will be had by reducing the size of the EDT.

Second, the increase in run time is about a factor of 12. Since the test cases produce empty EDTs this figure does not provide a good measure of how long the

⁷The testing discussed in later chapters is based on the same setup.

debugger is likely to take. However, it does indicate that there is a fairly high cost to be paid by the transformation, irrespective of the EDT. Profiling each of the test programs reveals that the computation time is dominated — as one would expect — by `call` and `constant` (about 30 percent). This suggests that the time complexity of the debugger can be improved by reducing the number of calls to `call` and `constant`, and also by reducing the work done on average in each call.

In Section 7.4 we return to the issue of runtime performance, and we show how the space usage of the EDT can be reduced; we use the same test programs in that section as the ones we have used here.

5.10 Final remarks

This chapter has shown how the EDT can be constructed by program transformation. However, the story does not end here. In Chapter 6 the rules of the transformation are extended slightly to allow printable representations of values to be observed. And Chapter 7 extends the transformation rules even further to improve the time and space complexity of the debugger.



Chapter 6

Observing Values

The problem is that it can become very hard to find the small residue of bugs that get past the type checker, because the usual debugging tools (shoving print statements into the program, watching the sequence of events in an animated window, etc.) run into trouble with the restrictions imposed by pure functional languages.

www.dcs.gla.ac.uk/~jtod/research

JOHN O'DONNELL

6.1 Introduction



DEBUGGERS are used to “look inside” programs whilst they are under evaluation, providing a window on their otherwise hidden mechanics. Central to this is the illustration of intermediate values that are generated as the program executes. The values of interest in **buddha** are the arguments and results of function applications. The challenge is to find a way to communicate these values to the oracle in a manner that is compatible with its intended interpretation of the program. In essence, a representation is needed into which all values can be mapped, and from which a suitable printed form can be rendered. The process of mapping a value into a suitable representation is called an *observation* of

that value.

Non-strict semantics, pervasive higher-order programming, and strong static type all make observation *in Haskell* difficult.

Observing values from a running computation is one instance of the more general field of reflection. Some languages, such as Lisp and Prolog, excel in this area, because reflection is an integral part of their design. Haskell does not follow this tradition. The focus of this chapter is how to work around this limitation.

6.1.1 Outline of this chapter

The rest of this chapter proceeds as follows. Section 6.2 considers the key requirements of observation, which are used to guide us through the design space. Section 6.3 briefly investigates implementing observation entirely in Haskell, concluding that the language is not quite expressive enough for the task. The solution employed in *buddha* is to add a new observation primitive to the runtime environment, via the Foreign Function Interface. Section 6.4 covers this idea in detail. Section 6.5 looks at printing functional values in both the intensional and extensional styles. Section 6.6 defines an optimisation of the transformation rules from Chapter 5, which reduces the cost of the transformation for statically saturated function applications. Section 6.7 considers the problems of displaying large values and points to some potential solutions.

6.2 The requirements of observation

An observation must be:

- **Universal:** it must be able to derive a suitable representation for any value from any program.
- **Objective:** it must not inadvertently alter the state of evaluation of any observable part of the program being debugged.
- **Final:** it must observe the value in its most evaluated form.

Universality ensures that the debugger can support all Haskell programs.

Objectivity means that one observation does not change the outcome of any future observations, nor change the state of the value under inspection. It is generally unsafe to use a non-objective observation because any further evaluation of the program being debugged might trigger a divergent computation.

Finality is required by a big-step EDT, as noted in Section 3.3. It is not easy to predict when each value will reach its final state. The standard approach — which we follow — is to delay all observations until the debuggee has terminated. This means that intermediate values, that would normally be garbage collected, must be retained in the heap. This is done implicitly by the construction of the EDT.

6.3 A Haskell implementation

Writing the debugger in Haskell enhances its portability. This section considers whether Haskell is sufficiently expressive to implement an observation mechanism which satisfies the three requirements outlined above.

A universal observation facility must accept any type of value as its argument and map it into an appropriate representation:

```
data Rep = ...      -- representation of Haskell values
observe :: a -> Rep
```

Unfortunately, Haskell's type system prevents us from writing a useful `observe` with the above type scheme. An implementation of `observe` must inspect (and deconstruct) its argument to produce an accurate representation. The only way to inspect a value in Haskell is to pattern match against it, but this would constrain `observe`'s argument to a single monomorphic type.

Dynamic types provide a solution to this problem with a construct called *type-case*, which allows the type of a (possibly polymorphic) expression to be scrutinised at runtime. With this facility a program can dispatch to a type-specific observation method. There has been much interest in providing dynamic typing capabilities in otherwise statically typed languages, see for example [Leroy and Mauny, 1993,

```
data Rep
  = Ident String
  | App Rep Rep

class Observable a where
  observe :: a -> Rep

instance Observable Bool where
  observe True  = Ident "True"
  observe False = Ident "False"

instance Observable a => Observable [a] where
  observe [] = Ident "[]"
  observe (x:xs)
    = App (App (Ident ":") $ observe x) $ observe xs
```

Figure 6.1: A type class for generic observation.

Abadi et al., 1995, Bentley Dornan, 1998, Pil, 1998]. Haskell does not have type-case, but the feature can be simulated to some extent with type classes. The idea is to constrain the argument of `observe` to a class, say `Observable`, such that each instance of the class provides a type specific observation method:

```
observe :: Observable a => a -> Rep
```

Then it is simply a matter of providing instances of `Observable` for each type in the program.

Figure 6.1 provides a very simple version of `Rep` and shows what the class might look like, including example instances for booleans and lists. A similar approach is proposed by Sparud [1999].

The main limitation of type classes, when compared with typecase, is that only values of monomorphic type can be observed. In *buddha* this means that polymorphic expressions can become ambiguous after transformation. The following example illustrates the problem:

```
let id x = x in id []
```

In the hypothetical transformed version of `id` we would apply `observe` to `x` to obtain its printable representation. This would give rise to a type class constraint like so:


```
id :: Observable a => ...
```

The application ‘`id []`’ gives rise to the constraint ‘`Observable [a]`’, but this cannot be satisfied, since `a` is a type variable, not a concrete type. A possible solution would be to coerce all unresolved instances of `Observable` to some default concrete type such as the unit type, akin to Haskell’s defaulting rules for unresolved numeric overloading. To do this, one would need to either perform type analysis of the transformed program, or extend the defaulting mechanism of the Haskell compiler.

There are two other problems with writing `observe` in Haskell, though they are not specifically related to the use of type classes:

1. There is no way to determine if something is a thunk.
2. Function values cannot be observed.

The first problem occurs because it is impossible to determine the state of evaluation of an expression from within Haskell. Pattern matching is the only tool that Haskell provides for decomposing values, but it prohibits objective observation because it forces the evaluation of the object being scrutinised.

The second problem occurs because functions are abstract values. Unlike algebraic data types, functions cannot be decomposed by pattern matching.

6.4 An observation primitive

Adding `observe` as a primitive function in GHC is relatively simple, at least for non-functional values, because the Foreign Function Interface (FFI) makes it possible to call C from Haskell, and the GHC runtime environment is written in C. Also, the runtime environment provides a convenient interface for traversing the internal representations of heap objects.

Observing functions, especially with the extensional representation, remains difficult, but this can be solved by enhancing the program transformation, as discussed shortly in Section 6.5.

It is straightforward to make observation objective and universal in C. Crawling over the heap representation of a value in C does not cause it to be evaluated, and thunks are easily identified. Also, in C, all representations of Haskell values in the heap have the same type, which means that a fully polymorphic implementation of `observe` is achievable.

We provide an observation function with this type:

```
observe :: a -> Rep
```

Underneath this is a call to a C function, `observeC`, which takes a pointer to the object under consideration as an argument, and proceeds to build a representation of that object. The representation encodes identifiers, data constructor applications, primitive types, and thunks, as follows:

```
data Rep
  = Ident Identifier  -- identifiers, from Figure 3.3
  | App Rep Rep       -- applications
  | Prim PrimData     -- primitive data
  | Thunk             -- unevaluated expressions

data PrimData
  = PrimChar Char     -- Characters
  | ...              -- Ints, Integers, etcetera
```

Note that `observe` is impure because it encodes thunks explicitly in the representation. The possibility of discovering a thunk in a given value depends on when it is observed relative to the progress of the rest of the program. Therefore the result of `observe` depends on the value of its argument *and* implicitly an external environment. Normally this kind of impurity is handled in Haskell by the `IO` type, but a pure interface makes it simpler to use. Though `observe` is technically impure, it is used safely in `buddha`, because values are only observed once the execution of the debuggee has terminated; after that point any two observations of the same value will always return the same result.

The implementation of `observeC` requires two problems to be solved:

1. How to construct Haskell values of type `Rep` in C.
2. How to obtain source-level identifiers.

Haskell values can be constructed in C using the following function provided by GHC's runtime interface:

```
HaskellObj rts_apply (HaskellObj, HaskellObj);
```

where `HaskellObj` is the C type of all Haskell expressions. Values of type `Rep` and `PrimData` are built up from their data constructors, which are passed in to `observeC` as pointer arguments.

Source-level identifiers are obtained by compiling the program for profiling. In this mode GHC retains source-level identifiers for Haskell objects, which can be accessed directly from an object's heap representation. This is really a stop-gap measure, rather than the final ideal solution. The main problem with this approach is that a program compiled for profiling runs considerably slower than a non-profiled version — even when no profiling statistics are gathered (as is the case in `buddha`).

One possible solution is to modify GHC (or whatever compiler is used) to include source names for data constructors in non-profiled code. Another alternative is to extend the program transformation so that data constructors are paired with their source-level name. The main benefit of the latter is that it is less reliant on the features of the compiler, and thus more likely to make the debugger portable to other compilers. One way to do this is to make the name the first argument of the constructor. For example, the `Maybe` type:

```
data Maybe a = Just a | Nothing
```

could be transformed to:

```
data Maybe a = Just String a | Nothing String
```

All occurrences of `Just` in the original program must be changed to `mkJust`:

```
mkJust = Just "Just"
```

Likewise for `Nothing`. Pattern matching rules need to be transformed accordingly to account for the additional argument to every constructor.

6.4.1 Cyclic values

Cyclic values require special consideration. Initially it was decided that cycles should be detected and made explicit in the representation. Observing cycles from within C is not difficult since pointer equality can be used for object identity, however it imposes some constraints on the way `observeC` must be implemented. The main limitation is that it is not possible to interleave the observation of a single value with execution of Haskell code. This is due to the garbage collector, which frequently moves objects around in the heap. Pointers to Haskell objects from C are invalidated once the object is moved to a new location, since pointers in C are not tracked and updated by the garbage collector. The net effect is that each object must be observed in entirety if it is observed at all. In some cases an object is so large that it is not feasible to print the whole thing at once; but even if only a small fraction is printed, it must be completely traversed. As a result, observing cycles adds considerable complexity to the code. This is rather unfortunate since cyclic values are relatively rare in practice.

It is also quite difficult to display cyclic values in a manner which is easy to comprehend. The most obvious way to print cyclic values is with the recursive *let* syntax of Haskell, but this can get quite unwieldy for large values. An alternative is to draw diagrams on a graphical display. After experimenting with various ways to display cyclic values we found that often the simplest view is to unfold them into trees, truncating them at some point to avoid an infinite printout. An added incentive of this approach is that it is much easier to implement than any method which shows cycles explicitly. Most importantly, the declarative program semantics does not distinguish between cyclic terms and their unfolded representations. There should never be an intended interpretation which relies on a value having a cyclic representation. The biggest problem with unfolded trees is that there is no way for the debugger to know how much printing is enough for a cyclic value. The solution in `buddha` is to let the user decide, by making the truncation threshold adjustable.

By ignoring cycles, and using truncation to limit the size of printed representa-

tions, `observe` can be lazy. For this to work, the implementation of `observeC` must simulate in C the kind of lazy construction of values that we would normally get in Haskell. For example, consider the following application of a data constructor `Con` to some argument values:

```
Con arg1 ... argn
```

A single application of `observeC` to this value should return the following Haskell expression as its result:

```
App ... (App (identifier for Con) (observe arg1))
      ... (observe argn)
```

Note that the observation of the sub-terms are suspended applications of (the Haskell function) `observe`, which means that a pointer to `observe` must be passed as an argument to `observeC`. Below is the C code which builds the appropriate Haskell expression for data constructor applications:

```
/* build an Ident from the constructor name */
tmp = rts_apply (ident, name (obj));

/* apply observe to each of the argument terms */
for (i = 0; i < numArgs (obj); i++)
{
    tmp = rts_apply
        (rts_apply (app, tmp),
         rts_apply (observe, obj->payload[i]));
}
return tmp;
```

The various new functions and variables are defined as follows: `obj` points to the Haskell object under scrutiny; `name` returns the source name of a data constructor; `numArgs` returns the number of arguments in a constructor application; `obj->payload[i]` accesses the *i*th argument of the application; `ident` points to the `Ident` constructor; `app` points to the `App` constructor; and `observe` points to the Haskell function of the same name.

6.5 Observing functional values

Buddha can print functions in two ways: the so-called intensional and extensional styles, which were first introduced in Chapter 3. The intensional style could be handled by `observe` in the same way as non-functional values, but the extensional style requires more support. This section shows that both styles of printing can be built on top of the program transformation rules from Section 5.6, with only minor modifications.

Recall that transformed functions are “encoded” in a new type called `F`:

```
newtype F a b = MkF (a -> Context -> b)
```

Encoded functions are created by a family of functions `funn`, where n is the arity of the original function, and are applied using `apply`. Below is the transformation of `const`, which serves as a running example throughout this section:

```
const :: Context -> F a (F b a)
const c0
  = fun2 (\x c1 y c2 ->
          call "const" [V x, V y] (c0, c2) (\i -> x))
```

The original purpose of the encoding is to avoid a typing problem, but it can also be used for printing. The idea is extend `F` by pairing the function with its representation like so:

```
data F a b = Intens (a -> Context -> b) IntensRep
           | Extens (a -> Context -> b) ExtensRep
```

IntensRep and *ExtensRep* stand for the intensional and extensional representations respectively.

When an encoded function is observed, the function value is skipped, and the representation is used. The data constructors `Intens` and `Extens` tell the printer which encoding is used in a given instance.

For each style of representation there are three issues that need to be resolved:

1. How to encode the two types of representation.
2. How to generate an initial representation of a function.
3. How to generate a new representation when a function has been applied to an argument.

The family of encoding functions is split into two, `funIn` for an intensional representation, and `funEn` for an extensional representation. New representations are generated when the encoded function is applied, combining the old representation with the new argument, and in the case of the extensional representation, the result of the application.

6.5.1 Intensional printing of functions

The intensional style is based on Haskell terms, using the existing `Rep` type:

```
data F a b = Intens (a -> Context -> b) Rep
```

The definition of `Rep` from Section 6.4 already accommodates partial applications of let-bound functions and data constructors, but not lambda functions. It would be possible to extend `Rep` to include lambda notation, however source coordinates are much simpler, and in principle the debugger could use the coordinates to fetch the original expression directly from the source file.

Rather than add a new entry to `Rep` for the sake of lambda functions, we just print them using their source code coordinates, which is contained in the identifier for the function. Lambda functions have the empty string as their name, which distinguishes them from regular identifiers.

The `apply` function is extended to accommodate the `Intens` constructor:

```
apply :: F a b -> a -> Context -> b  
apply (Intens f _) x context = f x context
```

The `funn` functions are extended to include a representation in the encoded version of a function.

Here is its original definition of `fun1`:

```
fun1 :: (a -> Context -> b) -> F a b
fun1 g = MkF g
```

And here is the new version which includes an intensional representation of the function:

```
funI1 :: (a -> Context -> b) -> Rep -> F a b
funI1 g rep = Intens g rep
```

For functions of arity more than one, `funIn` must build a new function representation each time the old encoded function is applied to an argument. Here is its original definition of `fun2`:

```
fun2 :: (a -> Context -> b -> Context -> c) -> F a (F b c)
fun2 g = MkF (\x c -> fun1 (g x c))
```

If `rep` is the representation of the function, and `x` is its argument, then a representation of an application of the function to `x` can be built using the following utility:

```
appRep :: Rep -> a -> Rep
appRep rep x = App rep (observe x)
```

Note the call to `observe` which generates a printable representation of the function's argument. Here we can see the advantage of avoiding the `I0` type in `observe`'s result.

Here is the extended version of `fun2`:

```
funI2 :: (a -> Context -> b -> Context -> c) -> Rep -> F a (F b c)
funI2 g rep
  = Intens (\x c -> funI1 (g x c) (appRep rep x)) rep
```

The definition of `funI3` and above follow the same pattern:

```
funI3 g rep = Intens (\x c -> funI2 (g x c) (appRep rep e)) rep
```

and so forth.

At some point an initial representation of a function must be created. This is done where the function is defined, for example:

```
const :: Context -> F a (F b a)
const c0
  = funI2
    (\x c1 y c2 ->
      call "const" [V x, V y] (c0, c2) (\i -> x))
    (Ident ("Prelude.hs", "const", 12, 1))
```

Note that the result type contains two nested applications of `F`, because `const` has two arguments. The outer instance of `F` encodes the function applied to zero arguments; its representation is simply the identifier which is passed as the second argument of `funI2`. The inner instance of `F` encodes the function applied to one argument; its representation is the combination of the function identifier and the representation of the argument. This new representation is built within `funI2` and passed as the second argument to `funI1`. For example, `const` applied to some expression `exp`, returns a function whose representation would be generated as follows:

```
App (Ident ("Prelude.hs", "const", 12, 1))
    (observe exp)
```

One thing to note is that laziness is crucial for correctness. The argument `exp` could be any expression. It may well be a saturated function application that is later reduced to WHNF. One of the requirements of observation is that values are shown in their most evaluated form, including those arguments of partially applied functions. Therefore it is unsafe to evaluate `'observe exp'` eagerly. Under lazy evaluation the application of `observe` will not be reduced until its value is needed, in other words, if and when the representation of the function is printed. The design of `buddha` ensures that printing only happens after the evaluation of the debuggee is complete.

Two small improvements of the transformation are possible. First, the identifier information can be shared between the function encoding and the construction of the EDT, like so:

```

const :: Context -> F a (F b a)
const c0
  = funI2
    (\x c1 y c2 ->
      call identifier [V x, V y] (c0, c2) (\i -> x))
    identifier
  where
    identifier = Ident ("Prelude.hs", "const", 12, 1)

```

Second, the correct parent for the intensional style is found when the function is fully applied. This corresponds to `c2` in the above example. Thus there is no need to pass the other parent, `c0`, to `call`, so it can be eliminated:

```

call identifier [V x, V y] c2 (\i -> x)

```

6.5.2 Extensional printing of functions

The extensional style collects the so-called minimal function graph for a particular invocation of a function. The representation type is a list of values, injected into the universal type `Value`:

```

type FunMap = [Value]

```

For example, a function that takes `'a'` to `'b'` and `'c'` to `'d'`, might be represented with the following map:

```

[V ('a', 'b'), V ('c', 'd')]

```

The order of entries in the map is determined by the order that the applications occur, but for the purposes of debugging it can be regarded as a multiset.

Each time an encoded function is applied, the argument and result values are added to the existing function map by destructive update. An `IORef` provides the mutable reference:

```

data F a b = Intens ...
           | Extens (a -> Context -> b) (IORef FunMap)

```

A new equation is added to `apply` for this purpose:

```

apply (Extens f funMap) arg context
  = recordApp arg (f arg context) funMap

```

The application of the function to its argument is recorded in the function map by way of `recordApp`:

```

recordApp :: a -> b -> FunMap -> b
recordApp arg result funMap
  = unsafePerformIO $ do
    map <- readIORef funMap
    writeIORef funMap (V (arg, result) : map)
    return result

```

A fresh empty map is allocated for every function value that is created in the execution of the program, *i.e.* whenever a function is invoked for the first time, including functions which are created by partial application. While it is possible to share function maps between two distinct instances of a given function, it is desirable to keep them separate in order to reduce the size of their printout. The initial function maps are constructed when encoded functions are created by the `funEn` family of functions. A first attempt at `funE1` might look like this:

```

funE1 :: (a -> Context -> b) -> F a b
funE1 g = Extens g (newIORef [])

```

However, a problem occurs because the underlined expression is constant. An optimising compiler can lift this expression to the top-level of the program, thus causing it to be evaluated once, instead of every time `funE1` is called. This means that all encoded functions produced by `funE1` will share the same mutable function map. The intended semantics is that each instance of an encoded function gets its own fresh copy. The solution employed in `buddha` is to make the construction of the new function map depend (artificially) on the argument of `funE1`, like so:

```

funE1 g = Extens g (newFunMap g)

newFunMap :: a -> IO FunMap
newFunMap x = newIORef [V x]

```

The compiler cannot lift the right-hand-side to the top level because it is no longer a constant expression. Of course this value should not be part of the final display

of the function, so the last value in every map is ignored by the printer. A similar approach is used to add function maps to `fun2` and above:

```
funE2 g = Extens (\x c -> funE1 (g x c)) (newFunMap g)
funE3 g = Extens (\x c -> funE2 (g x c)) (newFunMap g)
...
```

The observation of a function encoded with an extensional representation works in almost exactly the same way as an ordinary non-functional value, except when it comes to printing. The object is passed to `observe` as usual, which returns a representation. Normally representations are pretty printed in the obvious way, using Haskell-like syntax, however the pretty printer treats data wrapped in the `Extens` constructor specially.

The small example map mentioned above, for the function over characters, would appear to `observe` as the following data structure:

```
Extens <function>
  (<IORef>
    (V ('a', 'b') : (V ('c', 'd') : (<init> : []))))
```

`<function>` denotes the unencoded function, `<IORef>` denotes the internal representation of `IORefs` used by GHC, and `<init>` denotes the initial value placed in the map by `newFunMap`. When the pretty printer encounters a representation involving the `Extens` constructor it switches to an interpreted mode of printing, which effectively ignores everything in the above data structure except those parts which are underlined. The output is a string which shows just the mappings of the function, like so:

```
{ 'a' -> 'b', 'c' -> 'd' }
```

The transformation of `const` for the extensional style goes as follows:

```
const :: Context -> F a (F b a)
const c0
  = funE2
    (\x c1 y c2 ->
      call identifier [V x, V y] c0 (\i -> x))
  where
    identifier = Ident ("Prelude.hs", "const", 12, 1)
```

The correct parent for the intensional style is found in the context where the function is first mentioned by name, which corresponds to `c0`. Therefore, only `c0` is passed to `call`, as the underline indicates.

6.5.3 Combining both styles

The possibility different printing styles for functions raises three questions for the user interface of the debugger:

1. How does the user tell the debugger which style to use for a particular function or group of functions?
2. Can the two styles be used at the same time?
3. What is the appropriate style for library functions?

Buddha provides two methods for the user to determine which style of printing to use. The first method applies to a whole module at a time, which is given as a command line flag to the transformation program: `'-t extens'` for the extensional style, and `'-t intens'` for the intensional style. If no flag is set then the intensional style is taken as the default. The second method applies to individual function definitions, and is given in a separate “options file” (which also caters for trust annotations, see Section 7.3). The user can supply one such options file for each module in the program. Within the options file individual function names are associated with a flag which indicates the desired printing style. For example:

```
const ; extens
```

The options file also allows default styles to be declared, and has special syntax for functions defined in nested scopes, type classes, and instance declarations. Currently there is no way to refer to a lambda function specifically, but a workaround is for the user to manually bind the function to a name. If there is no options file for a given module, the command line settings are used.

Both styles of printing can be used in the same program without any difficulty, and this can give rise to composite function representations. Consider the following example:

```
compose f g x = f (g x)
inc x = x + 1
start = map (compose inc inc) [1,2]
```

If `inc` uses extensional style, and `compose` uses the intensional style, the argument of `map` will be printed as follows:

```
compose { 3 -> 4, 2 -> 3 } { 2 -> 3, 1 -> 2 }
```

While composites are possible, it must be noted that individual let-bound and lambda functions can only be printed in one way in a given program. This can cause problems with library code, because libraries are transformed once, which means the style of printing for each library function is permanently fixed.¹ A work-around is to re-define the function locally in user code, which can be as simple as wrapping the function in a let-binding, like so:

```
start = let m = map in m (compose inc inc) [1,2]
```

If a large number of functions need to be re-defined then it is more practical to copy the whole library module to the user's source tree, and transform it there. The downside with both solutions is that they require manual modification of the user's program, and it would be preferable to automate this in the debugger's interface. Currently all library modules are transformed so that functions use the intensional style. This is because the extensional style is generally more expensive in space and time than the intensional style. The reason is that the extensional style introduces more work for each function application, and it tends to retain references to more values, which prevents their garbage collection.

¹A similar issue occurs with trusted functions. Library functions are trusted by default, but sometimes the user might like to see the applications of library functions in the EDT.

6.6 Optimisation

The transformation of saturated function applications can be optimised by avoiding the use of encoded function representations. Where the optimisation applies, function values can inhabit their ordinary Haskell representation, allowing the built-in version of application to be used instead of `apply`. The benefit is an improvement in the execution time of transformed programs because some of the overheads involved with encoded functions are avoided. Saturated applications are common in practice and thus the optimisation is widely applicable.

The transformation of applications from Figure 5.8 (`ExpApp`) assumes that all functions are encoded. For saturated applications, intermediate encoded functions are created only to be immediately decoded and applied. The encoding serves no useful purpose in this case, and should be avoided. Consider an expression of the form ‘ $f\ E_1\ E_2$ ’, where f is a let-bound function of arity two. After transformation it becomes:

$$\text{apply} (\text{apply} (f\ i)\ \mathcal{E}\llbracket E_1 \rrbracket_i\ i)\ \mathcal{E}\llbracket E_2 \rrbracket_i\ i$$

There are two things to note. First, two encoded functions are constructed only to be immediately deconstructed by `apply`. Second, the three context arguments of f (which are denoted by i) are all the same. Ideally, the expression should be translated into something like the following which avoids the redundancies:

$$f\ i\ \mathcal{E}\llbracket E_1 \rrbracket_i\ \mathcal{E}\llbracket E_2 \rrbracket_i$$

f is applied to its arguments using normal function application, and it receives only one context value. This requires two versions of f : one which works in the general case of partial application, and one which is specialised for saturated application.

Figure 6.2 contains the optimised transformation rule for function declarations. Essentially the old rule for function bindings from Figure 5.7 (`FunBind`) is split in half, producing two functions instead of one. The first caters for saturated applications, and carries out the work of building EDT nodes. The second function provides

$$\begin{aligned}
\mathcal{D} \llbracket x \ y_1 \ \dots \ y_n = E \rrbracket &\Rightarrow \\
&\text{sx } c \ y_1 \ \dots \ y_n = \text{call } c \ "x" \ [\text{V } y_1, \ \dots, \ \text{V } y_n] \\
&\quad (\backslash i \rightarrow \mathcal{E} \llbracket E \rrbracket_i) \\
\text{px } c_0 &= \text{fun}_n (\backslash y_1 \ c_1 \ \dots \ y_n \ c_n \rightarrow \text{sx } c_{0/n} \ y_1 \ \dots \ y_n)
\end{aligned}$$

Figure 6.2: Optimised transformation of function declarations.

$$\begin{aligned}
\mathcal{D} \llbracket x^n :: T \rrbracket &\Rightarrow \text{sx} :: \text{Context} \rightarrow \mathcal{S} \llbracket T \rrbracket_n \\
&\Rightarrow \text{px} :: \text{Context} \rightarrow \mathcal{T} \llbracket T \rrbracket \\
\mathcal{S} \llbracket T_1 \rightarrow T_2 \rrbracket_0 &\Rightarrow \text{F } \mathcal{T} \llbracket T_1 \rrbracket \ \mathcal{T} \llbracket T_2 \rrbracket \\
\mathcal{S} \llbracket T_1 \rightarrow T_2 \rrbracket_{n+1} &\Rightarrow \mathcal{T} \llbracket T_1 \rrbracket \rightarrow \mathcal{S} \llbracket T_2 \rrbracket_n \\
\mathcal{S} \llbracket T \rrbracket_n &\Rightarrow \mathcal{T} \llbracket T \rrbracket \quad (T \neq T_1 \rightarrow T_2)
\end{aligned}$$

Figure 6.3: Optimised transformation of types.

an encoded interface which is suitable for partial applications, and is simply a thin wrapper around the first function. The two functions cannot have the same name, so a single letter prefix is added to distinguish them; **s** for the saturated variant and **p** for the partial variant. A renaming phase prior to transformation ensures that no name clashes are introduced.

The transformation rules for types and signatures are also modified. Figure 6.3 contains the new rules. The type of the partial variant is transformed in the same way as the original rule, whereas the type of the saturated variant is transformed in an arity dependent manner. The notation x^n means that the function x has an arity of n . If a function has n arguments then the first n function arrows in the spine of its type remain intact (in the partial variant all arrows are replaced by **F**). This is done by \mathcal{S} which takes an additional parameter which gives the arity of the function. There are two ways to determine a function's arity: by counting the parameters in its head, or by counting the number of function arrows in the spine of its type. The second count can be greater than the first because the type abstracts over the distinction between the head and body of a function definition.

For example consider this function:

```
f :: a -> b -> c -> b
f x = const
```

The first count gives an arity of one, and the second gives three. Arities are calculated in `buddha` in the first way, for two reasons. First, it accords with our concept of a redex, which is a term which matches the left-hand-side of a program equation. Second, because it can be done without type inference. This means that `f` is considered saturated when it has been applied to one argument, giving rise to these two type signatures:

```
sf :: Context -> a -> F b (F c b)
pf :: Context -> F a (F b (F c b))
```

A minor technical problem occurs because two different type class instances of a particular overloaded function can have different numbers of parameters in their heads. For example:

```
class Show a where
  show :: a -> String

instance Show Bool where
  show x = showBool x

instance Show Char where
  show = showChar
```

According to the arity calculation mentioned earlier, the `Bool` instance of `show` has an arity of one, but the `Char` instance has an arity of zero. In this situation `buddha` counts the arity of the function from the type scheme in the class declaration. Class instances which have a different number of parameters than the expected arity have arguments added or subtracted to their definition in order to rectify the mismatch. In the above example the definition of `show` for the `Char` instance will be modified to `'show x = showChar x'`.

Function applications are transformed to accommodate the optimisation as shown in Figure 6.4. The first rule transforms applications of let-bound identifiers with arity m to n arguments. If $m = n$ the application is saturated, which

$$\begin{aligned}
& \mathcal{E} \llbracket x^m E_1 \dots E_n \rrbracket_i \\
& \quad (m = n) \Rightarrow \mathbf{s}x \ i \ \mathcal{E} \llbracket E_1 \rrbracket_i \dots \mathcal{E} \llbracket E_n \rrbracket_i \\
& \quad (m > n) \Rightarrow \mathbf{apply} \dots (\mathbf{apply} \ (\mathbf{p}x \ i) \ \mathcal{E} \llbracket E_1 \rrbracket_i \ i) \dots \mathcal{E} \llbracket E_n \rrbracket_i \ i \\
& \quad (m < n) \Rightarrow \mathbf{apply} \dots (\mathbf{s}x \ i \ \mathcal{E} \llbracket E_1 \rrbracket_i \dots \mathcal{E} \llbracket E_m \rrbracket_i) \dots \mathcal{E} \llbracket E_n \rrbracket_i \ i \\
\\
& \mathcal{E} \llbracket k^m E_1 \dots E_n \rrbracket_i \\
& \quad (m = n) \Rightarrow k \ \mathcal{E} \llbracket E_1 \rrbracket_i \dots \mathcal{E} \llbracket E_n \rrbracket_i \\
& \quad (m > n) \Rightarrow \mathbf{apply} \dots (\mathbf{apply} \ (\mathbf{con}_i \ k) \ \mathcal{E} \llbracket E_1 \rrbracket_i \ i) \dots \mathcal{E} \llbracket E_n \rrbracket_i \ i \\
\\
& \mathcal{E} \llbracket E_1 E_2 \rrbracket_i \Rightarrow \mathbf{apply} \ \mathcal{E} \llbracket E_1 \rrbracket_i \ \mathcal{E} \llbracket E_2 \rrbracket_i \ i
\end{aligned}$$

Figure 6.4: Optimised transformation of function application.

means that the saturated variant of the function ($\mathbf{s}x$) can be called using normal function application. If $m > n$ the application is partial, which means that the partial variant of the function ($\mathbf{p}x$) must be called. The function is applied to each argument one-at-a-time and the intermediate encoded functions are decoded by \mathbf{apply} . If $m < n$ then the application is over-saturated. The saturated part is dealt with as earlier. The result of the saturated application is an encoded function, which must be applied to its arguments in the same way as partial applications. The second rule transforms applications of data-constructors with arity m to n arguments. This is much the same as the case for let-bound identifiers, except that data constructors cannot be over-saturated. The optimisation is especially effective in this case because data constructors are most often used in saturated contexts. Other types of function application can also be optimised, such as when the leftmost expression is a lambda function, but these cases are quite uncommon, and thus unlikely to be fruitful. Therefore all other applications types are transformed by the third rule which is identical to the original unoptimised approach.

6.7 Displaying large values

A major problem for all debugging tools is handling values which are too large to display all at once. Truncation is a cheap solution, but it does not always give the best results.

Often the correctness of a reduction will depend only on a relatively small sub-part of a value. In those cases it is much more useful to be able to focus on the relevant pieces, which requires a more fine-grained control of how values are dissected and printed. Various techniques are possible. Examples include:

1. An interactive term browser, which allows the user to explore values in a fashion similar to the way hierarchical file-systems are used on modern operating systems.
2. A term query language, akin to a query language in a relational database.
3. Custom printing routines, which are crafted for particular types of values, producing whatever syntax is most convenient for the user, and filtering out parts which are not of interest.

All of this ties into the general field of usability, which, though very important, has not been the primary focus of this thesis. Future work on `buddha` will certainly address these issues.

Another interesting idea, which seems to have originated in Nilsson and Fritzson [1993], is to use type information to simplify the display of reductions involving polymorphic functions. For example, `list reverse` has this type scheme:

```
reverse :: [a] -> [a]
```

`reverse` does not depend on the actual values in the list, only their order, therefore it is possible to replace the values with labels. For instance this reduction:

```
reverse ["Fred", "Ned", "Barry"] => ["Barry", "Ned", "Fred"]
```

can be abbreviated like so:

```
reverse [e1, e2, e3] => [e3, e2, e1]
```

which is clearly much simpler.

Unfortunately this idea does not have a big effect in practice. The reason is that it is limited to parametric polymorphism. This kind of polymorphic function tends to be small and of general utility, and as such it is most common in libraries. However, library code is usually well tested, and thus trusted by the debugger. Hence there are typically few nodes in the EDT for parametric polymorphic functions. An interesting question for future research is whether something similar can be done for overloaded functions, which is the type of polymorphism that occurs more frequently in user code.



Chapter 7

Practical Considerations

Programs are complex artifacts, and debugging is often a struggle against complexity. Why should the programmer have to face it alone? Let's make the computer take an active role in helping the programmer deal with complexity.

The Debugging Scandal and What to do About It

[Lieberman, 1997]

7.1 Introduction



WHILE the theory of declarative debugging is attractive and fairly well known, the practice of applying it to real programs remains difficult, especially for lazy functional languages.

This chapter addresses two problems that have hindered earlier efforts: I/O, and resource intensive computations.

The inherently stateful nature of I/O means that the relative order of operations is central to the question of program correctness. For this reason Haskell has evolved an imperative style to cope with the demands of I/O, based on monads (see Section 2.4). It is not surprising then that programmers tend to adopt an imperative mode of thought when developing the I/O parts of their code. Can we apply

declarative debugging to I/O? We can, provided that we can print the *effects* of I/O functions in a fashion which is meaningful to the programmer. The extensional style of printing higher-order values provides a very simple solution to this problem.

Long running computations are difficult to debug, regardless of the programming language, simply because the accumulated history of individual execution events can grow to sizes which no human can feasibly digest. The post-mortem nature of declarative debugging introduces scalability problems because the size of the EDT grows proportionally to the running time of the debuggee. It is not difficult to find program runs for which the size of the EDT far outstrips the available memory on even the most powerful of modern machines. The most obvious solution is to materialise only some parts of the whole EDT at any one time. In this chapter we explore two ways to reduce the size of the EDT. First, we only record nodes for suspicious functions, and second, we employ a piecemeal approach to EDT construction. Both ideas are well established in the literature; our contribution is to show how they can be achieved in a debugger based on program transformation.

7.1.1 Outline of this chapter

The rest of this chapter proceeds as follows. In Section 7.2 we consider debugging the I/O parts of programs, with special emphasis on the printing values of the `IO` type. We also consider the relationship between the structure of the EDT and code which arises from the use of `do`-notation when used for the `IO` type. Haskell 98 style exceptions are briefly discussed. At the end of the section the performance of our `IO` implementation is evaluated. In Section 7.3 we show how the performance of the debugger can be improved by optimising the treatment of trusted function applications. In Section 7.4 we consider piecemeal EDT construction, which keeps the memory requirements of the debugger in check by building only part of the EDT at a time. First a very simple scheme is described, which is easy to implement. The performance of this approach is measured on five programs, then various improvements are discussed. In Section 7.5 we discuss the status of the implementation.

7.2 I/O

7.2.1 Printing IO values

Incorporating IO functions in the debugger is challenging because it is not obvious how to print values of the IO type. Merely printing the internal representation used by the compiler is unlikely to be fruitful because IO is an abstract data type, which means that the programmer is not (supposed to be) aware of its representation. Nonetheless, programmers *do* have intended interpretations for IO functions, which can be stated declaratively. The basic idea is that the result of an IO function can be viewed as the combination of the returned value plus a sequence of side-effects. Our goal is to find a way to print IO values in a fashion that contains all that information.

Consider the following buggy code. It is supposed to prompt the user with a question, return `True` if they input `'t'`, return `False` if they input `'f'`, and repeat the question in a loop if they input anything else:

```
ask :: String -> IO Bool
ask question
  = do putStr question
       response <- getChar
       act response question

act :: Char -> String -> IO Bool
act response question
  = case response of
      't' -> return True
      'f' -> return True
      other -> do putStr "\nPlease enter t or f\n"
                  ask question
```

Suppose that `ask` is called with the question *“Do you know the way to Santa Fe?”*, resulting in the following interaction with the user (whose responses are typed in italics):

```
...
Do you know the way to Santa Fe? N
Please enter t or f
Do you know the way to Santa Fe? f
...
```

Owing to the bug, `ask` returns `True` when it should have returned `False`. Now consider judging the correctness of the first call to `ask`; how should the reduction be shown? The following reduction is insufficient:

```
ask "Do you know the way to Santa Fe?" => True
```

The correctness of the call depends on the interaction with the user, but that information is missing.

The most common way of implementing IO (in current Haskell compilers) is as a state-of-the-world transformer function, like so:

```
type World = ...
newtype IO a = IO (World -> (World, a))

instance Monad IO where
  return x = IO (\w -> (w, x))

  IO f >>= next
    = IO (\w1 -> case f w1 of
                  (w2, v) -> case next v of
                              IO g -> g w2)
```

The `World` type is merely a token representing the state of the real world; its value is of no particular significance. At first sight it would appear that this implementation of IO is unsuitable for printing because there is no mention of the actual effects, but this can be rectified by using the extensional style, as the following discussion demonstrates.

The key idea is to turn the `World` into a counter, which is incremented each time a primitive IO operation is performed:

```
type World = Natural -- unsigned integer
```

The link between IO values and the world counter becomes clear when the state transformer function is printed in the extensional style. For instance, the reduction for the first call to `ask` might be printed like so:

```
ask "Do you know ... Santa Fe?" => { 11 -> (16, True) }
```


This means that before the call the world counter was 11, and after the call it was

16. Five effects happened within the call, namely:

```
12: putStr "Do you know the way to Santa Fe?"
13: 'N' <- getChar
14: putStr "\nPlease enter t or f\n"
15: putStr "Do you know the way to Santa Fe?"
16: 'f' <- getChar
```

The effects include those that arise directly from the body of `ask` and also those that arise from the functions it calls.

When an `IO` value is printed, the user must be able to discover exactly what effect happened at each world increment. This is done by caching all the primitive effects in an array which is indexed by the `World` counter. The user can query the array to discover individual effects, or sequences of effects, like so:

```
buddha> show io 12-16
```

which prints a list of the effects numbered 12 to 16, similar to that shown above. Each element in the array records an *event*, which is a pair containing the action that was performed and its return value:

```
type IOEvent = (IOAction, Value)

data IOAction
  = PutStr String
  | GetChar
  ...

ioTable :: IOArray World IOEvent
recordIOEvent :: IOEvent -> IO ()
```

From the example above, the element at index 16 in the array would contain the entry: `(GetChar, V 'f')`. `IOArray` is an array of mutable elements available in the `IO` monad. The maximum number of `IO` effects produced by a program is normally not known in advance, so the array is dynamically resized, by doubling its previous size whenever it gets full.

The program transformation for debugging must be applied to the whole program, including the `IO` type. Since the argument of the `IO` constructor is a function,

it must be transformed into the encoded form of functions using the `F` type constructor:

```
newtype IO a = IO (F World (World, a))
```

The `IO` primitives need special support because they are not implemented in Haskell. Rather than write our own set of primitives, it is much simpler to reuse those provided by the compiler. This requires an interface between the `IO` type in `buddha` and the `IO` type of the compiler. Since this interface is needed for every primitive, it also makes the perfect place to update the state counter:

```
performIO :: IOAction -> World -> Prelude.IO a -> (World, a)
performIO action world io
  = seq nextWorld $ unsafePerformIO $
      do val <- io
         recordIOEvent (action, V val)
         return (nextWorld, val)
  where
    nextWorld = world + 1

threadIO :: IOAction -> Prelude.IO a -> Buddha.IO a
threadIO action io
  = IO (funE1 (\world _ctxt -> performIO action world io))

primGetChar :: Context -> Buddha.IO Char
primGetChar context = threadIO GetChar Prelude.getChar
```

`threadIO` turns the standard `IO` type (`Prelude.IO`) into `buddha`'s `IO` type, and via `performIO`, updates the state counter and records the effect in the `IOEvent` array. Note the use of `funE1` to produce an extensional encoding of the function inside the `IO` type. `primGetChar` illustrates the way that individual primitives are implemented. It takes a context argument (which it ignores) to give it an interface which can be called from transformed code.

The monad class instance for `IO` is transformed in the usual way, with a proviso that the state transforming functions (the lambda abstractions underneath the `IO` constructor) use the extensional style. `Do` notation is desugared before program transformation, which makes the calls to `>>=` explicit.

7.2.2 EDT dependencies for IO code

An important consideration is the shape of the EDT. There is a potential discrepancy between the dependencies suggested by the `do`-notation, and those that arise from the desugared version of the code. In the above example, the `do`-notation suggests that there is a direct dependency between `ask` and `act`. However, in the desugared code below, the link is interrupted by the introduction of `>>=` and nested lambda abstractions:

```
ask question
  = (putStr question) >>=
    (\_ ->
      getChar >>=
        (\response ->
          act response question))
```

The issue is easier to discuss if all the lambda abstractions in `ask` and `>>=` are given names, and the different uses of `>>=` are named apart:

```
ask question
  = (putStr question) >>=_a lamAsk1
  where
    lamAsk1 _ = getChar >>=_b lamAsk2
    lamAsk2 response = act response question

IO f >>= next
  = IO lamBind
  where
    lamBind w1 = case f w1 of
      (w2, v) -> case next v of
        IO g -> g w2
```

It is important to note that we could transform any individual function in the above code (including the lambda abstractions) in either the intensional or extensional style. The most important question is what is the link, if any, between `ask` and `act`, when different styles of transformation are used? Ultimately it comes down to how each of `lamAsk1`, `lamAsk2`, and `lamBind` determine their parents.

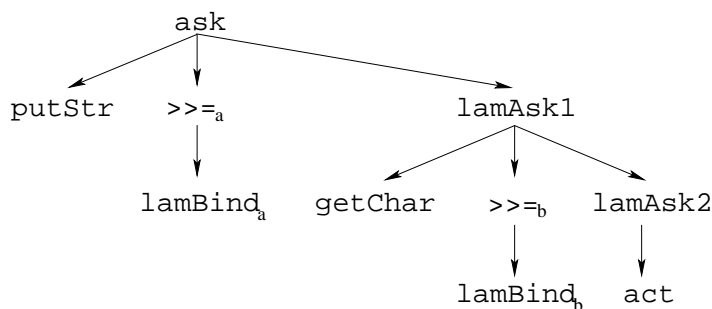


Figure 7.1: IO dependencies (extensional).

For the purpose of illustration we consider three different cases:

1. All functions are transformed using the extensional style.
2. All functions are transformed using the intensional style.
3. All functions are transformed using the intensional style, except `lamBind`, which is transformed using the extensional style (as it is done in the current version of `buddha`).

Figure 7.1 illustrates the dependencies that arise in the first case. All let-bound functions determine their parent at the place where they are first mentioned. Here `act` is a descendant of `ask`, which reflects the kind of relationship suggested by the `do`-notation. As an aside, in practice, the lambda abstractions would be trusted functions, therefore it would appear to the user as if `act` was a child of `ask`. It is also worth noting that in this situation the kind of transformation applied to `>>=` and `lamBind` has no effect on the relationship between `ask` and `act`.

Figure 7.2 illustrates the dependencies that arise in the second case. All let bound functions determine their parent at the place where they become fully saturated redexes. In this situation the EDT appears to be rather disjoint, because the calls to `lamBind` are no longer children of the respective calls to `>>=`. This means that the sub-trees for each `lamBind` are lifted out of the sub-tree for `ask`. Therefore `act` is no longer a descendent of `ask`.

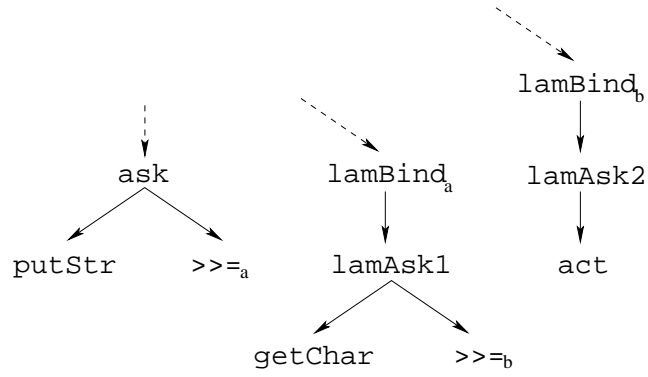


Figure 7.2: IO dependencies (intensional).

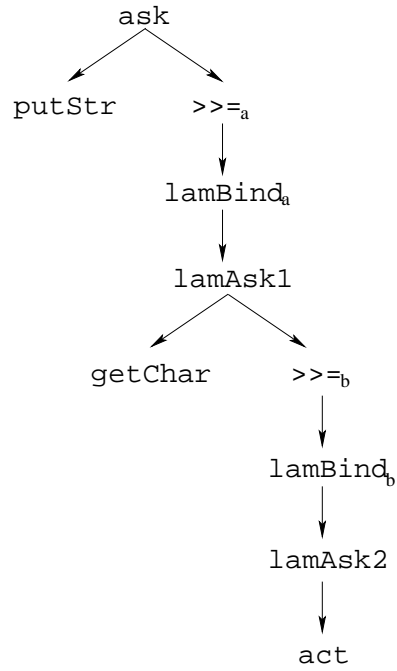


Figure 7.3: IO dependencies (intensional, except lamBind).

Figure 7.3 illustrates the dependencies that arise in the third case. Compare this EDT with the one in Figure 7.2. Note that it is the position of `lamBind` which ensures that `act` is a descendent of `ask`.

Which EDT is preferable? It is difficult to say that one EDT is always better than the others because the monadic style permits at least two different views of a piece of code. The “low-level” view includes the plumbing machinery inside the definition of `>>=`, and the “high-level” view abstracts over this. Printing `IO` values (*i.e.* `lamBind`) in the intensional style reflects the low-level view because it requires the user to be aware of the internal workings of `>>=`. The extensional style reflects the high-level view because `IO` values are shown as abstract mappings over states of the world (with no internal structure). Most of the time the high-level view is desirable, since we tend to think of a monad as a mini domain specific language, which provides an interface to a special computational feature. The `do`-notation is the syntax of the new language. In most cases the best EDT is the one which reflects the dependencies suggested by that syntax, rather than its desugaring, hence in `buddha` we transform `IO` values to reflect the high-level view.

7.2.3 Exceptions

Haskell 98 supports a limited kind of exception handling mechanism for errors involving `IO` primitives. The `IO` type above must be extended slightly to accommodate this feature. The standard library provides the `IOError` data type which encodes various kinds of errors that can happen when an `IO` operation is performed, for example attempting to open a file which does not exist:

```
data IOError = ...
```

Programmers can trap these errors with the function `catch`, but only in the `IO` monad:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

The first argument is an `IO` computation to perform, and the second argument is an exception handler. The value of the expression `'catch io handler'` is equal to

‘handler *e*’, if *io* raises the exception *e*, otherwise it is equal to *io*.

We extend the `IO` type slightly to encode the possibility of an exception in the result:

```
data Either a b = Left a | Right b
newtype IO a = IO (World -> (World, Either IOError a))
```

which means that `catch` can then be implemented in ordinary Haskell code in a fairly obvious way. The only remaining problem is how to transfer exceptions from the built-in version of `IO` to `buddha`’s version of `IO`. `IOErrors` can only be raised by one of the built-in primitive functions. Thus, all potential exceptions can be caught using the built-in `catch` at the point where the primitives are called within `performIO`:

```
performIO action world io
  = seq nextWorld $ unsafePerformIO $
      do x <- try io
      case x of
        Left e -> do recordIOEvent (action, V e)
                     return (nextWorld, Left e)
        Right val
          -> do recordIOEvent (action, V val)
               return (nextWorld, Right val)

  where
    nextWorld = world + 1

try io
  = Prelude.catch (do { v <- io; return (Right v) })
    (\e -> return (Left e))
```

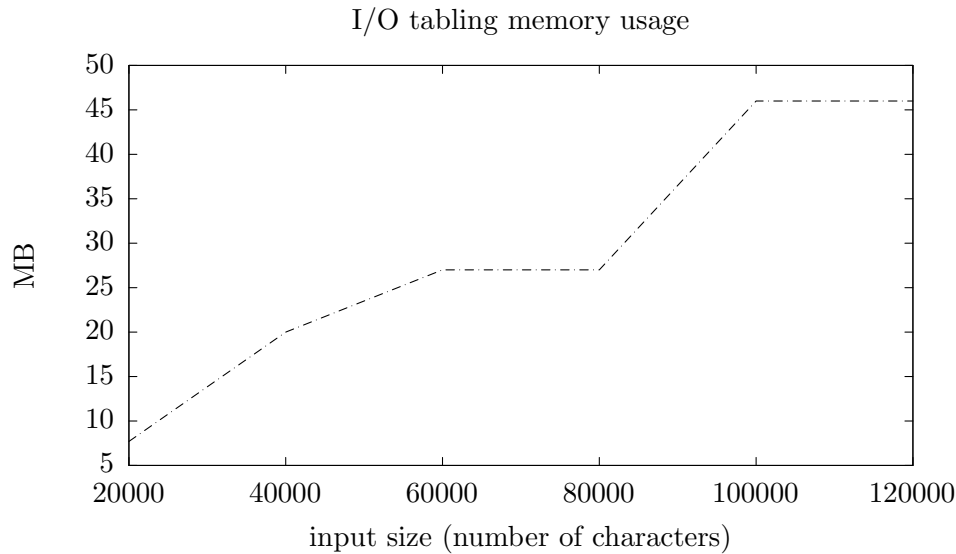


Figure 7.4: Memory usage versus input size with I/O tabling enabled.

7.2.4 Performance

The performance cost of tabling primitive I/O actions can be gauged with the following simple program:

```
main = loop 0

loop i = do
  getChar
  print i
  loop (i+1)
```

The program implements a tight loop, which repeatedly reads a character from the standard input and prints an integer to the standard output. The program stops (and raises an exception) when the input is exhausted (*i.e.* when the end of file is encountered on Unix). We transformed the program for debugging and measured its space and time behaviour for several large input sizes. To measure just the cost of tabling (as closely as possible) we modified the debugger so that the EDT was not constructed.

Figure 7.4 plots the memory usage of the transformed program versus the number

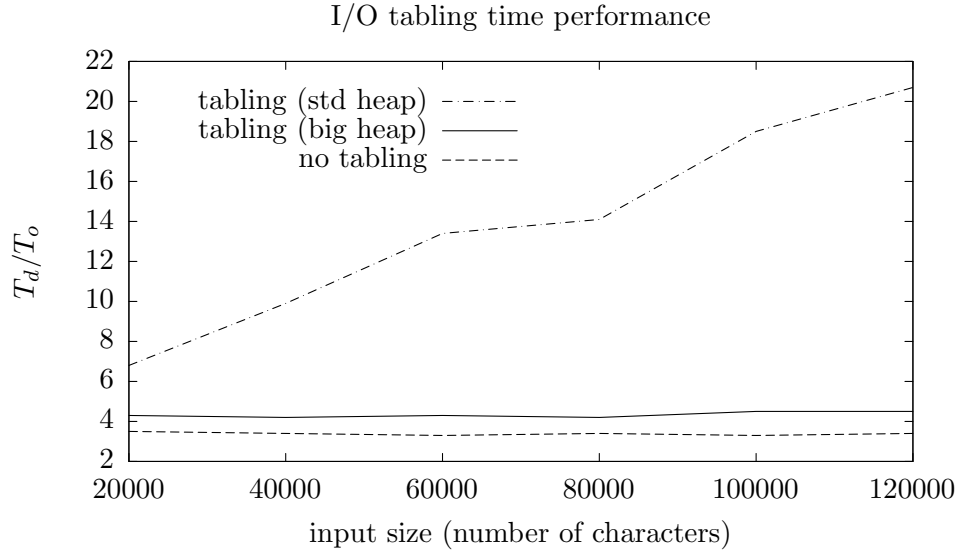


Figure 7.5: Running time with I/O tabling enabled relative to the original program.

of characters in the input (ranging in size from 20,000 to 120,000). As expected the memory usage grows linearly with the size of the input, though the growth is stepped owing to the fact that the array which stores the I/O events is doubled in size each time it is extended. For each input character the program performs three primitive I/O actions: a `getChar` to read the character, a `putStr`¹ to print the integer, and a `putChar` to print a newline (`print` calls `putStrLn`, which in turn calls `putStr` and `putChar`). Therefore the gradient of the graph suggests that in this example each tabled I/O action requires about 130 bytes of memory on average.

Figure 7.5 plots the time usage of the transformed program versus the number of characters in the input (using the same data as before). The time is measured relative to the running time of the original untransformed program on the same input. Three lines are drawn on the graph. The bottom line (dashed) shows the performance of the program where I/O tabling is disabled (that is, the program is transformed for debugging but I/O events are not recorded in the table). The top

¹In *buddha* `putStr` is treated as a primitive I/O function even though it could be implemented in terms of `putChar`. It is generally preferable for the user if they have to consider just one `putStr` event instead of many separate `putChar` events.

line (dot-dashed) shows the performance of the program with IO tabling enabled. There is a substantial difference between this line and the bottom one, and worse still, the relative cost keeps growing with the size of the input.

The reason for this poor performance is that IO tabling uses a mutable data structure which triggers a well-known problem with GHC's garbage collector. GHC employs a generational collector, where long lived data values are scanned less frequently than younger values. The heap is divided into at least two partitions, called generations. For the sake of this discussion we will assume there are just two, which is the default anyway. The old generation contains values which are retained in the heap for relatively long periods of time. The young generation contains newly constructed values. An object is moved from the young generation to the old generation if it stays alive for some threshold period of time. The garbage collector scans the heap in two ways. A *minor* collection just looks at the young generation of values, whereas a *major* collection looks at both generations. Minor collections happen more often than major ones because scanning values which are not garbage is wasted work, and values in the young generation are more likely to be garbage than values in the old generation. In a system with only immutable values, elements of the young generation can refer to elements of the old generation, but not *vice versa*. So if a minor collection finds that there are no references to an object in the young generation then it is safe to say that it is garbage: there is no need to check for references to that value in the old generation. However, mutable values allow values of the old generation to refer to values of the young generation. To work around this problem, GHC's collector (up until version 6.4 of the compiler) scans all mutable values at each minor collection. This means that the IO table is scanned at every minor collection, despite the fact that it never contains any garbage. This results in very poor performance. As the table gets bigger the scanning time gets longer.

We can work around this problem by setting the minimum heap size used by the program to a large value. This causes fewer minor collections, which reduces

the number of times the IO table is traversed. The middle line in Figure 7.5 (solid) shows the performance of the program with IO tabling enabled and a minimum heap size of 100MB. In this case the program is about four times slower than the original, regardless of the size of the input, which is not much worse than the case where IO tabling is disabled.

It is difficult to make definitive conclusions from this one example, but it seems that the overheads incurred by IO tabling are within reasonable limits, at least when a large minimum heap is used. Our experience is that debugging with IO tabling enabled is quite acceptable for program runs which are not particularly IO intensive.

7.3 Trusted functions

In most debugging sessions the programmer will have a fair idea which functions are likely to be buggy, and which are not:

- It is possible to rule out parts of the program which are executed after the bug has been observed in the output.
- The results of testing — especially unit tests — may also provide useful information about where to hunt for bugs.
- Some functions may be considered correct, perhaps by proof, or by rigorous testing; for instance library functions.

Declarative debuggers can capitalise on this information by pruning nodes from the EDT which correspond to applications of trusted functions. The benefits of fewer nodes in the EDT are twofold: fewer judgements are required from the oracle, and the EDT consumes less space. Trusting is particularly attractive because it is both simple to implement and very effective. Figure 7.6 lists the percentage of all EDT nodes which arise from functions in trusted standard libraries in the suite of five non-trivial sample programs which were first mentioned in Chapter 5. It is clear, at least in these examples, that big savings can be had by optimising the behaviour of the debugger for trusted functions.

<i>program</i>	<i>trusted nodes</i>
Pancito	73 %
Cacheprof	89 %
Raytrace	96 %
Prolog	64 %
Mate	73 %

Figure 7.6: Percentage of nodes which come from functions in trusted standard libraries.

Which nodes should be removed from a sub-tree whose root contains a trusted application? Various approaches are possible, for example:

- Prune the whole sub-tree.
- Prune only the root node. Suspicious descendents of the node become children of the root node's most recent suspicious ancestor.
- Prune nodes after a certain depth has been reached.
- Only prune nodes whose arguments match certain requirements.

The first two are the most commonly implemented in practice. The second option is useful in cases where a higher-order function is trusted, but it is passed a suspicious function as an argument, which is subsequently applied and reduced inside its body (although this is only relevant when the higher-order argument is printed in intensional style).

Buddha employs the second option. Each let-bound function can be declared to be either trusted or suspicious. Suspicious functions are transformed as before, but trusted functions are transformed so that they do not build EDT nodes. Instead, a trusted function passes its parent context directly onto the functions which are called in its body. Therefore, a suspicious function call inserts its node into the context from its most recent suspicious ancestor, which may be the parent of the node, or its grand-parent, or great-grand-parent *etcetera*.

Fortunately it is easy to change the transformation rules to support trusted functions. In Chapter 5 we employed the `call` function to build EDT nodes (see

Figure 5.2). For trusted functions `call` is replaced by a new function, `trustCall`, which is defined as follows:

```
trustCall :: Context -> (Context -> a) -> a
trustCall context body = body context
```

The first parameter, `context`, represents a reference to the call's closest suspicious ancestor. The second parameter, `body`, is the transformed version of the function's body. It is clear that the parent context is simply passed on to all the children of the call. Trusted pattern bindings are handled in a similar way.

The confidence value of each function is set at the time of program transformation. There are two ways that a user can declare the confidence values. First, they can issue a command line argument, `'-t trust'` or `'-t suspect'`, which sets the confidence for every function in a given module. Second, they can provide an options file for a given module, which sets the value for individual functions. The command line overrides anything which is set in an options file.

7.4 Piecemeal EDT construction

The biggest impediment to implementing a practical declarative debugger is the space consumed by the EDT. Space is used in two ways: first, for the EDT nodes themselves, and second, for the argument and result values pointed to by the nodes. In a naive implementation the whole EDT is constructed by a single execution of the debuggee. The EDT is not traversed until the debuggee has terminated which means that the whole tree must be retained in memory. The argument and result values referred to by the EDT cannot be garbage collected as they would have been in the original program. Thus the space usage of a naive debugger grows proportionally to the running time of the program. Only the shortest of program runs can be debugged this way on a modern machine.

This is a well known problem, and various solutions have been proposed in the literature:

- Trusting: as discussed in the previous section.
- Partial EDT construction: when the size of the EDT becomes too big, stop collecting nodes [Sparud, 1999].
- Saving to disk: write the whole EDT (or program trace) to the much larger (but slower) hard disk drive [Chitil et al., 2002].
- Piecemeal EDT construction: only a small sub-tree of the EDT is kept at any time. Debugging proceeds with the sub-tree as normal. If a pruned leaf node is reached, its own sub-tree is re-generated by re-executing (perhaps part of) the program again [Naish and Barbour, 1996, Nilsson, 1998, Pope and Naish, 2003b, MacLarty and Somogyi, 2006].

Trusting is not a complete solution on its own, because the number of nodes generated by suspicious functions can still be prohibitively large; therefore it must be accompanied by another scheme.

Partial EDT construction is a fairly cheap solution, but pruned sub-trees may contain buggy function applications, which reduces the accuracy of the diagnosis.

Saving to disk is motivated by economics. Per megabyte, disk drives are roughly two orders of magnitude cheaper than RAM (chip memory). Consequently, disk drive capacities on typical personal computer workstations are about two orders of magnitude larger than their RAM capacities. However, the free space available to an individual user may be much less, perhaps only one order of magnitude larger. Saving to disk does not solve the space problem, but simply moves it to a cheaper device. An advantage of saving to disk is that the EDT can be reused many times without having to execute the debuggee again; you can even run the debuggee on one computer and debug on another. The main disadvantage of saving to disk is that RAM is much faster; the access time of RAM is about five orders of magnitude faster

than disk, and the bandwidth of RAM is about one-to-three orders of magnitude faster than disk.²

Piecemeal generation materialises only part of the EDT at a time, and re-generates the missing parts on demand by re-executing the program: a classic space/time tradeoff. This section shows how a fairly simple version of piecemeal EDT construction can be added to `buddha`, and discusses various ways in which it can be improved.

7.4.1 The basic idea

Initially the debuggee is executed and the EDT is collected with a root node corresponding to the evaluation of `main`. The EDT is materialised up to a predetermined depth from the root. Nodes which form the fringe of this initial EDT contain a flag which tells the debugger their sub-trees have not been constructed. Nodes which would normally be deeper than the fringe are not constructed at all. Debugging proceeds with this initial tree. If a buggy node is located, the debugging session is complete. If a fringe node is encountered, its sub-tree is materialised up to a certain depth by running the program again. Debugging then resumes with the new sub-tree. The process continues along the same lines until the debugging session is finished.

7.4.2 Problems

Running a stateful fragment of code multiple times may not always produce the same behaviour, unless the state is re-set before each run. At first sight it would seem that re-executing parts of a purely functional program should be straightforward, because there is no implicit state. Unfortunately, this is not entirely true, principally because the debugger can observe the (stateful) effects of computation which are not visible at the Haskell level.

²These are very approximate figures based on the peak performance of commodity hardware components.

There are three places where state is observed by the debugger: lazy evaluation, CAF evaluation, and I/O.

To materialise the sub-tree at a fringe node, it is necessary to re-execute the part of the program that produces nodes in that tree. Intuitively it should be possible to save a copy of the call in the fringe node and re-execute it when the node is encountered. Lazy evaluation makes this difficult because the extent to which an expression is evaluated is context dependent. The result of a call might have only been partially computed in the original program run. A safe re-execution should only demand the result to the same extent. To do this we would need to somehow reconstruct the same level of demand as the original call, but that information is not necessarily locally available, and may depend on an intricate chain of calls spread across many parts of the EDT. In earlier work [Pope and Naish, 2003b] we tried to solve this problem by using the result of the original call as an indicator of the level of demand. Given a copy of the function and its arguments, a new application can be constructed and called, such that its result is evaluated up to, but not exceeding its previous state. There are a couple of problems with this solution: the result is not necessarily an accurate reflection of how much work is done at a node in the EDT because a large part of the result can come from argument values, and the implementation is difficult without extensive support from the runtime environment. The implementation described in this section avoids the issue by re-executing the program each time from the beginning, just like Freya [Nilsson, 1998]. This has an obvious performance penalty because, in most cases, re-execution performs more work than necessary. In future work we will revisit this problem, perhaps by using the sequential numbering of nodes as a means to measure and control how much work is needed to re-generate a sub-tree.

The value of a CAF is saved after the first time it is evaluated. Subsequent references to the CAF get the value immediately without causing the CAF to be evaluated again. All nodes in the EDT appear underneath `main`, which is a CAF. To re-execute the program it is not sufficient to merely call `main` again, since its result is

saved. Therefore it is necessary to revert `main` (and all other CAFs) to their initial state before the program can be re-executed. This cannot be done from within Haskell, so it requires special support from the runtime environment. With GHC this can be achieved with dynamic loading via a tool called `hs-plugins` [Pang et al., 2004]. The object code of the debuggee (including library imports) is dynamically loaded into the debugging executable. Just before re-executing the debuggee its object code is re-loaded, which sets all CAFs back to their unevaluated state.

Re-executing a program requires all parts of the computation to be run again, including I/O operations. However, many I/O operations cannot be safely re-executed, simply because they change the state of the world in irrevocable ways. It is not feasible to re-set the state of the world back to its original value before running the program again; that would require a time machine! Somogyi [2003] solves this problem in the declarative debugger for Mercury by making primitive I/O operations idempotent. In the initial run of the program all I/O primitives are numbered and their results are saved in a global array. When the program is re-executed, I/O primitives retrieve their previous result from the table instead of firing another side-effect. The same solution is employed in `buddha` using the `ioTable`, introduced in Section 7.2.1 for printing IO values.

7.4.3 Implementation

A new type of EDT node is introduced for fringe nodes:

```
data EDT
  = EDT ...
  | Fringe { nodeID :: NodeID }
```

It contains a unique identity just like regular nodes.

The wrong answer diagnosis algorithm from Figure 3.4 is extended with a new equation to handle the new type of node:

```

debug diagnosis [] = ...                                -- as before

debug diagnosis (Fringe id : siblings)
  = do newNode <- materialise id
      debug diagnosis (newNode : siblings)

debug diagnosis (node : siblings) = ...                -- as before

materialise :: NodeID -> IO EDT
materialise node = ...

```

`materialise` re-executes the program and collects nodes from the sub-tree whose root node has identity `id`. The materialised sub-tree is hereafter called the *target*. Debugging resumes as usual once the target has been constructed.

The most complex part of the implementation is deciding which nodes to keep when the debuggee is executed. Each node in the EDT is uniquely identified by a number, and each node gets the same identity each time the program is run (assuming the program is executed deterministically). A global variable called `rootID` records the identity of the root node for the target:

```
rootID :: IORef NodeID
```

The value of `rootID` is set by `materialise` before the debuggee is executed.

Nodes are materialised if they match one of two conditions:

1. The identity of the node is equal to `rootID`. This node is called the *target root*.
2. The node is a descendent of the target root, within the depth bound.

Fringe nodes are created when a descendent of the target root appears at a depth equal to the depth bound.

The `Context` type is extended to indicate whether a node's context is inside or outside the target:

```
type Depth = Natural    -- unsigned integer

data Context
  = Pre
  | In Depth (IORef [EDT])
  | Post
```

`Pre` means the target root has not been found but it might be found in a subtree of the node. `In` means the target root has been found and the node is within the target. `Post` means that the target root has been found and the node is outside the target. Contexts change from `Pre` to `In` at the target root, and from `In` to `Post` at the fringe. The first argument of `In` records how deep the node is from the target root, and the second argument is a mutable list of sibling nodes.

Another global variable, called `top`, records the target root:

```
top :: IORef [EDT]
```

There is only ever one target root, but the code is somewhat simpler if `top` is treated as a list of sibling nodes. `materialise` passes the first (and only) node in this list back to the debugger when re-execution is complete.

As before, nodes are constructed by `call` (see Section 5.4). Its definition is modified to support selective materialisation, by splitting it into three equations, one for each kind of context.

The first equation of `call` handles `Pre` contexts:

```
call Pre name args body
  = unsafePerformIO $ do
    id <- nextCounter
    start <- readIORef rootID
    if start == id
      -- this is the target root node, depth = 1
      then createNode 1 id name args top body
      -- this is not the target root
      else return (body Pre)
```

If the node is the target root then it is materialised and inserted into `top`, otherwise the `Pre` context is propagated downwards to its children.

Nodes are constructed by `createNode`, in a similar fashion to the original version of `call`:

```
createNode :: Depth -> NodeID -> String -> [Value] ->
            IORef [EDT] -> (Context -> a) -> IO a
createNode depth id name args ref body
  = do children <- newIORef []
      let result = body (In depth children)
      let node = EDT { ... } -- as in the original version of call
      insertInSiblings node ref
      return result
```

The main difference is that it takes a depth value as an argument, and it supplies an `In` context to the body of the function (as indicated by the underline).

The second equation of `call` handles the `In` context:

```
call (In d ref) name args body
  = unsafePerformIO $ do
      id <- nextCounter
      if d == maxDepth
      then fringeNode id ref body
      else createNode (d+1) id name args ref body
```

If the depth of the node, denoted by `d`, is equal to the depth threshold, a fringe node is created, otherwise the node is materialised at depth ‘`d+1`’, and inserted into its list of sibling nodes. Fringe nodes are created like so:

```
fringeNode :: NodeID -> IORef [EDT] -> (Context -> a) -> IO a
fringeNode id ref body
  = do let result = body Post
      let node = Fringe { nodeID = id }
      insertInSiblings node ref
      return result
```

The children of a fringe node are passed a `Post` context indicating they are beyond the depth bound.

The third equation of `call` handles the `Post` context:

```
call Post name args body
  = unsafePerformIO $ do
    nextCounter
    else return (body Post)
```

`Post` contexts are simply propagated downwards, which makes them the cheapest context to handle. Note that it is still necessary to update the global counter to ensure that all nodes get the same identity regardless of whether they are materialised or not.

The scheme described so far works for nodes which correspond to function applications but there is a problem with CAFs. Nodes for CAFs are created once and shared between multiple references (see Section 5.5). The body of a CAF must not depend on the value of its context argument — if it did, sharing would be lost. As a consequence, when a CAF is evaluated it is not known whether it will be a descendent of the target root, and it is not known at what depths it will appear in the EDT. Therefore the method of passing depth information down through context arguments does not work for CAFs.

There are numerous ways to tackle this problem. We adopt a simple technique that appears to work well in practice. All CAFs appear at depth zero. When a CAF is evaluated, its sub-tree is speculatively materialised up to the depth bound. If the CAF is a child of some node within the target then its EDT is retained, otherwise it is garbage collected.

A fourth kind of context is added for speculative EDT construction:

```
data Context
  = ...
  | Speculate Depth (IORef [EDT])
```

`Speculate` contexts are only generated by CAFs. Nodes for CAFs are constructed by `constant` as before (see Section 5.5):

```
constant :: String -> (Context -> a) -> (Context -> a)
constant name body = ref (valueAndNode name body)
```

Below, the definitions of `valueAndNode` and `ref` are modified to suit the selective materialisation of nodes. Children of a CAF node start at depth one. If the node is the target root, it is inserted into `top`:

```
valueAndNode :: String -> (Context -> a) -> (a, EDT)
valueAndNode name body
  = unsafePerformIO $ do
    children <- newIORef []
    id <- nextCounter
    start <- readIORef rootID
    if id == start
      -- it is the target root
      then do let result = body (In 1 children)
              let node = EDT { ... } -- as before
              insertInSiblings node top
              return (result, node)
      -- it is not the target root
      else do let result = body (Speculate 1 children)
              let node = EDT { ... } -- as before
              return (result, node)
```

The node for a CAF is only inserted into the rest of the EDT when it is referred to from an `In` or `Speculate` context:

```
ref :: (a, EDT) -> Context -> a
ref (value, node) (In d ref)
  = unsafePerformIO $ do
    insertInSiblings node ref
    return value
ref (value, node) (Speculate d ref)
  = unsafePerformIO $ do
    insertInSiblings node ref
    return value
ref (value, node) other = value
```

The definition of `call` must be updated to support `Speculate` contexts. They are handled in much the same way as `In` contexts, except that when the fringe is encountered, the children nodes are passed a `Pre` context instead of an `Out` context. This is because the target root can be a descendent of a speculated CAF, and it might be found outside the depth bound.

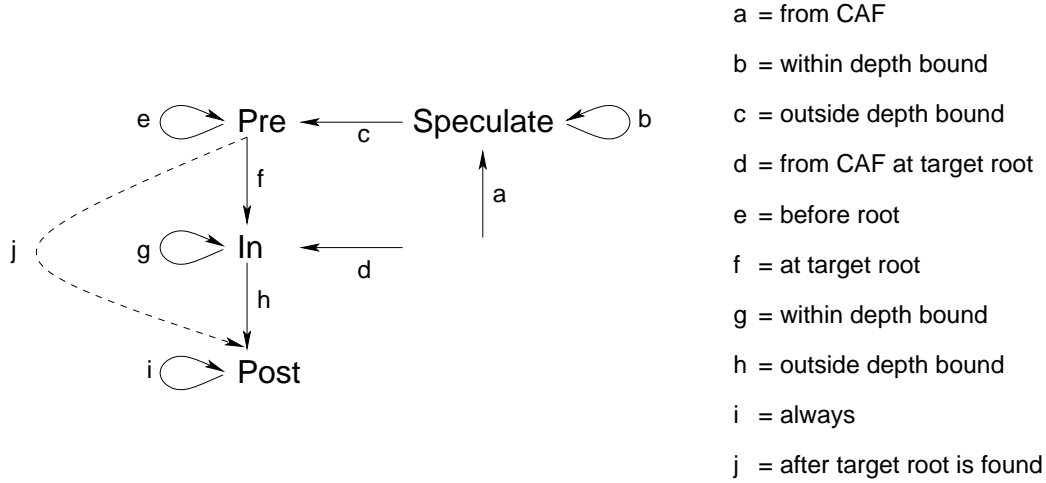


Figure 7.7: Transitions between contexts.

The transitions between contexts are illustrated in Figure 7.7. The transitions labeled *a* and *d* are starting points because they originate from CAFs. All other transitions are performed by `call`. The transition labeled *j* is dashed because it indicates a possible optimisation. The target root is only found once. Therefore all `Pre` contexts can be changed to `Post` contexts after the target root has been identified. This has a potential to improve the running time of the program because `Pre` contexts cause `call` to check for the target root, which makes them less efficient than `Post` contexts.

The functions for generating trusted nodes are modified in a similar way to `call` and `constant`. For brevity they are not shown here.

One final point deserves a mention. Constants can be defined in local declarations, such as `qs` in `scanr` from the Haskell Prelude:

```

scanr f q (x:xs)
  = f x (head qs) : qs
  where
    qs = scanr f q xs
    
```

`qs` is not a global constant because it depends on lambda-bound variables from the head of `scanr`, but it is constant when the values of those variables are fixed — one

could say it is constant within its context. Note that `qs` is used twice in the body and it is important for efficiency that those two uses are shared. Also note that `qs` appears in the middle of a recursive dependency:

```
scanr → qs → scanr
```

If `qs` were treated like a top-level CAF there would be a problem. Suppose that the first call to `scanr` appears inside the materialised EDT. If `qs` were treated as a CAF it would have a depth of zero, which would mean that the recursive call to `scanr` would have a depth of one. If one is less than the depth bound (which is most likely), all the recursive calls to `scanr` in a chain would appear within the materialised EDT.

This problem can be avoided by transforming local constants as if they were function calls, except that their context is not passed in as an argument, but bound in an enclosing scope. So `qs` is transformed like so:

```
qs = call c "qs" [] (\i -> ...)
```

The context value `c` is bound in the enclosing scope by the transformation of `scanr`. References to `qs` in the rest of the transformed code use its name only, they do not pass a context argument (compare this with global constants which do get passed a context argument).

Transforming local constants in this way means that inter-constant dependency information is not reflected in the EDT. Suppose that `scanr` is re-written to include another local definition:

```
scanr f q (x:xs)
  = f x (head ps) : ps
  where
    ps = qs
    qs = scanr f q xs
```

The transformation will make `qs` a child of `scanr` rather than `ps`. If `qs` is buggy, but the wrong answer diagnosis visits `ps` before `qs`, the bug will be mis-diagnosed with `ps`. In fact, this is exactly the same problem identified for global constants, which was discussed in Section 5.5. In the case of global constants it was argued that the

EDT should reflect inter-constant dependencies. The solution presented here goes against that argument. However, the problems identified for global constants are not as serious for local constants. In particular local constants are always defined in the same module, so it is easy to sort them into dependency order.

7.4.4 Performance

We now consider the time and space overheads introduced by the debugger using piecemeal EDT construction. To gauge these costs the full debugging transformation was applied to the suite of five example programs. Measurements were taken based on the construction of the topmost target EDT, at varying depth thresholds. Since we only constructed the topmost target EDT, each program was executed only one time to completion. This provides us with an idea of what the time and space costs are between the start of program execution and the point when the debugger can first begin to explore the EDT. Similar costs will be incurred for subsequent re-executions of the program as additional target sub-trees are generated. It must be noted that the actual costs for any given program re-execution will depend on the size of the target tree, and that can vary considerably with the depth threshold. Therefore these figures should be considered only as a guide. All higher-order functions, except those within the `I0` type, were transformed using the intensional style, and `I0` tabling was enabled. All functions, including those in standard libraries, were considered suspicious.

Figure 7.8 illustrates the growth of EDT size (in nodes) as the depth bound increases. *Pancito*, *Raytrace* and *Mate* exhibit an exponential growth, whereas *Cacheprof* and *Prolog* exhibit a near linear growth.

Figure 7.9 illustrates the growth of memory usage (relative to the original program) as the depth bound increases. Memory usage should grow proportionally to the size of the target EDT. Comparing Figure 7.9 to Figure 7.8 suggests that this is indeed the case.

Figure 7.10 illustrates the growth of running time (relative to the original pro-

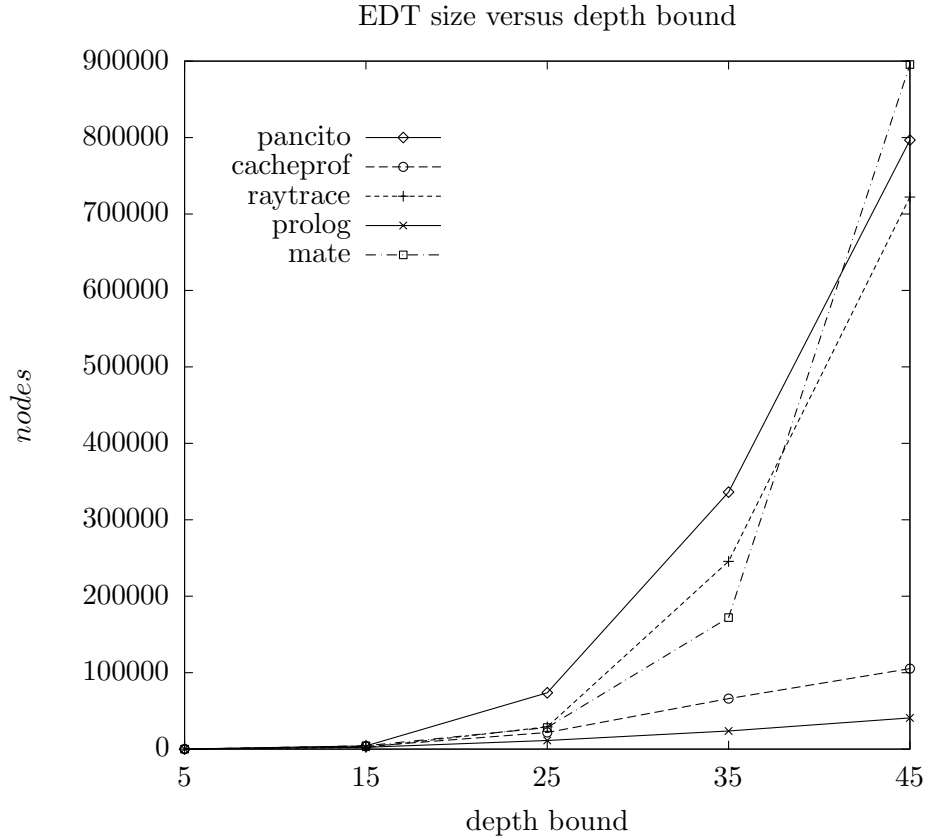


Figure 7.8: EDT size versus depth bound.

gram) as the depth bound increases. To avoid the prohibitive cost of garbage collection on the EDT (caused by the use of `IORefs`) the initial minimum heap for each program was set to 100MB. The overhead of debugging is close to constant when the memory required by the transformed program remains within that limit. This is expected because the debugging transformation introduces only a constant cost for each reduction of a user defined function. As the depth of the tree increases, the cost rises because there are more nodes in the target EDT, and materialised nodes require slightly more work than non-materialised ones. When the program exceeds the 100MB limit the time usage begins to climb rapidly because the garbage collector is invoked more often which triggers its pathological behaviour with mutable

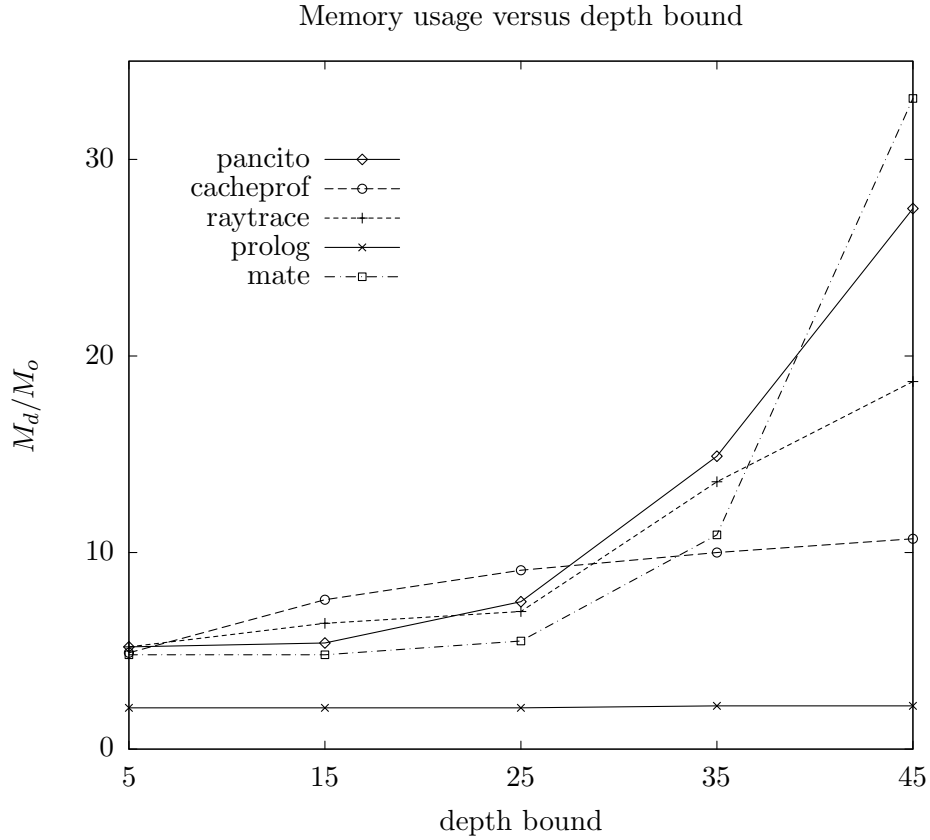


Figure 7.9: Memory usage (relative to the original program) versus depth bound.

data structures. This happens in the *mate* example at a depth between 25 and 35 nodes, which is the same point where its memory usage starts to grow rapidly. It is not clear why *prolog* enjoys a significantly smaller time overhead than the other programs. One factor appears to be that its EDT grows at the slowest rate.

7.4.5 Improvements

It is important to reduce the number of times the debuggee must be re-executed, which means striving for the largest depth bound which produces a target tree which can still fit in memory. Figure 7.11 shows the maximum EDT depths for each of the example program runs. This illustrates that in practice we can expect a diverse

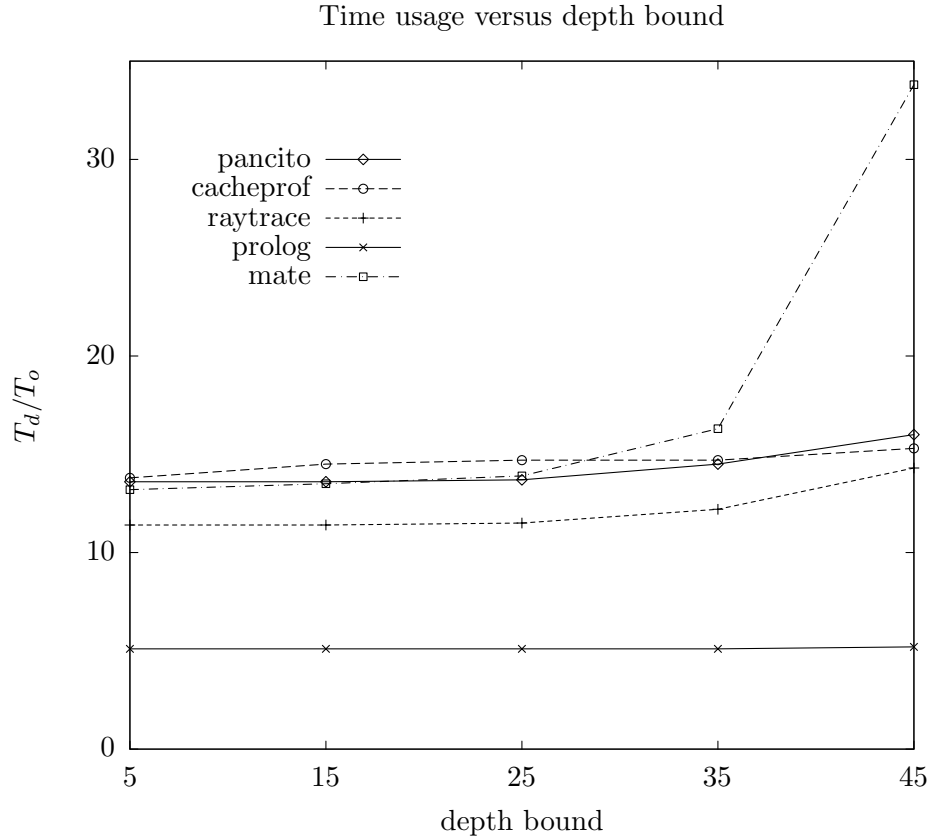


Figure 7.10: Running time (relative to the original program) versus depth bound.

range of EDT shapes, from long sticks to dense bushes. For sufficiently complex programs the same kind of diversity will appear amongst the subtrees of a single EDT. The decision as to what is the most suitable depth threshold is inherently a dynamic one.

There are two approaches proposed in the literature for dynamically determining the best size to materialise parts of the EDT: *query distance* by Nilsson [1998] for Freya, and the *ideal depth strategy* by MacLarty and Somogyi [2006] for Mercury.

Query distance optimises the materialised EDT for a top-down wrong answer diagnosis algorithm. The query distance of a node N measures how many steps are required to get from the target root to N using a top-down left-to-right traversal.

<i>program</i>	<i>max. depth</i>
Pancito	5011
Cacheprof	107007
Raytrace	62507
Prolog	6996
Mate	139

Figure 7.11: Maximum depth of the EDT for the example programs.

Nodes are materialised until some threshold value is reached, based on the amount of memory used by the target. When the threshold value is over-stepped, a pruning process is invoked. Pruning deletes nodes from the target until the memory requirements are back within the allowed limit. Nodes are deleted in order of their query distance, in largest to smallest fashion. This ensures that the target retains the nodes which are most likely to be visited next by the debugger. The size of the target is computed by the garbage collector, which means that the debugger must be tightly integrated with the runtime environment.

The ideal depth strategy calculates exactly how many levels can be generated in a subtree such that some threshold number of nodes (T) is not exceeded. The user of the debugger chooses T to be as large as possible such that any materialised subtree of that size will not exhaust the available memory resources. In practice T is chosen based on previous experience and experimentation. An initial small depth threshold is chosen for the first execution of the debuggee. The fringe nodes of the target store the ideal depth of their subtree, which tells the debugger how deep to materialise that tree if it is eventually needed. The ideal depth for each subtree is calculated as follows. An array A of counters, indexed by node depths, is allocated. All counters are initially zero. The size of A is equal to T , since that is the deepest a materialised tree can be without exceeding size T (which happens if the tree has a branching factor of one at each level). Function calls which occur at depth $d \leq T$ beyond the fringe increment the counter in $A[d]$. The ideal depth is then equal to

the maximum value of D such that:

$$\sum_{i=1}^D A[i] \leq T$$

An important part of the implementation in the Mercury debugger is that only one instance of A is needed, therefore the additional memory requirements are bounded by T . This is possible because Mercury is a strict language. Function calls proceed in a depth-first, left-to-right manner, so that the computation of the result of one fringe node is completed entirely before the next fringe node is encountered. In a lazy language the temporal order of function calls can be interleaved amongst the subtrees of many fringe nodes, which means that it is not possible to use just one instance of A . Instead, every fringe node would require its own copy of A simultaneously. If there are N fringe nodes, the additional memory requirements are $N \times T$ (which approaches T^2 as the average branching factor increases). MacLarty and Somogyi [2006] report that for many programs T should be set to something like 20000 (we expect something of similar magnitude for Haskell programs). For large values of N the required memory is likely to undo any benefits gained from using piecemeal EDT construction.

A possible solution is to approximate A with a smaller array such that the indices represent depths which grow faster than linear, for instance a quadratic function. That is, index one represents level one, index two represents levels two to four, index three represents levels five to nine, and so on. With a quadratic approximation the size of A is bounded by \sqrt{T} . Each fringe node would have its own copy of A , so the total memory requirements would be bounded by $N \times \sqrt{T}$, which is $T^{3/2}$ in the worst case when $N = T$. If $T = 20000$, the worst case would require nearly three million array elements across all the copies of A . With 32 bit integer counters, this would require at least twelve megabytes of memory (assuming an efficient array representation). In practice the worst case is extremely unlikely to happen, so the average memory requirements are expected to be much less.

Another consideration is the cost of the function which converts real depth values

into indices of A . This function must be applied at every call which is beyond the fringe but within a depth of T , which could be a very large number of times, therefore it must be as cheap as possible. For the quadratic function, the conversion is not cheap:

```
index :: Int -> Int
index depth = ceiling (sqrt (fromIntegral depth))
```

Fortunately there is a simple solution to this problem. The `index` function can be pre-computed and stored in an array (Pre), such that:

$$Pre[i] = \text{index } i, \quad \text{for each } i \in 1 \dots T$$

For a given depth d , its counter in A can be incremented like so (using pseudo C-style array indexing and increment syntax):

$$A[Pre[d]]++$$

which is more efficient than applying `index` at every increment. The cost is an additional T integers needed for the elements of Pre . To save space, each integer could probably be 8 bits, since the range of indices for A is bounded by \sqrt{T} , which is unlikely to be bigger than 2^8 .

A quadratic function is just one possible approximation to the linear array. The choice of function is based on a tradeoff between the memory used by all the copies of A versus accuracy in the ideal depth calculation. The larger the error in the ideal depth the more times the debuggee will need to be re-executed. By pre-computing the index function it is possible to employ much more complex approximations to the ideal depth bound. For instance, instead of a quadratic, it might be useful for the function to be linear up to some depth, and then perhaps a polynomial from that point onwards. There is much room for experimentation in this regard.

The above method is conservative in that it will never cause a subtree to be built with more than T nodes, however, in some cases it might be overly conservative.

For instance, suppose that $T = 1000$, and that the ideal depth in some instance is 12. The approximation technique might find that at depth 9 the tree has 300 nodes, but at 16 (the next known data point) the tree has 9000 nodes. The conservative approach is to materialise the tree up to depth 9, but this results in 70 percent fewer nodes than desired. It would not be reasonable to go to depth 16 because this would result in far too many nodes in the tree. It is possible to reduce the error in the approximation by interpolating a curve through the data points. In practice a small positive error in the number of nodes collected is likely to be acceptable, since the value of T is inherently an approximate figure.

Counting EDT nodes provides a reasonable bound on the memory requirements of the debugger in many cases because each node corresponds to a function call, and each function call can only allocate a constant amount of memory. Often sub-parts of data structures are shared between nodes in the target, so the cost of keeping the whole structure in memory is amortised. However, this approximation breaks down when a data structure grows well beyond the boundary of the target. Large structures can arise even in programs which use only a constant amount of memory under normal execution, because only a constant amount of the structure is live at any one time. In the worst case the top node in the EDT refers to a large structure which cannot fit into the available memory of the machine. For example, this situation can arise in **buddha** for programs which perform a long recursive chain of **I0**. The lambda abstractions inside the **I0** type form a long linear structure which is kept in memory because the node for **main** contains a pointer to topmost part of the structure.

The most obvious solution, which is frequently suggested in the literature, is to truncate the data structures which are referred to by the EDT once they grow beyond a certain size. The problem with this approach is that the truncated parts might be important in the diagnosis of a bug. The next refinement is to use truncation but allow the missing parts to be reconstructed by re-executing the program again. This can be achieved within the piecemeal construction technique by expanding the

EDT such that data constructor applications and lambda abstractions get their own nodes. It remains to be seen whether this can be made sufficiently time efficient.

Another tack is to (partially) abandon the top-down search through the EDT. For instance it is possible to start by applying the wrong answer diagnosis algorithm to subtrees which are found deep in the EDT. If no bugs are found in those nodes, the debuggee must be re-executed so that debugging can resume with nodes which are higher in the EDT, continuing upwards to the root if necessary. The benefit is that, generally speaking, nodes which are deeper in the EDT will tend to hold onto data structures which are relatively smaller than nodes which are higher in the EDT. The trouble with this approach is that it is very difficult to decide how deep to start in the EDT. One advantage of the top-down approach is that the discovery of correct nodes can eliminate large amounts of the search space in one step. A bottom-up approach loses some of this advantage, which might result in many more nodes being visited by the debugger.

An even more radical idea, which does not appear to have been previously investigated, is to combine incremental program execution with declarative debugging. A rough sketch follows. Initially the debuggee is executed for a short period, which produces a partial EDT. Declarative debugging is applied to that tree, and if we are lucky, a buggy node can be found, which will eventually lead to a diagnosis. Otherwise the debuggee is executed a bit further, producing more of the EDT, followed by more debugging, and so on. To be effective, some parts of the EDT which have already been visited must be discarded at each step. One way to do this is to make the full term representation of thunks visible in reductions. If a reduction is considered correct, but it contains some thunks, the node containing the reduction and its subtree can be discarded, but the thunks must be tracked by the debugger. If those thunks are eventually reduced, the debugger must revisit them to see whether they contain any bugs. An optimisation is to stop tracking a thunk if it becomes garbage before being reduced. The following example illustrates a possible scenario. Suppose that the debuggee contains a function for producing a list of primes from

some number down to zero. After an incremental execution of the debuggee the oracle might be faced with this reduction:

```
primesFrom 10 => 7 : 5 : primesFrom 4
```

This is correct providing that the underlined thunk is correct. The debugger can discard the current node, and its subtree, but it must track the thunk in case it actually turns out to be erroneous. In some cases there will be too many thunks in a reduction and/or the term representation of thunks will be too unwieldy for the oracle to make a judgement. In that case the debugger will have to hold onto the reduction, and allow the debuggee to be executed further in the hope that some of the thunks will be turned into WHNFs.

There are two main attractions of the incremental approach. First, the debuggee only needs to be executed once, and in some cases a partial execution may be sufficient. Second, reductions with large data structures can be debugged in smaller steps, which will allow memory to be recycled as debugging proceeds. However, it is not without problems. Printing the term representation of thunks is likely to be a technical challenge, especially for a debugger based on program transformation. Then there is the question of how to pause and resume the debuggee — though perhaps this can be achieved on top of a threaded execution environment. But the most pressing issue of all is how to design the user interface so that debugging is not too confusing for the user. The temporal evaluation of tracked thunks is likely to be spread across numerous subtrees of the EDT, and it may be difficult for the user to follow the order in which reductions are visited. On balance it seems that the scalability benefit of an incremental approach has the potential to outweigh the usability problems, making this an interesting topic for further research.

7.5 Final remarks

The current public version of `buddha` does not support piecemeal EDT construction because our implementation relies on `hs-plugins` to re-set CAFs to their original

state, but `hs-plugins` does not support code compiled for profiling. As noted in Section 6.4, profiling is currently needed to obtain printable representations of data constructors in GHC. We are confident that this can be resolved by modifying `bud-dha` so that data constructor names are encoded in the transformed program, thus removing the dependency on profiling.



Chapter 8

Related Work

Constructing debuggers and profilers for lazy languages is recognised 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 functional languages makes us researchers look, well lazy.

Why no one uses functional languages

[Wadler, 1998]

8.1 Introduction



VERY so often a question is posted to one of the Haskell discussion groups on the Internet to the effect of “How do you debug a Haskell program?” When Wadler wrote about “the slow rate of progress on debuggers for lazy functional languages”, there were no debuggers for Haskell. That was several years ago, and happily the rate of progress has increased. Several debugging tools have emerged, with varying approaches to explaining the behaviour of programs. This chapter provides an overview of the most important developments in this area.

8.1.1 Outline of this chapter

The rest of this chapter proceeds as follows. Section 8.2 discusses the most basic of all debugging techniques, diagnostic writes. Section 8.3 considers declarative debugging, starting with the original work in Prolog, then moving to functional languages. Section 8.4 looks at reduction tracing, with an emphasis on Redex Trails. Section 8.5 shows how a step-based tracing debugger can be built on top of optimistic evaluation. Section 8.6 discusses a framework for building many different kinds of debugging tools, based on a special operational semantics for program monitoring. Section 8.7 covers randomised testing. Section 8.8 classifies all the different tools according to their type and implementation, and summarises all their features.

8.2 Diagnostic writes

The most basic approach to debugging, in any language, is the *diagnostic write*. A diagnostic write is a print statement placed at a carefully chosen point in the program in order to reveal its flow of execution or show some intermediate part of its state. The enormous popularity of this technique is largely due to its simplicity. Everything is provided by the programming environment; no other tools are required.

It is desirable to allow diagnostic writes anywhere in the program that computation is performed. In imperative languages this requirement is easily fulfilled. The basic building blocks of those languages are (possibly side-effecting) statements. Squeezing additional side-effects in between existing ones is not difficult and fits with the underlying paradigm. Diagnostic writes are not so straightforward in Haskell, because side-effects are difficult to manage with non-strict evaluation.¹ Input and output must be properly structured within the I/O monad, which limits the places in which print statements can be inserted into a program. Rewriting purely functional code to use the I/O monad is rarely a good option. It imposes a rigid sequential structure on the code where it is not otherwise needed, and it is simply too much

¹The difficulty of using diagnostic writes in pure languages is a well known and long standing problem, for example see the extensive discussion in [Hall and O'Donnell, 1985].

effort for what should be a throw-away piece of programming.

8.2.1 The trace primitive

Most Haskell implementations come with a primitive tracing function, called `trace`, for adding diagnostic writes anywhere in a program:

```
trace :: String -> a -> a
```

Given a string argument, it returns the identity function, but behind the scenes it causes that string to be printed to the output device. In effect, `trace` is just a convenient “hack” to circumvent the type system.

It is quite clear that `trace` is a poor solution to the problem. If and when a diagnostic write will be performed is difficult to predict from the structure of the program source. The documentation accompanying Hugs’² implementation of `trace` shares this view:

[*trace*] is sometimes useful for debugging, although understanding the output that it produces can sometimes be a major challenge unless you are familiar with the intimate details of how programs are executed.

Observer effects are also problematic. Often diagnostic writes are used to display the intermediate value of the program state, such as a local variable in a function call. To use `trace` for this task the value must first be turned into a string. This is usually not difficult, but the act of printing the string often causes the underlying value to be entirely evaluated. This could cause more evaluation than what would normally happen in the program. At best this will mean extra work for the computer, at worst it will result in non-termination or a runtime error. To make matters worse, the extra evaluation might trigger more calls to `trace` which are nested inside the value being printed. This problem was encountered when a `trace`-like facility was added to the Chalmers Lazy ML compiler [Augustsson and Johnsson, 1989]:

²www.haskell.org/hugs

it generally turned out to be very difficult to decipher the output from this, since quite often (due to lazy evaluation) the evaluation by `trace` of its arguments cause other instances of `trace` to be evaluated. The result was a mish-mash of output from different instances of `trace`.

Another problem is that abstract types cannot be easily mapped into strings: functions are a prime example. This is particularly annoying in Haskell where higher-order programming is commonplace. A debugger that cannot display all the values of the language is severely hampered.

8.2.2 The Haskell Object Observation Debugger

The limitations of `trace` are addressed by the Haskell Object Observation Debugger (Hood) [Gill, 2001]. Hood is implemented as a Haskell library which provides a diagnostic writing facility called `observe`. The advantages of `observe` over `trace` are twofold:

1. `observe` preserves the evaluation properties of the program: values are printed in their most evaluated state and no more, and calls to `observe` do not cause their subject values to be evaluated any more than they would have been in an `observe`-free execution of the program.
2. `observe` can handle more types than `trace`; most importantly it can display functional values.

The type of `observe` is similar to that of `trace`:

```
observe :: Observable a => String -> a -> a
```

But the behaviour of the two functions is quite different:

- `trace` can only print a value if it is first turned into a string, it does not look at its second argument at all.
- `observe` records the evaluation progress of its second argument directly, the first argument (a string) is merely a tag for the observation.

It is desirable to have just one function for observing all values, however Haskell's type system does not allow one function definition to pattern match against a given argument at different types. For example, the following definition is ill-typed:

```
toString :: a -> String
toString True    = "True"
toString False   = "False"
toString ()      = "()"
toString []      = "[]"
toString (x:xs) = toString x ++ ":" ++ toString xs
...
```

Hood uses a type class to work around this problem, allowing `observe` to be implemented in a type-dependent manner, hence the '`Observable a`' constraint in its type signature. Only types that are instances of the `Observable` class can be observed. Instances for all the base types are provided by the library, and it is relatively easy to write instances for user defined types.

Calls to `observe` are simply wrapped around expressions of interest. The following example shows how to observe an argument of the `length` function:

```
length (observe "list" [1,2,3])
```

which gives rise to the following observation:

```
-- list
_ : _ : _ : []
```

Underscores indicate terms that were not evaluated by the program. This particular observation shows that `length` demands the spine of a list but not its elements.

A simple implementation of Hood

When a data constructor of an observed value is evaluated, a side-effect records that event in a global mutable table. When the value is a structured object, such as a list, `observe` propagates down through the structure to capture the evaluation of its sub-parts. To ensure that `observe` does not change the meaning of the program

(other than by printing observations at the end), the side-effects must be hidden from the rest of the program.

Hood attaches observation hooks to pure computations using side-effects in much the same way as we do in `buddha`.

First, we consider the encoding of observation events. In a very simple implementation only two events are needed: the start of a new observation, and the evaluation of a data constructor. Start events allow more than one instance of a call to `observe` in any given program run. For easy identification each start event is tagged with a string. Constructor events record the evaluation of a data constructor, giving its name and arity. The following type encodes what kind of event has occurred:

```
data Kind = Start String | Cons String Int
```

It is also necessary to relate events with one another. A constructor will always be the child of another event, either a start event, if it is the outermost constructor of a value, or another constructor, if it is internal to the value. Some constructors have multiple arguments making it necessary to record an index for each of their children. This relation is represented as a list of integers:

```
type Index = [Int]
```

The index of a start event is always a singleton list containing a unique integer, thus all start events can be distinguished. The index of each constructor event is some list of integers of length greater than one. Parent-child relationships are recorded based on positional information. The index `[12,1]` is the first child of the start event numbered twelve. In fact, all start events have just one child, which is always at position one. A more interesting index is `[12,1,3]`, which identifies the third child of the constructor described by the previous event. Combining kinds and indexes gives the full event type:

```
data Event = Event Index Kind
```

All the events in a program run are recorded in a global mutable list:

```
events :: IORef [Event]
events = unsafePerformIO (newIORef [])

updateEvents :: Event -> IO ()
updateEvents event
  = do es <- readIORef events
      writeIORef events (event : es)
```

A unique supply of start event numbers is provided like so:

```
uniq :: IORef Int
uniq = unsafePerformIO (newIORef 0)

newUniq :: IO Int
newUniq = do
  u <- readIORef uniq
  writeIORef uniq (u + 1)
  return u
```

The following helpful utility, called `postEvent`, takes an event and a value, adds the event to the global table and returns the value unchanged:

```
postEvent :: Event -> a -> a
postEvent e x
  = unsafePerformIO $ do
    updateEvents e
    return x
```

This allows us to write an `observe`-like facility for `Int`s:

```
observeInt :: Int -> Index -> Int
observeInt int index
  = postEvent thisEvent int
  where
    thisEvent = Event index (Cons (show int) 0)
```

Given an `Int` and an `Index`, `observeInt` constructs a new event, posts it, and returns the `Int`. The event kind `'Cons (show int) 0'` records the string representation of the integer and the fact that it has zero arguments. The `Index` argument says where this particular `Int` occurs inside a given observation.

Lists of Ints can be handled in a similar way:

```
observeListInt :: [Int] -> Index -> [Int]
-- the empty list
observeListInt list@[] index
  = postEvent thisEvent list
  where
    thisEvent = Event index (Cons "[]" 0)

-- the non-empty list
observeListInt (x:xs) index
  = postEvent thisEvent obsList
  where
    thisEvent = Event index (Cons ":" 2)
    obsList   = observeInt x (index ++ [1]) :
                observeListInt xs (index ++ [2])
```

Start events are created like so:

```
startObsListInt :: String -> [Int] -> [Int]
startObsListInt label list
  = unsafePerformIO $ do
    u <- newUniq
    let rootIndex = [u]
    updateEvents (Event rootIndex (Start label))
    return (observeListInt list (rootIndex ++ [1]))
```

Each call to `startObsListInt` creates a unique “root index” value and adds a start event to the global table. Observations on the list are performed by `observeListInt`, which has an index value of `‘rootIndex ++ [1]’`.

Observations are displayed at the end of the program using the `run0` function:

```
run0 :: IO a -> IO ()
run0 io = do
  io
  es <- readIORef events
  putStrLn (prettyPrintEvents es)
```

The typical use of `run0` is to wrap it around the body of `main`. In this way the argument `io` corresponds to the “original program”, which is run to completion first, after which the global event table is read and pretty printed, via `prettyPrintEvents` (which is not defined here). Wrapping the body of `main` with `run0` ensures that all

```

length (startObsListInt "list" [1,2,3])
  - Post start event with index [0] and label "list"

length (observeListInt [1,2,3] i01)
  - Post constructor event for : with index [0,1]

length (observeInt 1 i011 : observeListInt [2,3] i012)

1 + length (observeListInt [2,3] i012)
  - Post constructor event for : with index [0,1,2]

1 + length (observeInt 2 i0121 : observeListInt [3] i0122)

1 + 1 + length (observeListInt [3] i0122)
  - Post constructor event for : with index [0,1,2,2]

1 + 1 + length (observeInt 3 i01221 : observeListInt [] i01222)

1 + 1 + 1 + length (observeListInt [] i01222)
  - Post constructor event for [] with index [0,1,2,2,2]

1 + 1 + 1 + length []

1 + 1 + 1 + 0
...
3

```

Figure 8.1: Evaluation of an observed expression.

possible updates to the global event table are performed before the observations are printed.

Figure 8.1 illustrates the evaluation of the following expression as a series of term reductions:

```
length (startObsListInt "list" [1,2,3])
```

The figure shows how the side-effects of observation are interwoven in the lazy evaluation of the expression.

Underlining indicates which expression is to be reduced next. A remark underneath an expression indicates what, if any, side-effects are triggered by the reduction

step. Within the expressions, index numbers are written as i_n instead of list notation. For example, i_{01221} would be encoded in Haskell as the list `[0,1,2,2,1]`.

The reduction sequence begins with the posting of a **Start** event, and continues with interleaved reductions of calls to **observeListInt** and **length**. It is interesting to note how the observation calls are propagated down the list. Most importantly the observations of the list elements never occur because they are never demanded by the reduction. Only the parts of a value that were needed by the program are recorded. It is also worth pointing out that an event is posted for a constructor only the first time it is demanded. Future references to the constructor do not trigger any more side-effects.

At the end of the reduction the global state will contain the following list of events:

```
[ Event [0]          (Start "list")
, Event [0,1]        (Cons ":" 2)
, Event [0,1,2]      (Cons ":" 2)
, Event [0,1,2,2]    (Cons ":" 2)
, Event [0,1,2,2,2] (Cons "[]" 0)
]
```

Printing the observations in a more comprehensible manner is straightforward.

One major problem with this simple version is that it requires the definition of an observation function for every *monomorphic* type. Hood avoids this redundancy by overloading **observe** using a type class.

Functions are also observable in Hood, however they cannot be handled in the same way as first-order values because they are abstract — there are no constructors to pattern match against. Hood employs a similar technique to **buddha**, by recording each application of a monitored function.³ For example, if **inc** is a function that increments its argument, it is possible to observe the use of the function in the following expression:

```
map (observe "fun" inc) [1,2,3]
```

³The extensional style of printing functions in **buddha** was inspired by Hood.

If all the elements of the resulting list are needed by the program, the following observation will result, using an extensional representation:

```
-- fun
{ 3 -> 4
, 2 -> 3
, 1 -> 2
}
```

The types of the function's arguments and result must also be instances of the `Observable` class.

8.2.3 Graphical Hood

In the version of Hood described above, all observations are printed statically at the end of the program, by way of `run0`. This kind of printing misses out on one interesting piece of information which is contained in the table of events: the order of evaluation. The event table is ordered by the time at which events occur in the program. Static printing of this information shows *what* constructors were demanded, and *where* they occurred inside other values, but it does not show *when* they occurred. In the original paper on Hood [Gill, 2001], Gill described a more advanced back end that can give a dynamic view of the observations by revealing their order of appearance in the table. Graphical Hood (GHood) is an extension of this idea that uses a tree-based graphical display to show observations that also reveals when they occur relative to one another [Reinke, 2001].

GHood employs `observe` in exactly the same way as Hood. The advantage of GHood is that the dynamic behaviour of the observed value can be played like an animation (forwards and backwards). Values are drawn as trees in the obvious way, and thunks are drawn as red boxes. As parts of the value are demanded, the tree expands with thunks being replaced by constructors: non-nullary ones giving rise to sub-trees. Figure 8.2 shows the GHood interface on a small example.

An interesting application of this animation is the illustration of space leaks. A particular problem with non-strict evaluation is the creation of long-living thunks,

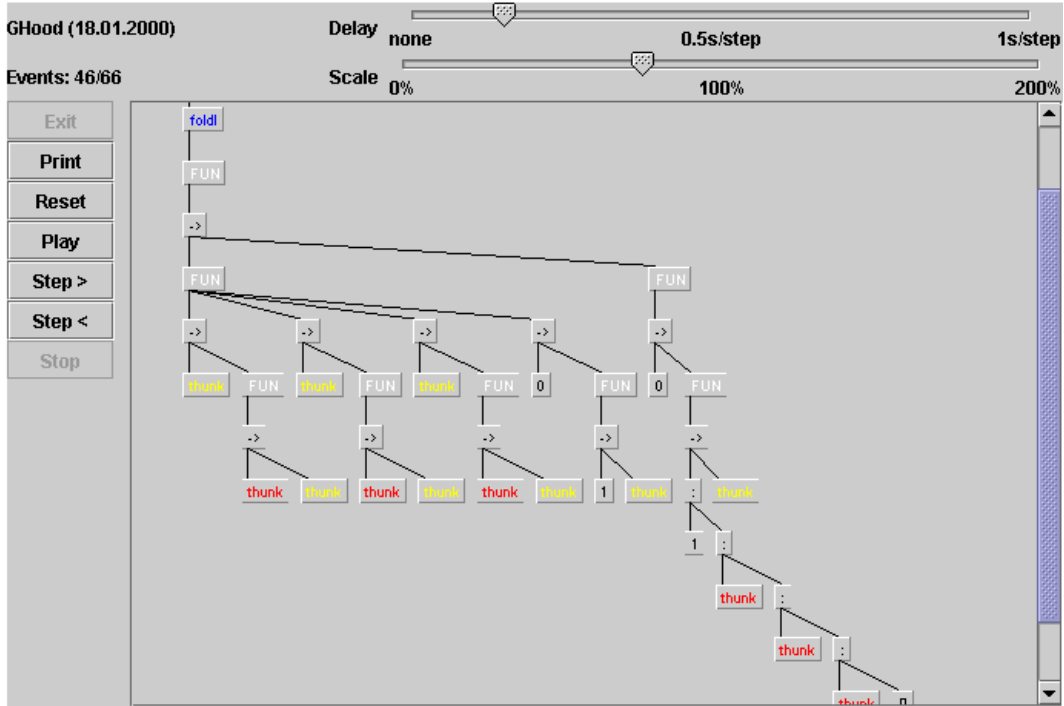


Figure 8.2: The GHood interface.

as noted in Section 2.2.5. Potential space leaks can be identified with GHood by taking note of the thunks that remain untouched for relatively long periods of time.

8.2.4 Limitations of diagnostic writes

Diagnostic writes are a cheap way to probe the behaviour of programs, but their effectiveness is limited in a couple of ways:

- Programs must be modified by hand. This tends to obscure the code and increases the maintenance complexity since diagnostics are usually only needed in development and should not appear in the deployed program. Adding and removing diagnostics can be troublesome and the additional restructuring of the program needed can be a source of errors itself.
- The main faculty provided by diagnostics is vision: the ability to see what value a variable has, or which parts of the program are executed. But seeing does

not necessarily equate with understanding. Though it is useful to know what value a variable is bound to at some point, it is more useful and important for debugging to know why it is bound to that value. The reason why a variable has a given value, or why a function returns a particular result is often related to an intricate web of causation that involves numerous parts of the program source. Diagnostic writes do not help the user to systematically decide what information is relevant in explaining the (mis)-behaviour of the program, so we must resort to trial and error to unravel the web of causation.

8.3 Declarative debugging

Shapiro was the first to demonstrate how to use declarative semantics in a debugging tool [Shapiro, 1983]. He called the process *Algorithmic Program Debugging*, and proposed various diagnosis algorithms for pure Prolog, notably:

1. Wrong answer: a goal (call to a predicate) produces the wrong result for its given arguments.
2. Missing answer: the set of solutions to a non-deterministic goal does not contain some expected answer.
3. Non-termination: a goal diverges (computes indefinitely).

He also investigated *inductive program synthesis* as a way of constructing (correct) programs from the information learned about their intended behaviour during debugging.

Debuggers for functional languages have taken two ideas from Shapiro's work: the wrong answer diagnosis algorithm, and the idea of a semi-automated oracle. Missing answers are not relevant to functional languages because functions are deterministic.⁴ Debugging non-termination remains an open research project.

⁴Though there are ways of simulating non-deterministic computations in Haskell, such as using lists to represent multiple solutions [Wadler, 1985]. One could argue that missing answer diagnosis is suitable for that type of programming.

Shapiro’s debuggers are constructed as modified meta-interpreters (Prolog programs that can evaluate Prolog goals). This greatly simplifies their construction and enhances portability. The reflective nature of Prolog is crucial to his work, as he notes:

Considering the goals of this thesis, the most important aspect of Prolog is the ease with which Prolog programs can manipulate, reason about and execute other Prolog programs.

Modern statically typed languages have not inherited the reflective capabilities of their ancestors like Lisp and Prolog. Therefore much of the research involved in adapting Shapiro’s ideas to languages like Haskell has been in replacing the use of meta-interpretation.

Though his thesis is cast in terms of Prolog, Shapiro notes that the ideas of algorithmic debugging are applicable to a wide class of languages. His requirements are that the language must have procedures (such as predicates in Prolog and functions in Haskell), and that computations of the language can be defined by *computation trees*, with a “context free” property. The idea is that any sub-tree of a computation tree can be considered and debugged without regard to its context (*i.e. referential transparency*). Impure languages do not exhibit this property because the meaning of a sub-tree may depend on some external program state. In logic programming it is natural to use proof trees (or refutation trees as Shapiro calls them) for this purpose [Sterling and Shapiro, 1986], however the same kind of structure is not typical in functional programming (functional languages have traditionally used operational semantics, like reduction, to describe the meaning of programs).

Mercury⁵ is a logic programming language which shares with Prolog a background in predicate logic and a syntax based on horn-clauses, though unlike Prolog, it has a strong static type system including mode and determinism information. Also, Mercury is purely logical in the same sense that Haskell is purely functional, with similar type-level restrictions on where side-effects may be performed. Most

⁵www.cs.mu.oz.au/mercury/

relevant to this discussion is that Mercury comes with a full-featured declarative debugging environment with the following interesting aspects [MacLarty, 2005]:

- Predicates that perform I/O can be declaratively debugged.
- The EDT is constructed from a lower level program trace. A trace is a linear sequence of program events such as call entry, call exit, and for non-deterministic predicates, failure and retry.
- The EDT is built up to a depth bound to save memory. Pruned sub-trees are re-generated by re-executing the sub-goal at their root.
- I/O actions are recorded in a table (memoised) upon the first run of the program. Their results are retrieved from the table if those actions are needed by re-execution of a sub-part of the program. This avoids the need to run an action twice and makes their effects idempotent.
- A user may switch between procedural and declarative debugging in the one session.

Mercury’s approach to debugging I/O [Somogyi, 2003] has been especially influential on the design of *buddha*, as discussed in Section 7.2.

Transferring the declarative debugging ideas from logic languages to Haskell is complicated by non-strict evaluation (Prolog and Mercury are strict). Strict evaluation simplifies the construction of declarative debuggers because:

- There is an obvious correspondence between the shape of the dynamic call graph and the EDT.
- Arguments and results of calls do not contain thunks, which makes them easier to display.
- It is easier to re-execute procedure calls in order to re-generate parts of the EDT on demand.

An additional challenge for Haskell is the tendency for programs to make extensive use of higher-order code. Prolog supports higher-order predicates, but they are uncurried, and their flattened syntax discourages the kind of “deeply nested function composition” style of higher-order programming that you often find in functional languages. A consequence is that support for displaying functional values in logic debuggers is of less importance than it is in Haskell debuggers, where it is crucial.

Two approaches have been taken to build declarative debuggers for non-strict functional languages. The first approach is to create a specialised compiler and runtime environment which builds the EDT internally as a side effect of running the program. This is exemplified by Freya, a compiler and debugger for a large subset of Haskell [Nilsson, 1998, 1999]. The second approach is to apply a source-to-source transformation, producing a program that computes both the original program’s value and an EDT Program transformation [Naish and Barbour, 1996, Pope, 1998, Sparud, 1999, Caballero and Rodríguez-Artalejo, 2002].

The design of `buddha` is inspired by both of these ideas. On the one hand, the method we use to construct the EDT is based in part on the technique used in Freya. On the other hand, we use program transformation to instrument the program, instead of instrumenting the runtime environment.

In the next part of this section we look more closely at Freya, and after that we consider the basic source-to-source transformation schemes proposed in earlier work.

8.3.1 Freya

Freya is a compiler and declarative debugger for a language quite close to Haskell. The only major differences are an absence of type classes and I/O.

For efficiency reasons Freya constructs the EDT at the graph reduction level. The runtime representation of graphs has a couple of important features to facilitate debugging:

- All objects have distinct tags. This is useful to recognising sharing and cycles in values.
- Functions and data constructors are decorated with their name and source locations. Functions also have links to their free variables. This facilitates the printing of arbitrary values.

The garbage collector is aware of the EDT, keeping alive references to values that might later be printed during debugging.

Figure 8.3 illustrates how Freya constructs the EDT for the following small program:

```
double x = x + x
start = double (3 * 2)
```

Refer to Section 2.3.2 to see how graph reduction normally works on this code, especially Figure 2.3. The graph is drawn as before and EDT nodes are indicated inside boxes. The graphs are labeled A - D to indicate the order in which the reduction steps occur. Note that EDT nodes also refer to the arguments and results of function applications, but to simplify the explanation, they are not shown here. It is assumed that at the very beginning a node is allocated for `start`. New nodes are added to the EDT as each reduction takes place. For example, evaluation proceeds from graph A to graph B by reduction of the application of `double`. This event is recorded by the addition of a node in the EDT situated under the node for `start`.

The most difficult part is maintaining parent-child relationships between EDT nodes. Under lazy graph reduction, the context in which an application is *created* may be different to the context in which it is *reduced*. The problem is that the “syntactic” relationship between an application and its parent is only visible in the graph when the application is constructed. However, EDT nodes are only built for

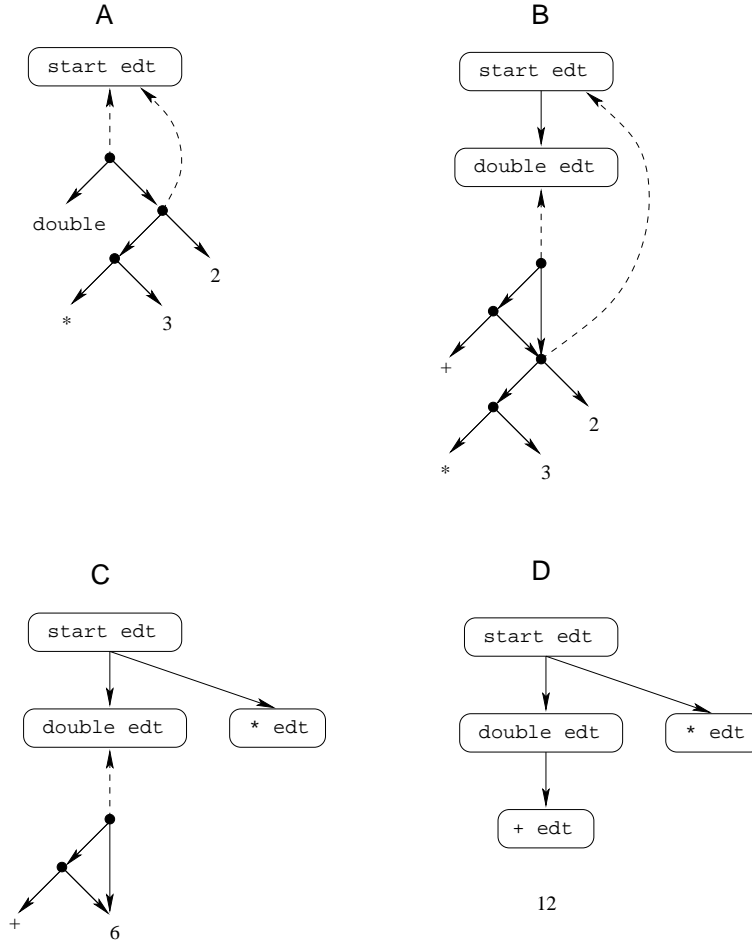


Figure 8.3: Freya's construction of the EDT during reduction.

applications if and when they are reduced, which may be under a different context. Consider the application of `*`. The body of `start` creates the application, but its reduction is demanded underneath an application of `double`. Therefore it is not clear in the normal graph reduction whether `start` or `double` is the parent of `*`. Strict languages do not suffer this problem because function applications are always reduced immediately after they are constructed (creation context and reduction context are the same).

Freya solves the problem by annotating application nodes with pointers back to their creation context (an EDT node). These pointers are indicated in Figure 8.3

by dashed lines. When a reduction takes place three things happen:

1. A new EDT node is allocated, recording the applied function’s name, pointers to the arguments, and a pointer to the result.
2. The reduced application graph is overwritten by the body of the applied function in the usual way. Any new application nodes that are created by this step are annotated with pointers back to the new EDT node. This records the fact that these new applications are “syntactically” children of the applied function.
3. The newly created EDT node is made a child of its own parent. The parent is found by following the application node’s annotation pointer.

Freya builds a big-step EDT, with higher-order values printed in the intensional style. So function applications determine their parents at the point where they are fully applied (following the definition of direct evaluation dependency in Figure 3.2). Thus, as an optimisation, only saturated applications are annotated with pointers back to their creation context.

The idea of annotating application nodes with pointers inspired the design of *buddha*. However, *buddha* is based on program transformation so we cannot annotate the application nodes directly. Instead, we add extra arguments to functions, which serve the same purpose. Also, the idea of annotating graphs is extended in *buddha* because we are interested in two possibly different creation contexts, owing to the fact that we can print higher-order values in two different ways.

To avoid prohibitive memory usage caused by the EDT (and its links to the intermediate values of the program being debugged), Freya employs *piecemeal generation*, as discussed in Section 7.4.

From the user perspective, Freya differs from *buddha* in two main ways:

1. Each CAF produces a separate EDT in Freya resulting in a forest of EDT nodes which must be traversed independently. *Buddha* produces a single EDT rooted at `main`. Both approaches were compared in Section 5.5.

2. Freya only supports the intensional style for printing functions.

The two main limitations of Freya are that it is not portable, and that it does not support full Haskell. At present it only works on the SPARC architecture and it would be a considerable amount of work to port it to other types of machine. It would probably be better to make use of existing compiler technology and transfer the ideas of Freya into a mainstream compiler such as GHC. The necessary modifications to the STG machine are considered briefly in Nilsson [1999], however it is unclear how much work this would be in practice.

The main advantage of Freya's implementation is efficiency: a program compiled for debugging only takes between two and three times longer than normal to execute [Nilsson, 2001], whereas `buddha` incurs a slowdown of around fifteen times. However, it must be noted that `buddha` is built on top of GHC, and GHC is an optimising compiler whereas Freya is not. To some extent the relative overheads introduced by debugging depend on the underlying compiler implementation. Higher values are expected for optimising compilers, which suffer relatively larger penalties by the introduction of debugging instrumentation.

8.3.2 Program transformation

Naish and Barbour [Naish and Barbour, 1996], and Sparud [Sparud, 1999], have suggested program transformations for declarative debugging. They are fairly similar, and to simplify this discussion we present a rough sketch that resembles a union of their ideas. A prototype implementation, and a more detailed discussion can be found in [Pope, 1998].

The transformation goes as follows. A program, which computes some value v , is transformed into one that computes a pair (v, e) where e is an EDT which describes the computation of v . The construction of the EDT is done at the level of functions, so that each function application produces a sub-tree of the EDT as well as its usual value. A new top-level function is added to the transformed program which demands the evaluation of v first and then applies a diagnosis algorithm to e .

We illustrate the transformation of functions by way of a small example. Below is an implementation of list reversal using the well known naive reverse algorithm (it depends on `append` which is not defined here):

```
nrev :: [a] -> [a]
nrev zs
  = case zs of
      [] -> []
      (x:xs) -> append (nrev xs) [x]
```

First, the type signature must change to reflect that all transformed functions return a pair containing their normal result and an EDT node:

```
nrev :: [a] -> ([a], EDT)
```

Second, the body of the function must be transformed to build an EDT node. The body of `nrev` is a case expression with two alternatives. In the first alternative there are no function applications, so there are no nodes to collect. In the second alternative there are two applications, which will each return a value-EDT pair. Nested function applications are flattened, and new bindings are introduced to access the respective values and EDT nodes resulting from them:

```
let (v, ts)
  = case zs of
      [] -> ([], [])
      (x:xs) -> let (v1, t1) = nrev xs
                  (v2, t2) = append v1 [x]
                  in (v2, [t1, t2])
in ...
```

Note the decomposition of the nested function applications:

<u>before</u>		<u>after</u>
<code>append (nrev xs) [x]</code>	\longrightarrow	$(v1, t1) = \text{nrev } xs$ $(v2, t2) = \text{append } v1 \ [x]$

The variables `v1` and `v2` are bound to the original value of the intermediate applications, and `t1` and `t2` are bound to children nodes. An EDT node is constructed for the application of `nrev` as follows:

```

nrev zs
  = let (v, ts)
      = transformed case expression
      edt = EDT "nrev"  -- name
              zs       -- arguments
              v        -- result
              ts       -- children
    in (v, edt)

```

The result of the transformed function is (v, edt) , where v is the value of the original function and edt is the newly constructed EDT node.

Difficulty arises with higher-order functions. Consider the transformation of `map`:

```

map f list
  = let (v, ts)
      = case list of
          [] -> ([], [])
          x:xs -> let (v1, t1) = f x
                     (v2, t2) = map f xs
                   in (v1:v2, [t1,t2])
      edt = ...
    in (v,edt)

```

The transformation of ‘ $f\ x$ ’ implies that the argument f is a transformed function that produces a value and an EDT as its result. The type signature of `map` might be changed to reflect this:

$$\text{map} :: (a \rightarrow (b, \text{EDT})) \rightarrow [a] \rightarrow ([b], \text{EDT})$$

However, this implies that `map`’s first argument is a function that produces an EDT after being given just one argument, which is not always true. The problem is that the transformation only constructs EDT nodes for saturated function applications. However, it is not known whether ‘ $f\ x$ ’ is saturated because, in the definition of `map`, the arity of f is not known.

Type-directed program specialisation was briefly considered as a solution to the problem [Pope and Naish, 2002]. The idea is to analyse the ways in which `map` is called with respect to the arity of its first argument and produce specific clones for each case. Within each clone it is known the ‘ $f\ x$ ’ is saturated. Calls to `map` have to be redirected to the appropriate clone based on the arity of the first argument.

This is quite similar to *de-functionalisation* which translates higher-order programs into first-order ones by specialisation [Bell et al., 1997]. There are many problems with this approach:

- It causes code expansion because of function cloning. In pathological cases this could result in exponential growth of code.
- Type information must be pushed down through the static call graph. This requires a whole program analysis which goes against bottom-up (separate) compilation.
- The possibility of polymorphic recursion in Haskell means that there might be no static limit on the number of clones needed for a given function based on the arities of its higher-order arguments, such as:

```
f :: (a -> b) -> c
f g = f (\x -> g)
```

Each recursive call applies `f` to a function of arity one more than the previous call.

A better solution was proposed by Caballero and Rodríguez-Artalejo [2002]. Each curried function of arity n is unravelled into n new functions with arities 1 to n . All of the new functions except the highest arity represent partial applications of the function and produce “dummy” EDT nodes, which are ignored by the debugger. For example, suppose the program contains a function called `plus` for adding two integers. After transformation the program will contain these two declarations:

```
plus1 :: Int -> (Int -> (Int, EDT), EDT)
plus1 x = (plus2 x, EmptyEDT)

plus2 :: Int -> Int -> (Int, EDT)
plus2 x y = ... usual transformation ...
```

Calls to `plus` with zero arguments are replaced by calls to `plus1` throughout the program. So `'map plus [1,2,3]'` would be transformed to `'map plus1 [1,2,3]'`.

When `plus1` is applied to an argument in the body of `map` it will produce a pair containing a partial application of `plus2`, and a dummy EDT node. The result of the call to `map` will be a list of functions, which after transformation will have the type `'[Int -> (Int, EDT)]'`. If one of those functions in the list is eventually applied to another argument and then reduced it will produce a pair containing an integer and a proper EDT node. Therefore the node for `plus` will be inserted into the EDT in the context where it is saturated.

We improved upon this scheme by using monadic state transformers to simplify the plumbing of EDT nodes throughout the bodies of function definitions [Pope and Naish, 2003a]. The basic idea is that the list of sibling nodes can be treated as program state, which is threaded through all the function applications in the body of a function definition. Saturated function applications insert new nodes into the state, whereas partial applications pass the state along unchanged. This allowed us to avoid the need to introduce dummy nodes in the EDT.

It is interesting to compare this style of transformation with the one used in *buddha*. For the sake of the discussion we will call the style discussed above the *purely functional* style, since, unlike the transformation of *buddha*, the EDT is computed as a result of the program rather than as a side effect.

One of the advantages of the purely functional transformation is that it makes better use of idiomatic Haskell style — no side-effects. This ought to make it easier to reason about, and this is probably true for a declarative reading of the transformed program. However, we found that the purely functional style is more difficult to reason about operationally, and this kind of reasoning is very important in the construction of debugging tools, especially when we are interested in keeping overheads down.

For example, consider this function for computing the length of a list:

```
length [] acc      = acc
length (x:xs) acc = length xs (1 + acc)
```

The function uses an accumulating parameter so that it can be written in the tail

recursive style, and thus uses constant stack space.⁶

Let us compare the two styles of transformation on the second equation. First, in the purely functional style:

```
length (x:xs) acc
  = let (v1, t1) = 1 + acc
        (v2, t2) = length xs v1
        edt = ...
    in (v2, edt)
```

And second, in the style of `buddha`:⁷

```
length context (x:xs) acc
  = call context ... (\i -> length i xs ((+) i 1 acc))
```

In the purely functional style, information about the EDT travels “upwards” through the call graph, because each function returns a node as part of its result. Whereas, in `buddha`’s transformation, information about the EDT travels “downwards” through the call graph, because each function receives a pointer to its parent via an argument.

An important consideration is whether the transformation preserves the stack usage of the original definition. Whilst `buddha`’s version is no longer tail recursive, it is clear that its stack usage remains constant. That is because the body of the original function is wrapped by `call`, but `call` only does a small amount of work to build an EDT node and then returns without evaluating the body of the original function. When `call` is finished we are left with ‘`length c xs ((+) c 1 acc)`’, where `c` is a pointer to the node constructed by `call`. This expression is now a tail call, so the stack usage remains constant. Analysing the stack usage of the purely functional version is much more difficult. The construction of the EDT node is interleaved with the calculation of the normal value of the function. It is rather difficult to see in what order each part is performed without a careful analysis of the intricacies of `let` bindings and lazy pattern matching; so we can be less confident about additional space usage being kept to a minimum.

⁶We should also force the accumulator to WHNF in the recursive equation to avoid an $O(N)$ space leak caused the successive applications of `+`, but we omit that detail to simplify the discussion.

⁷We use a simplified version of the transformation, but the essence is the same.

Another problem with the purely functional version is that EDT nodes are constructed (as thunks) regardless of whether their result value is needed by the program. For example, consider this piece of code:

```
f x = length [plus 1 x, plus 2 x]
```

The values inside the list are not needed so they will not be evaluated under lazy evaluation. Now consider its transformation in the purely functional style:

```
f x = let (v1, t1) = plus 1 x
        (v2, t2) = plus 2 x
        (v3, t3) = length [v1, v2]
        edt = ... [t1, t2, t3] ...
      in (v3, edt)
```

The thunks created for the applications of `plus` cannot be garbage collected because the EDT for `f` refers to `t1` and `t2`. These thunks consume heap space unnecessarily, and the node for `f` appears to have two more children than it ought to. During debugging we have to discard EDT nodes whose result is not in WHNF. Now consider *buddha's* transformation:

```
f context x
  = call context ...
    (\i -> length i [plus i 1 x, plus i 2 x])
```

No new references are introduced to the results of function applications that did not already exist in the original program. So the applications of `plus` can be garbage collected as usual. Furthermore, the node for `f` will have only one child, because nodes are only created for applications that are actually reduced.

8.4 Hat

Many tracing schemes have been proposed, for example [Watson, 1997, Gibbons and Wansbrough, 1996, Goldson, 1994] — too many to explain in detail here — but few have been implemented for nontrivial languages. Probably the lack of large scale implementations is due to the enormous effort required, combined with limited resources. Most tracing debuggers have halted at the proof-of-concept stage. The rest

of this section is dedicated to Hat, which unlike other tracers is a mature debugging tool that supports the full Haskell language (and various popular extensions).

Hat is similar to `buddha` in several ways. Both tools

- are implemented by program transformation;
- aim to be portable;
- give a high-level view of the program execution;
- require the program to be run to completion before debugging begins.

Where they differ is how and what information is recorded. Hat records a very detailed account of every value's reduction history, called a *redex trail*, whereas `buddha` only records an annotated dynamic call graph (EDT). Another difference is that Hat writes its trace to a file, whereas `buddha` keeps the EDT in main memory.

Hat's history goes roughly as follows. Initially, Runciman and Sparud devised the redex trail and a program transformation to produce it. This work is detailed in Sparud's PhD thesis [Sparud, 1999]. The trail was stored in memory, and to save space they proposed various pruning techniques. Runciman, Wallace and Chitil continued with this work and developed the first usable incarnation which was tied to the `nhc98` compiler. This differed from the original work in that a complete trace was recorded and written to file instead of being stored in main memory. Later, the program transformation part of the debugger was removed from the front end of `nhc98`. This allowed a more portable version which works in a similar way to `buddha`. The original program is transformed into a self-tracing one which can be compiled by any full-featured Haskell implementation (at the time of writing Hat works with `nhc98` and `GHC`).

The most important feature of Hat is that it allows many different views of the trace history, giving rise to a number of debugging techniques [Brehm, 2001], for example:

1. `hat-trail`: backwards traversal through the trace starting at a value that appears in the program output.

2. hat-observe: show all the calls to a given function.
3. hat-detect: a declarative debugger.
4. hat-stack: stack tracing for aborted computations, simulating what the stack might look like under eager evaluation.

We will concentrate on hat-trail because no other Haskell debugger offers an equivalent facility (compare: hat-observe with Hood, hat-detect with `buddha` or Freya, and hat-stack with HsDebug).

Below is a buggy program that is supposed to compute the factorial of 3:⁸

```
fac 2 = 3
fac n = n * fac (n - 1)
main = fac 3
```

Running this program gives the answer 9, which is clearly wrong.

Debugging in hat-trail is a search backwards through the reduction history starting from a wrong output value, aiming to discover where it originally came from. More realistic programs will have many outputs which must be searched through to find a suitable starting point. In this case there is only one output value, so the first step is trivial. Selecting an expression causes hat-trail to show where it came from. The initial expression is called a *child*, and the “came from” component is called the *parent* of the child. Parents are themselves expressions. Roots are expressions that have no parents, which correspond to CAFs. The user can decide to select sub-expressions of the parent or delete the parent and go back to the child. The interface resembles an upside-down stack which expands one level downwards each time a new expression is selected, and shrinks one level upwards each time one is deleted. As it happens, the parent of 9 is ‘3 * 3’, within which three possible expressions can be selected: the whole application, and either of the 3s.⁹ Following the parent of the right 3 will lead to ‘fac 2’ and to the buggy line of code.

⁸Inspired by a similar example in Brehm [2001]

⁹Actually there is another sub-expression corresponding to `*` on its own, though it shares the same parent as the whole application. The sub-expressions for function names are left out to simplify the presentation.

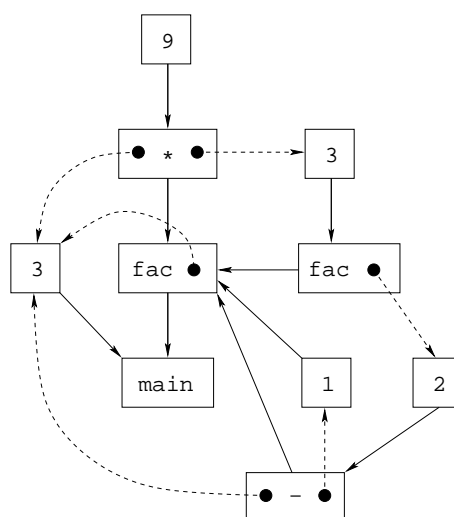


Figure 8.4: Redex trail for the factorial computation.

Figure 8.4 depicts the redex trail for this computation. Solid lines represent paths from children to parents, and dashed lines represent links to sub-expressions. In terms of hat-trail’s interface, movement along a solid line corresponds to pushing a parent expression onto the stack (or popping it off in the opposite direction), whilst movement along a dashed line corresponds to selection of a sub-expression. There are ten different paths leading from `9` to `main`, so there are ten different ways a user can navigate from leaf to root. Each node in the trail has exactly one parent except for `main`, which has no parents because it is a CAF. Nodes in the trail are annotated with source code locations which are displayed when an expression is highlighted.

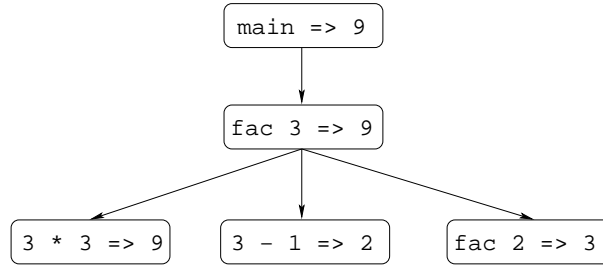


Figure 8.5: EDT for the factorial computation.

In terms of usability there are a number of potential problems with the hat-trail interface:

1. It is easy to get lost in the trail.
2. Exploration starts with program outputs and proceeds backwards. This is useful when the size of the output is small and character based, but one can imagine difficulties with other patterns and types of I/O. A related problem is the situation where the bug symptom is the *absence* of some kind of output. It is not clear where to start exploration in that situation.
3. Higher-order programs can give rise to particularly complex traces which can be hard to understand.

It is unlikely that only one view of the trace will be ideal for all situations, thus it is a major advantage of Hat that multiple views are available. The ability to move seamlessly between views in a single debugging session is being investigated by the developers [Wallace et al., 2001].

An EDT for the factorial computation is given in Figure 8.5 for comparison. To support declarative debugging on top of the redex trail it is convenient to have pointers from parents to children, thus mimicking the links between nodes in the EDT. The current version of Hat employs an Augmented Redex Trail which has both backward and forward pointers between nodes for this purpose [Wallace et al., 2001].

The redex trail is produced by a self-tracing Haskell program, which is generated by a transformation of the original code. The details of the transformation are somewhat complicated, but the principle is simple. The trace file contains a sequence of trace nodes. Each node encodes an application or a constant,¹⁰ and is identified by a reference value which is its position in the file. Trace file references link parents to their children, and application nodes to their arguments, allowing a sequential encoding of a graph. Each expression in the transformed program is paired with its trace file reference. Evaluation of a wrapped expression causes a node to be written into the trace file at the corresponding reference location. Ordinary function application is invalidated because the function and arguments are wrapped up, so special library combinators are introduced to record application nodes and unwrap the components for evaluation.

Storing the trace structure in a file has two distinct advantages over main memory:

1. On modern machines the file-system is typically at least an order of magnitude larger than main memory. This makes it possible to store much larger traces, and thus debug longer running programs. There is one important caveat: the trace viewers must be carefully constructed to avoid reading large portions of the trace at once, lest they re-introduce the need for very large main memories.
2. The trace can be generated once and used many times, amortising the otherwise high cost of trace generation.

A downside is that writing to the file-system is typically several orders of magnitude slower than writing to main memory. The slowdown introduced by Hat is reported in one instance to be a factor of 50 [Wallace et al., 2001], and in another instance between a factor of 70 and 180 [Chitil et al., 2002] (70 when used with `nhc98` and 180 when used with `GHC`). Though main memories tend to be much smaller than the file-system it is possible to keep the space requirements down by

¹⁰There are other types of nodes, for example to identify applications that were not reduced to WHNF and so on, but we overlook such details to simplify the description.

keeping only part of the trace and recomputing the missing pieces on demand. It is much more difficult to take advantage of re-computation when the trace is kept on disk.

8.5 Step-based debugging

Generally speaking, step-based debugging tools are a bad match for non-strict languages because they expose the order in which function calls are made at runtime, which is generally difficult for the user to understand.

There is one case where this argument has proven to be wrong, or at least inaccurate. Ennals and Peyton Jones have shown that step-based debugging is possible in a non-strict language if *optimistic evaluation* is employed instead of lazy evaluation [Ennals and Peyton Jones, 2003a]. Their debugger is called HsDebug and it works just like the kind of debuggers people use in imperative languages: one can set break points on function calls, and single step through each call as they happen.

Optimistic evaluation causes function applications to be evaluated eagerly, sometimes this is called *speculation* [Ennals and Peyton Jones, 2003b]. It is important to emphasise that optimistic evaluation is still non-strict, and on occasion a branch of execution, such as the body of a let-bound variable, might be suspended if the runtime decides that it is too costly. The authors call this technique *abortion*. Also, suspended computations can be resumed at later times if more of their value is needed by the program.

The justification for optimistic evaluation is that laziness is rarely ever needed in practice and most parts of a program can be evaluated eagerly.

Optimistic evaluation provides two main advantages for debugging:

1. The stacking of function calls usually resembles the nesting of applications in the source code. This makes it easier to see how calls are related to the program structure.
2. Argument values are (mostly) evaluated before the body of the function is

entered, making them easier to display and comprehend.

A purely decorative call stack is maintained for tail recursive functions to give informative stack tracing information, though the stack is pruned (or collapsed) if it gets too big.

A consequence of optimistic evaluation is that it is possible to debug a program “as it happens”. A problem with program tracers and declarative debuggers is that they need to run the whole program first, or at least large parts of it, before debugging can commence. This makes debugging seem less immediate, but it also means that the debugger must conservatively record large amounts of information about program values just in case the user might want to view them later on. A step based debugger only has to show a snap-shot of the program state at any given point. Building space efficient step based debuggers is therefore much easier than for tracers and declarative debuggers.

One concern with this approach is the effect of abortion (and resumption) of speculative branches. These jumps in control flow are likely to be hard to follow for the user. However, the authors report that the number of abortions in a typical program run is relatively small, so the extent of their disruption may only be minor in practice.

HsDebug is closely tied to an experimental version of GHC that supports optimistic evaluation. The intimate relationship between HsDebug and the compiler and runtime system means that it works with the same language as GHC, including all its extensions. At the time of writing there is no official release of optimistic GHC or HsDebug, and development has stalled at the prototype stage.

8.6 Monitoring semantics

Kishon et al. argue that a wide range of debugging tools (or monitors) can be built in a systematic way by derivation from a formal semantics [Kishon et al., 1991, Kishon and Hudak, 1995]. Rather than define a single tool, they provide a framework in

which a variety of tools can be understood and implemented. The benefits are:

- **Consistency:** it is easy to prove that the monitoring tools do not change the meaning of the program.
- **Modularity:** language semantics and monitors are described independently. This makes it easier to support new languages and new monitors. Programmers can also define their own custom monitors.
- **Clarity:** the authors lament the lack of formality in the design of most debugging tools. In many cases debuggers are too deeply entangled in the underlying compiler implementation which makes them difficult to understand, maintain and extend.
- **Compositionality:** monitors can be composed together to build more complex programming language environments.

The general idea can be summarised by a simple equation:

$$\text{monitoring semantics} + \text{monitor specification} = \text{monitor}$$

A *monitoring semantics* is a formal semantics in continuation passing style, parameterised with respect to a monitor state (the information manipulated by a monitor as evaluation proceeds). In a standard semantics the meaning of a program is some value $\alpha \in \mathbf{Ans}_{std}$, where \mathbf{Ans}_{std} is a domain of “final values”. In the monitoring semantics the meaning is a function: $f : \mathbf{MS} \rightarrow (\mathbf{Ans}_{std}, \mathbf{MS})$, where \mathbf{MS} is a monitor state. In more specific terms, f is a function from an initial monitoring state to a pair containing the normal answer of the program and a final monitoring state. The continuation style is useful because it exposes a linear order on program evaluation and monitoring is usually interested in “what happens when”. Also, many sequential languages can be given a semantics in this style, thus making the approach quite general.

The underlying language is extended with a finite set of labels, which can annotate any expression in the program. The type and purpose of labels is decided

by the particular monitor. For example, in a profiler, the body of some function g is wrapped in a ‘**Profile**’ label, to request that calls to this function be counted each time the body is evaluated. An annotation function inserts labels into desired program locations just prior to evaluation.

A *monitor specification* is a pair of monitor-state-modifying functions, called **pre** and **post**, which are invoked on every labeled expression. An interpreter for the monitoring semantics threads a monitor state through the evaluation of a program, which is updated by **pre** and **post** whenever a labeled expression is encountered. Each interpreter will also maintain its own state, such as an identifier environment and perhaps a heap, which are also passed as arguments to **pre** and **post** so that they can look up the value of variables and so on. As the names suggest, **pre** is called just prior to evaluating the labeled expression, and **post** just after (**post** is also given the resulting value of the expression as an argument). Some monitors do not need both functions, for example a profiler could use either **pre** or **post** (but not both) to count the number of times a labeled function body is evaluated.

A combining function, called **&**, plays the role of ‘+’ in the above equation. It takes an interpreter for a monitor specification, and a monitoring semantics as arguments, and it returns a monitoring interpreter as its result. Essentially **&** catches all labeled expressions and inserts calls to **pre** and **post** into the normal evaluation pipeline.

In a lazy interpreter, the **pre** and **post** functions can only witness the dynamic call-graph, which means that it is not possible to construct an EDT based on the monitoring semantics as it stands. Two possible solutions are:

1. Extend the annotation phase to a full debugging transformation which constructs the EDT explicitly (such as the one described in Section 8.3.2).
2. Allow the **pre** function to dynamically label expressions to simulate the construction of the EDT used in Freya. The idea is to label each saturated function application with backwards references to its parent redex whenever the body of a function is expanded.

The first solution is probably overkill, since it makes the rest of the framework redundant. The second solution is feasible but it requires a small change in the definition of the monitor specifications to enable dynamic labeling in addition to monitor-state modification.

An obvious problem is that building debuggers based on meta-interpretation can be rather inefficient. The authors refer to Lee [1989], saying that the cost of this technique can be several orders of magnitude slower than hand-written techniques. They propose to use partial evaluation as a solution. Partial evaluation is a technique which specialises a program with respect to part of its input. The result is a less general, but more efficient instance of the original program. For example, partially evaluating an interpreter with respect to a program gives a compiled program as output. There are three levels of partial evaluation available to monitors written in this way:

1. The monitoring semantics can be specialised with respect to the monitor specification to produce a monitoring interpreter as output.
2. The monitoring interpreter can be specialised with respect to a particular input program to produce an instrumented program as output (somewhat akin to the transformed program produced by `Hat`, and `buddha`).
3. The instrumented program can be specialised with respect to a partial input to produce a more specific version of the program.

For partial evaluation up to the second level, they cite three orders of magnitude improvement on some very simple programs. However, these remarkable results should be tempered with the fact that scaling this technique up to full Haskell appears to be a difficult engineering problem, and one that has not yet been overcome.

Even if the monitoring semantics does not produce practical full-scale tools it is nonetheless useful as a test-bed for new debugging ideas. For instance it would not be too difficult to prototype `Hood` in this framework. Problems identified and

solved in this more formal context could inform the implementation of practical hand-written systems.

8.7 Randomised testing with QuickCheck

Testing is one of those things in life that we all know is important but we hate having to do. It is dull laborious work, and when weighed against the more enjoyable parts of program development it can often be neglected. As Larry Wall once said:

Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris.

Of course it is well understood that testing has an important role in quality assurance, and it plays a big part in program debugging [Zeller, 2005, Chapter 3].

Much of the inertia against systematic testing can be attributed to the lack of support from programming environments, though there is ample opportunity for automation. This has motivated QuickCheck [Claessen and Hughes, 2000], a lightweight randomised testing library for Haskell. The idea is to encourage the programmer to codify formal properties of their functions using Haskell as the specification language.

For example, here is a property of `merge`: given two sorted lists it produces a sorted result:¹¹

```
property1 xs ys
  = sorted xs && sorted ys ==> sorted (merge xs ys)
```

Below is a buggy version of `merge`:

```
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y = y : merge xs (x:ys)
  | otherwise = y : merge (x:xs) ys
```

¹¹This property is not a *complete* specification for `merge`. For example, it is easily satisfied by a function that always returns the empty list. Note also that it assumes a correct definition of the `sorted` function.

which can be tested by QuickCheck with the following command:

```
► quickCheck property1
Falsifiable, after 12 tests:
[-4,-3,-3,6]
[3,4]
```

The function `quickCheck` is provided by the library, it takes a property as its argument and applies it to a large fixed number of randomly generated test cases. Here it only took 12 tests to come up with a counter-example. The message is that ‘`merge [-4,-3,-3,6] [3,4]`’ is not a sorted list (and indeed the expression evaluates to `[3,-4,4,-3,-3,6]`). `quickCheck` is overloaded with respect to the type of property it can take as an argument, and hence the domain of test data. Type classes provide the overloading mechanism as usual.

Random generation does not always guarantee a good selection of test cases, and this raises some doubts about the quality of coverage offered by such a tool. QuickCheck addresses this problem by allowing the programmer to write their own data generation methods with carefully skewed distributions. Various monitoring functions, such as histograms, are provided to show exactly what kind of test cases are being produced, which can help inform the creation of even better generators. As with all testing regimes, the lack of counter-examples should not be taken as proof of their absence — all the more reason to study the distribution of test data very carefully.

Perhaps the best aspect of QuickCheck is that it encourages the programmer to think about the formal properties of their functions and codify them in the program. This serves as useful documentation with the extra benefit of being testable. QuickCheck will often reveal problems in the corner cases that might otherwise have been overlooked.

The simplicity of QuickCheck is evidence of the benefits one gets from purely functional code. The absence of side effects greatly simplifies the codification of (partial) formal properties because the correctness of a function’s result only depends on the values of its arguments. This allows a fine-grained approach to testing, and

tool	type	implementation
trace	diagnostic	primitive/library
Hood	diagnostic	library
Freya	declarative	runtime instrumentation
Hat	tracer / multi-purpose	program transformation
HsDebug	step-based	runtime instrumentation
Mon. Semantics	multi-purpose	meta-interpretation
QuickCheck	randomised testing	library
buddha	declarative	program transformation

Table 8.1: Classification of Haskell debugging tools.

according to the authors, random testing generally works best on small portions of code rather than large units.

A helpful tutorial for QuickCheck and Hat is provided by Claessen et al. [2003]. They also highlight how the two tools can be used in tandem as a testing-debugging package.

8.8 Final remarks

8.8.1 Classification of the different tools

To get an idea of the “big picture” of debugging non-strict functional languages it is useful to classify each of the existing tools in terms of their type and functionality.

Table 8.1 classifies each tool under these headings:

- type: the style of debugging offered by the tool.
- implementation: how the tool is constructed.

tool	full Haskell	public	portable	mode	lazy	h/o
trace	yes	yes	no	manual	no	no
Hood	yes	yes	yes	manual	yes	yes
Freya	no	yes	no	auto	yes	yes
Hat	yes	yes	yes	auto	yes	yes
HsDebug	yes	no	no	auto	n/a	yes
Mon. Semantics	no	no	yes	auto	yes	yes
QuickCheck	yes	yes	yes	manual	n/a	yes
buddha	yes	yes	semi	auto	yes	yes

Table 8.2: Features of Haskell debugging tools.

Table 8.2 lists the features of each tool under these headings:

- full Haskell: does the tool support the full Haskell language?
- public: is the tool officially released to the public, and in a usable state (at the time of writing)?
- portable: is the tool compiler independent?
- mode: does the tool require manual intervention by the user, or is it automated?
- lazy: does the debugger deal with lazy evaluation?
- higher-order: does the tool deal with higher-order functions adequately?

Entries marked as “n/a” indicate that the heading is not applicable to the particular debugger. For instance:

- HsDebug supports non-strict evaluation, but not lazy evaluation.
- QuickCheck tests are independent of evaluation strategy.

Buddha is classified as semi-portable because it relies on a handful of compiler-specific hooks. Currently those hooks are only provided for GHC.

The design space for debugging tools is large, and the existing systems seem to be fairly diverse in their combination of type, implementation and features. Of all the tools, Hat and Freya are the most closely related to **buddha**.

8.8.2 Usability

In all this talk of implementation details, it is easy to lose sight of the fact that these tools are intended to be used on real debugging problems. As Richard Stallman once said:

Research projects with no users tend to improve the state of the art of writing research projects, rather than the state of the art of writing usable system tools.

Usability testing is therefore crucial. Chitil et al. [2001] compare Freya, Hat and Hood on several small-to-medium-sized programs. Their conclusions are that each system has its strengths and weaknesses, but no particular tool is optimal in all cases. Also, they generally agree with the advantages and disadvantages of each system identified in this chapter. Perhaps the main limitation of their test cases is that none of them extensively use difficult-to-debug higher-order styles, such as monads and continuation passing style. Since these tests were conducted, Hat has changed significantly, gaining multiple views and independence from `nhc98`, and `buddha` has become publicly available.



Chapter 9

Conclusion

It has been just so in all my inventions. The first step is an intuition — and comes with a burst, *then* difficulties arise. This thing gives out and then that — “Bugs” — as such little faults and difficulties are called — show themselves and months of anxious watching, study and labour are requisite before commercial success — or failure — is certainly reached.

Thomas Edison

[Josephson, 1959]

This chapter concludes our thesis; it has two sections. The first section reviews the main arguments and results from the previous chapters, and briefly summarises the evolution of `buddha`. The second section explores several avenues for future research.

9.1 Review

Debugging Haskell is an interesting research topic because, quite frankly, it is hard, and conventional debugging technologies do not suit it well.

Purely functional languages, along with logic languages, are said to be *declarative*. The uniting theme of these languages is that they emphasise *what* a program computes rather than *how* it should do it. In other words, declarative programs

focus on logic rather than evaluation strategy. The declarative style can be adopted in most languages, however the functional and logic languages tend to encourage a declarative mode of thinking, and are usually used most productively in that way. Proponents of declarative programming argue that the style allows programmers to focus on problem solving, and that the resulting programs are concise, and easier to reason about than equivalent imperative implementations. The declarative style allows more freedom in the way that programs are executed because the logic and evaluation strategy are decoupled. This means that declarative languages can take advantage of novel execution mechanisms without adding to the complexity of the source code; lazy evaluation in Haskell and backtracking search in Prolog are prime examples.

A key aspect of functional languages is that functions are first class values. Higher-order functions provide new opportunities for abstraction and modularity, and are fundamental for certain idiomatic programming styles, such as monads.

The problem with lazy evaluation and higher-order functions is that they make the operational behaviour of programs hard to understand. Lazy evaluation means that the order in which function calls are made at run time is not easy to relate to the static dependencies between functions in the source code. Higher-order functions make holes in the static call graph that are only filled in when the program is executed.

Debugging tools for Haskell must somehow overcome the problems introduced by lazy evaluation and higher-order functions. These issues have seen little attention in the design of debugging systems for mainstream imperative languages because those languages tend to be strict and first-order.

Declarative debugging is a promising approach because it abstracts away the difficult issues of evaluation order, and presents the dynamic behaviour of a program in a fashion which is easily related to the structure of the source code. Also, declarative debugging offers advantages which go well beyond the capabilities of conventional debuggers because:

- The debugger handles the search strategy. Most other debuggers place the burden of deciding “what to do” and “where to go” on the shoulders of the user. Declarative debuggers can take advantage of more sophisticated searches that would not be feasible by hand.
- Declarative debugging is stateless. There is no contextual information to be carried by the user in between interactions with the debugger. In other words, the user does not have to remember what happened in previous steps, or remember the state of any particular value in the program. Each question posed by the debugger can be answered independently, which makes it easy to suspend and resume debugging sessions over longer periods of time, and even swap users.

9.1.1 Chapter summary

In Chapter 1 we introduced the problem of debugging Haskell programs and advocated declarative debugging as a solution. We argued for the use of program transformation as a means to enhance the portability of an implementation.

In Chapter 2 we gave an overview of Haskell, focusing on its most interesting features, including: syntax, pure functions, higher-order functions, types, non-strict evaluation and monadic I/O.

In Chapter 3 we discussed declarative debugging in detail. We defined the evaluation dependence tree (EDT), and the wrong answer diagnosis algorithm. We demonstrated the application of our debugger (*buddha*) on a small example program. We discussed the intensional and extensional styles of printing higher-order values, and related them to the structure of the EDT. We considered the potential for cyclic paths in the EDT due to mutually recursive pattern bindings. We also briefly discussed improvements to the wrong answer diagnosis algorithm to reduce the number of nodes it must visit in order to make a diagnosis.

In Chapter 4 we considered the task of judging reductions which contain partially computed values. We showed that thunks which remain at the end of the pro-

gram execution can be abstracted away to variables which range over closed Haskell terms. Variables which appear on the left-hand-side of a reduction are universally quantified, and variables which appear on the right-hand-side of a reduction are existentially quantified. Inspired by Naish’s three valued debugging scheme [Naish, 2000], we showed that it is convenient to allow the intended meaning of functions to be only partially defined over their domain, thus motivating the use of *inadmissible* judgements. We argued that the extensional style of printing functional values is analogous to the printing of lazy data structures, allowing the same principles of quantification to apply.

In Chapter 5 we defined a source-to-source program transformation over the abstract syntax of “core” Haskell, which extends the behaviour of the original program to produce an EDT as well as its normal value. We showed how to preserve the sharing of pattern bindings, and how the transformation can support the evaluation dependencies needed by both the intensional and extensional styles of printing functional values. We argued for the correctness of the transformation, and measured the runtime performance of transformed programs (without building an actual EDT) on a sample of five non-trivial programs.

In Chapter 6 we considered the problem of implementing a universal facility for printing values. We showed that an implementation in pure Haskell is not feasible, and instead opted for a pragmatic solution based on a foreign function interface to the runtime system of GHC. We showed that a function can be made printable by wrapping it in a data structure which encapsulates both the function and its printable representation. We showed that the program transformation from Chapter 5 can be optimised for the common case of statically saturated function applications, which reduces the overheads of wrapping up functional values for printing.

In Chapter 7 we considered the practical problems of debugging I/O functions and the space usage of the EDT. We showed that the extensional style of printing functions provides a convenient way to display I/O values, which in-turn makes I/O functions amenable to declarative debugging. We showed that we can easily

avoid the construction of nodes for trusted functions, which are quite common in practice. We illustrated a prototype implementation of piecemeal EDT construction, inspired by related schemes in Freya [Nilsson, 1998] and the declarative debugger of Mercury [MacLarty and Somogyi, 2006], and we discussed various ways in which it can be improved. We also measured the runtime and space performance of the prototype on the same five example programs first introduced in Chapter 5.

In Chapter 8 we discussed related work, focusing on the tools built for Haskell, namely: `trace`, Hood, Freya, Hat, HsDebug, Monitoring Semantics and QuickCheck.

9.1.2 The evolution of `buddha`

`Buddha` began life as an honours project to implement the debugging scheme described in *Towards a portable lazy functional declarative debugger* by Naish and Barbour [1996]. The first prototype emerged in 1998 [Pope, 1998], but it had a few shortcomings. First, it only worked with Hugs. Second, it did not provide full support for higher-order functions, because functional values could not always be printed, and the transformation would sometimes produce incorrect output when applied to higher-order code (see Section 8.3.2). Third, it only supported a small subset of Haskell.

At about the same time, Sparud and Nilsson were also working on their own declarative debuggers for Haskell. One of their early contributions was a detailed definition of the (big step) EDT [Nilsson and Sparud, 1997]. Later Nilsson would produce Freya [Nilsson, 1998], and Sparud would produce a debugger based on Redex Trails [Sparud, 1999], which would later form the basis of Hat. Sparud also worked on a declarative debugger based on program transformation, but a usable implementation did not emerge, and it appeared that his approach suffered similar problems with higher-order functions to that of Naish and Barbour [1996].

For some time, Freya was the most complete debugging system available for a lazy functional language, but it did not support full Haskell, and only worked on one kind of system architecture. We believed that program transformation was a

reasonable way to overcome these limitations.

In 2000 `buddha` entered its second phase, as a PhD project. We wanted to port the earlier prototype from Hugs to the more substantial GHC. However, the problem of transforming higher-order functions correctly and efficiently remained unsolved, and printing values seemed even harder in GHC than it was in Hugs (mainly because GHC is a compiler and it does not maintain the same amount of meta-information about heap objects as does Hugs).

A debugger must be able to print *all* the values which arise in the execution of a program. In Haskell this means we must be able to print data values and functions. To print a function we must sometimes print data values, such as the arguments to a partial application. Therefore, we decided to solve the problem of printing data values first. Initially we experimented with an overloaded printing function based on type classes (similar to what was suggested in [Sparud, 1999]), but this failed because it cannot support polymorphic values (see Section 6.3). Instead we opted for a more pragmatic solution, based on an interface to the runtime system of GHC by way of the FFI. Whilst this reduced the portability of the debugger, it was simple to implement, and it worked well in practice. Unfortunately the same technique cannot be used to print functions because GHC’s heap representation does not carry enough source-level information. So we had to look for some way to encode the necessary information into the program. Curiously, HOOD was released in 2000, and it supported the printing of functions using an extensional style. We briefly considered adapting this approach for our purposes, but it seemed that the declarative debugging technique needed functions to be printed in the intensional style — at least that was the silent assumption in the literature up to that point. It was not until much later that we discovered how to re-structure the EDT to accommodate the extensional style. We eventually opted to wrap up functions inside a data structure which contained both the actual function and an encoding of its term representation (which we elaborated in [Pope and Naish, 2003b]); an idea that we had previously considered [Pope, 1998], and which was also suggested in [Sparud,

1999].

Once we had a working printer, we had to solve the problem of transforming higher-order functions to produce a correct EDT.

Our first approach was to specialise the program with respect to the arities of higher-order functions [Pope and Naish, 2002]. But this idea had a number of problems. It required a whole program analysis, which does not work well with separate compilation, and it did not support certain kinds of polymorphic recursion, as discussed in Section 8.3.2. It was also difficult from an implementation perspective because it required type information, and that meant we had to write a type checker for Haskell, which was an arduous task in itself.

Fortunately, a simple solution came from Caballero and Rodríguez-Artalejo [2002]. We took this idea and modified it to use a monadic style [Pope and Naish, 2003a]. We combined this with our earlier work on printing to produce the first proper release of `buddha` (version 0.1) in November 2002. This version supported most of the syntax of Haskell 98 as well as a large part of the standard libraries.

Having built a debugger we decided to test it on various example programs. Two things became immediately obvious: the space usage of the EDT would be a limiting factor for debugging real programs, and debugging certain kinds of higher-order code was nigh impossible. The space issue was already well known, but the second problem came as something of a surprise. The difficulty of debugging higher-order functions was quite apparent when we tried to debug a program which used parser combinators in the style of Hutton and Meijer [1998]. The intermediate parser values constructed by the program were large compositions of functions, including many lambda abstractions, and their term representations, as printed by the debugger, were extremely difficult to comprehend. Turning to the literature, we found that the issue of debugging this kind of code had not received much attention. We decided to postpone the problem of debugging higher-order code and work on the space problem, since this issue was better understood, and various solutions had already been proposed in the literature. Again we found that we could make some

traction by interfacing with GHC’s runtime environment, which lead to our first prototype implementation of piecemeal EDT construction [Pope and Naish, 2003b], based on the method suggested by Naish and Barbour [1996].

We returned to the issue of debugging higher-order code after making the important realisation that the extensional style of printing could be used in a declarative debugger if we employed a slightly different notion of evaluation dependency (see Section 3.5). Unfortunately adapting our existing transformation scheme to support this new notion of evaluation dependency proved difficult. The main problem was that our existing scheme related only saturated function applications, but the extensional style required a relationship between (possibly) partial applications. By building the EDT in a “bottom up” fashion, it was difficult to insert nodes for partial applications under their correct parent. We solved this problem by adopting a “top down” approach to building the EDT, such that function applications receive pointers to their parent nodes by way of an additional argument, called a context. Functions printed in the extensional style take their parent context at the point where they are first applied, and functions printed in the intensional style take their parent context at the point where they are saturated. However, to allow nodes to be inserted into the correct location we needed to represent parent contexts by mutable references. Thus we had to abandon the idea of building the EDT in a purely functional way. The first version of `buddha` to incorporate this scheme was version 1.2, released May 2004 [Pope, 2005].

The extensional style of printing functions dramatically improved the comprehensibility of questions posed by the debugger on the parser combinator example mentioned earlier. Encouraged by this result we began looking for other difficult higher-order examples. We considered other kinds of monads, and quickly our attention turned to the I/O monad. The I/O type is commonly implemented as a state threading function, where the state is simply a token that stands for the world. In previous work we had tried to print I/O values in the intensional style [Pope and Naish, 2003b]. But this often resulted in an unwieldy printout, and the resulting

structure of the EDT was difficult to relate to the source code. In particular, users tend to use the do-notation for I/O, but underneath that syntax is a complicated chain of higher-order functions. The intensional style places nodes for those functions in the context where they are saturated, but that is far removed from the place where the functions are first mentioned. It occurred to us that we could represent the world as a counter, where each increment of the counter corresponds to the execution of a primitive action. To make the primitive actions printable it was simply a matter of storing them in a table, indexed by the world counter. By printing the I/O type in the extensional style we found a simple way to relate the value of an I/O function with the primitive actions that it produced. We also discovered a secondary benefit of this approach, namely that the dependencies between nodes in the EDT closely resembled the dependencies suggested by the use of do notation (see Section 7.2.2).

The change in program transformation style also had a positive affect on piecemeal EDT construction. We found that the context arguments, which are used to link children nodes to their parents, also provide a useful way of controlling how deep to build a sub-tree (see Section 7.4.3). Based on this idea, we were able to build a prototype re-evaluation scheme, which works in a similar fashion to the scheme in Freya. We were also able to make I/O actions idempotent by retrieving the values of previous executions from the I/O table. The main limitation of our prototype is that the depth bound is a fixed value, however the branching factor can vary widely within the sub-trees of an EDT. In Section 7.4.5 we proposed an adaptive method, which is based on the algorithm used in the Mercury declarative debugger, but modified for a lazy language. We plan to incorporate this improved scheme into the next version of `buddha`, which will allow users to debug more substantial program runs.

```
type Root = Maybe (Double, Double)

quadRoot :: Double -> Double -> Double -> Root
quadRoot a b c
  | discriminant < 0 = Nothing
  | otherwise       = Just (x1, x2)
  where
    discriminant = b * b - 4 * a * c
    rootDiscrim = sqrt discriminant
    denominator = 2 * a
    x1 = ((-b) + rootDiscrim) / denominator
    x2 = ((-b) - rootDiscrim) / denominator

intersect :: Root
intersect = quadRoot 1 0 (-16)
```

Figure 9.1: Computing the roots of a quadratic equation in Haskell.

9.2 Future work

Now that we have a working debugger it is possible to consider how it can be improved. This section briefly discusses the main problems that we would like to tackle in the immediate future.

9.2.1 Printing free lambda-bound variables

Figure 9.1 contains Haskell code for computing the real roots of a quadratic equation $f(x) = ax^2 + bx + c$, using the well known formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Local definitions introduce nested scopes in the program. The `where`-clause in `quadRoot` illustrates the idea. Variables bound in outer scopes are also visible in the local definitions, including those variables that are bound in outer lambda-bindings. For example, `a`, `b` and `c` are in scope in the bodies of the functions defined in the `where`-clause in `quadRoot`, which means that occurrences of those

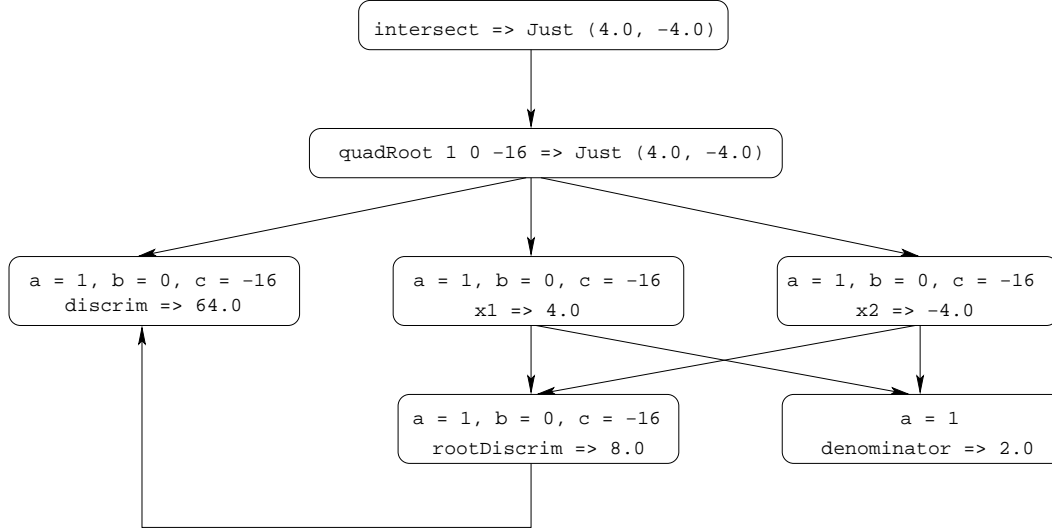


Figure 9.2: An EDT for the program in Figure 9.1.

variables are *free* in those definitions. Thus `discrim` is a function of `a`, `b` and `c`, even though those variables are not bound in its head. To determine the correctness of a reduction involving `discrim` the user must know the values of `a`, `b` and `c` — without this information it is impossible to say what the value of `discrim` should be. Therefore, reductions involving locally defined functions must indicate the values of those variables in the EDT. Dependence on a free lambda-bound variable can be indirect, for example `rootDiscrim` depends on `a`, `b` and `c` because `discrim` appears in its body. Not all local definitions depend on all the lambda-bound variables that are bound in outer scopes, for example `denominator` only depends on `a`. To minimize the amount of information contained in an EDT node, it is an advantage to show the values of only those free variables which are actually needed for any given reduction. Thus a dependency analysis is needed to determine which free variables are transitively depended upon by which local definitions.

Figure 9.2 illustrates the EDT for the example program. Nodes for nested bindings show the values of free lambda-bound variables above the reductions that depend on them.

Buddha does not yet support the printing of free let-bound variables, however it is

a relatively simple feature to add. We propose that the free variable information be added to the representation of identifiers, by extending the definition from Figure 3.3 like so:

```
type Identifier = (IdentSrc, FreeVars)
type IdentSrc   = (FileName, IdentStr, Line, Column)
type FreeVars   = [(IdentSrc, Value)]
```

Local functions can be partially applied and passed as higher-order arguments to other functions, just like top-level functions. When the partial application of a local function is printed in the intensional style we should also print the values of its free variables. This will be possible if we adopt the new representation of identifiers above, because we also use `Identifier` in the meta representation of Haskell terms (see Section 6.4), which forms the basis of the intensional representation.

9.2.2 Support for language extensions

At present `buddha` only supports Haskell 98, however many useful extensions to the language have been added to compilers (especially GHC). The most prominent of these are:

- Multi-parameter type classes [Peyton Jones et al., 1997]
- Concurrency [Peyton Jones et al., 1996]
- Imprecise exceptions [Peyton Jones et al., 1999]

The Haskell community is currently in the process of creating a new standard for the language, which is likely to include these extensions. Obviously it is a high priority for `buddha` to support the new standard when it is finalised.

It is expected that multi-parameter type classes will not pose any significant problems for the transformation. Concurrency and imprecise exceptions are more difficult, and will require some changes to be made to `buddha`. These are discussed briefly in the remainder of this sub-section.

There are three issues which need to be addressed for concurrency. The first issue is EDT construction. Children nodes are inserted into their list of siblings by destructive update (using `IORefs`). In a concurrent setting, there is a possibility of a race condition occurring such that the reads and writes to an `IORef` are interspersed between two different threads of execution. Therefore modifications to mutable sibling lists must be made atomic. The second issue is I/O event tabling. All primitive I/O events are logged in a table, which is indexed by a world counter. This assumes that there is a total order over all such events. In a concurrent setting, I/O events are only ordered with respect to a particular thread. The ordering between threads is non-deterministic, and two runs of the same program may produce different orderings. Therefore each thread will need its own unique counter. The third issue is re-execution for piecemeal EDT construction. Nodes in the EDT are uniquely numbered (as discussed in Section 7.4) using a global variable. When the program is re-executed it is essential that all nodes get the same number. The danger is that when the program is re-executed its threads will be scheduled in a different order than in a previous run. If this happens, the numbering of nodes will not be preserved. It is an open question whether we can ensure that threads are scheduled in the same order for each execution of the program, but it is likely to be difficult in a system like GHC where scheduling is influenced by memory allocation.

In Haskell 98 exceptions can only be raised by I/O primitives, which means that it is relatively simple for `buddha` to discover them (as discussed in Section 7.2.3). The main challenge with imprecise exceptions is that they can be raised in any type context, not just I/O. In GHC, imprecise exceptions are implemented in a very low-level way. `catch` calls a primitive function which places a special exception-handler frame on the call-stack. If an exception is raised, the runtime system collapses the call-stack until the topmost handler is reached (if one exists). All this machinery is invisible to `buddha` which makes it more difficult to implement a version of `catch` which works with `buddha`'s own I/O type.

9.2.3 Integration with an interpreter-style interface

A useful aspect of Haskell is that programs can be easily composed from smaller parts. Interactive development environments like Hugs and GHCi capitalise on this property. Programmers can build small pieces of a program at a time and then test them in isolation before moving on to other parts of the system. This is good for debugging too, because testing can be done when the code is fresh in the programmer’s mind, and the units of code involved in the test can be kept small. When the user discovers a bug, they normally want to start debugging immediately, without having to write scaffolding code or step out of their development environment. One of the biggest usability problems with `buddha` is that it only works with complete programs, and debugging always starts at `main`.

In future versions of `buddha` we will address this usability problem by imitating the interface of an interactive interpreter. We will offer the user a command prompt, at which they can type any valid Haskell expression, and have it evaluated immediately — just like Hugs or GHCi. This can be achieved (in a fairly standard way) by “faking” an interpreter with a compiler. When the user types an expression the interpreter writes out a new Haskell module to disk. The module contains the expression wrapped in sufficient code to make it a full program (in essence by demanding the expression to be printed). The module is compiled and dynamically loaded into the interpreter, then executed with its result printed at the prompt. We will modify this basic system by allowing the user to prefix an expression with a “debug” command. In this case the interpreter will follow a similar path as before, except this time the expression will first undergo the debugging transformation. The transformed code will be compiled and executed, and the result will be printed. However, instead of returning to the interpreter prompt, control will be given to a built-in declarative debugger, which will explore the EDT. Quitting the debugger will take the user back to the interpreter prompt. In many cases this kind of interface offers a performance advantage for debugging as well. The user can be more selective about which part of a program to debug, which is likely to produce an EDT which is much smaller

than the one for the whole program.

9.2.4 Customisation

Rather than write a new language from scratch — including all the infrastructure that goes with it — it is often easier to embed the new language in an existing host [Hudak, 1996]. Higher-order functions, a powerful type system with overloading, flexible syntax, and lazy evaluation all combine to make Haskell ideal for implementing domain specific languages (DSLs). Lava is one such example, which is used for describing digital circuits [Claessen, 2001], and there are many more.

Whilst embedding the DSL in a host language has many advantages, there are problems when it comes to debugging, as noted in [Czarnecki et al., 2003]:

Trying to use the debugger for the existing language is not always useful, as it often exposes the details of the runtime system for the host language which are of no interest to the DSL user.

We observe similar problems when it comes to debugging mini-DSLs, like monads (and no doubt arrows [Hughes, 2000]), as mentioned in Section 7.2.2.

Rather than write a new debugger for the DSL, it would be preferable to customise an existing debugger, so that it shows a view of the program which better reflects the new programming domain. We already do this in an *ad hoc* way in *buddha* for I/O. Values of the I/O type are shown in an abstract way, using the extensional notation, because the user is not interested in, or knowledgeable about, its concrete representation. Similarly we avoid showing reductions for `>>=` and `return`, by trusting them, since they are really part of the “host” language. An interesting direction for future research is to extend the concept of customisation, so that the debugger can be specialised for arbitrary DSLs. In the very least this will require specialised printing routines for data values (especially abstract data types), and a mechanism for taking views of the EDT, so that host language facilities are hidden from the user.

9.2.5 Improved EDT traversal

The current top-down left-to-right wrong answer diagnosis algorithm of Figure 3.4 is simple to implement, but it results in long debugging sessions when buggy nodes are found deep in the EDT. More efficient algorithms can be found in the literature, such as those discussed in Section 3.7. We will consider whether such algorithms can be adapted for `buddha`.

We will also look at improving the capabilities of the oracle. At present the oracle simply remembers previous judgements made by the user, but there are a whole host of improvements that could be added to make it more powerful. One interesting example is the use of QuickCheck properties (see Section 8.7), for partial definitions of the intended interpretation of the program. A node is erroneous if it falsifies any properties which are relevant to the function being defined. For this to work, a couple of problems have to be overcome. First, the debugger will somehow have to execute the properties over a reduction. This will require some kind of dynamic code execution (perhaps interpretation). Second, some reductions will contain non-Haskell values, such as question marks (for thunks), and extensional representations of functions. QuickCheck properties are only defined over normal Haskell values, so they will have to be “lifted” somehow to cope with the unusual parts of reductions.

9.2.6 A more flexible EDT

In the traditional view of declarative debugging, reductions in the EDT reflect a big step semantics (as discussed in Section 3.3). That is, argument and result values are shown in their most evaluated form. This view is based on an underlying heuristic that big step reductions are easier to understand than those which are in some kind of intermediate state.

In the process of developing `buddha` we began to question this heuristic. We discovered that debugging some instances of higher-order code can be quite difficult when higher-order values are printed using their term representation (see Sec-

tion 3.5). Sometimes reductions are easier to understand if the higher-order values contained in them are printed in an extensional way. Incorporating the extensional style of printing into `buddha` forced us to reconsider the structure of the EDT.

We realised that the concept of evaluation dependency can be made more flexible. For a given program execution there are many possible EDTs that can be superimposed over the underlying sequence of reductions. Any one of those trees can be used as the basis of a wrong answer diagnosis.

One problem with the big step EDT is that we must wait until the program has terminated before debugging can begin — so that all values are in their final state. This introduces problems with space consumption, as discussed in Section 7.4. Whilst piecemeal EDT construction provides a partial solution to that problem, there are numerous complexities in its implementation. Another problem with the big step EDT is that the final state of a value is not always the easiest to understand, especially if the final state is a very large object.

Some of the problems with the big step EDT can possibly be avoided by allowing for a more flexible definition of evaluation dependency — one that allows different reduction step sizes in reductions. For example, in Section 7.4.5 we suggested an incremental approach which allows debugging to be interspersed with program evaluation, in the hope that memory resources can be more effectively recycled.

In future work we will investigate a more general notion of evaluation dependency which allows variable reduction step sizes in reductions. Hopefully this will illuminate new possibilities in debugger design, and provide a basis for their implementation.



Bibliography

- M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.
- L. Augustsson and T. Johnsson. The Chalmers lazy-ML compiler. *Computer Journal*, 32(2):127–141, 1989.
- H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984.
- J. Bell, F. Bellegarde, and J. Hook. Type-driven defunctionalization. In M. Tofte, editor, *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 25–37, 1997.
- C. Bentley Dornan. *Type-Secure Meta-Programming*. PhD thesis, University of Bristol, United Kingdom, 1998.
- T. Brehm. A toolkit for multi-view tracing of Haskell programs. Master's thesis, RWTH Aachen, 2001.
- F. Brooks, editor. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, 1975.
- R. Caballero and M. Rodríguez-Artalejo. A declarative debugging system for lazy

BIBLIOGRAPHY

- functional logic programs. In Michael Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier Science Publishers, 2002.
- B. Cantwell Smith. Reflection and semantics in Lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, 1984.
- M. Chakravarty. Haskell 98 Foreign Function Interface. www.haskell.org/definition, 2002.
- O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP’01)*, pages 193–204, Firenze, Italy, September 2001. ACM Press.
- O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood - a comparative evaluation of three systems for tracing and debugging lazy functional programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, volume 2011 of *LNCS*, pages 176–193. Springer, 2001.
- O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In R. Pena and T. Arts, editors, *Implementation of Functional Languages: 14th International Workshop, IFL 2002*, volume 2670 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2002.
- A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- K. Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.
- K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM Press, 2000.

- K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and Tracing Lazy Functional Programs using QuickCheck and Hat. In *4th Summer School in Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 59–99. Springer, August 2003.
- H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.
- K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer, 2003.
- R. Dahl. *The Gremlins*. Collins, 1943.
- R. De Millo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.
- E. Dijkstra. On the reliability of programs (EWD303). URL <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF>. Circulated privately, n.d.
- R. Ennals and S. Peyton Jones. HsDebug: Debugging lazy programs by not being lazy. In *Proceedings of the ACM SIGPLAN 2003 Haskell Workshop*, pages 84–87, 2003a.
- R. Ennals and S. Peyton Jones. Optimistic evaluation: An adaptive evaluation strategy for non-strict programs. In *Proceedings of the Eighth ACM SIGPLAN Conference on Functional Programming*, pages 287–298, 2003b.
- K-F. Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12: 295–357, 2002.
- P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM LOPLAS – Letters of Programming Languages and Systems*, 1(4):303–322, December 1992.

BIBLIOGRAPHY

- E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot. www.research.att.com/sw/tools/graphviz/dotguide.pdf, 2002.
- J. Gibbons and K. Wansbrough. Tracing lazy functional languages. In *Proceedings of CATS'96: Computing the Australasian Theory Symposium*, Melbourne, Australia, January 1996.
- A. Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *Electronic Notes in Theoretical Computer Science*, August 2001.
- D. Goldson. A symbolic calculator for non-strict functional programs. *Computer Journal*, 37(3):178–187, 1994.
- C. Hall and J. O'Donnell. Debugging in a side effect free programming environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 60–68. ACM Press, 1985.
- C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.
- K. Hammond and C. Hall. A dynamic semantics for Haskell (draft). www.haskell.org/haskellwiki/Language_and_library_specification, 1992.
- W. Harrison, T. Sheard, and J. Hook. Fine control of demand in Haskell. In *Proceedings of the Sixth International Conference on the Mathematics of Program Construction*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer, July 2002.
- J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 1996.
- R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

- C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4), 1996.
- P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21:359–411, 1989.
- J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.
- J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.
- G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.
- T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, 1984.
- M. Jones. Typing Haskell in Haskell. Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, Utrecht, Netherlands, 1999.
- N. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 296–306, Florida, 1986. ACM Press.
- M. Josephson. *Edison*. McGraw Hill, 1959.
- A. Kishon and P. Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–574, 1995.
- A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: A formal framework for specifying, implementing and reasoning about execution monitors. In *Proceedings*

BIBLIOGRAPHY

- of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 338–352. ACM Press, June 1991.
- D. Knuth. *The Tex Book*. Addison-Wesley, 1984.
- J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM Symposium on the Principles of Programming Languages*, pages 144–154. ACM Press, 1993.
- P. Lee. *Realistic Compiler Generation*. MIT Press, 1989.
- X. Leroy and M. Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.
- H. Lieberman. The debugging scandal and what to do about it. *Communications of the ACM*, 40(4):26–29, 1997.
- I. MacLarty. Practical declarative debugging of Mercury programs. Master’s thesis, The University of Melbourne, 2005.
- I. MacLarty and Z. Somogyi. Controlling search space materialization in a practical declarative debugger. In P. Van Hentenryck, editor, *Proceedings of the Eighth International Symposium on Practical Aspects of Declarative languages*, volume 3819 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2006.
- I. MacLarty, Z. Somogyi, and M. Brown. Divide-and-query and subterm dependency tracking in the Mercury declarative debugger. In *AADEBUG’05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 59–68. ACM Press, 2005.
- J. Maessen. Eager Haskell: Resource-bounded execution yields efficient iteration. In M. Chakravarty, editor, *Proceedings of the Haskell workshop*, pages 38–50. ACM Press, 2002.
- R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

- L. Naish. A three-valued semantics for logic programmers. *Theory and Practice of Logic Programming*, 6(5):509–538, 2006.
- L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.
- L. Naish. A three-valued declarative debugging scheme. *Australian Computer Science Communications*, 22(1):166–173, January 2000.
- L. Naish and T. Barbour. A declarative debugger for a logical-functional language. In Graham Forsyth and Moonis Ali, editors, *Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers*, volume 2, pages 91–99, Melbourne, 1995. DSTO General Document 51.
- L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.
- N. Nethercote and A. Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, pages 44–55. ACM Press, 2002.
- H. Nilsson. Tracing piece by piece: Affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 36–47. ACM Press, 1999.
- H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.
- H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköpings Universitet, Sweden, 1998.

BIBLIOGRAPHY

- H. Nilsson and P. Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 117–134. Springer, 1993.
- H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, 1997.
- NIST. The economic impacts of inadequate infrastructure for software testing. www.nist.gov/director/prog-ofc/report02-3.pdf, May 2002.
- A. Pang, D. Stewart, S. Seefried, and M. Chakravarty. Plugging Haskell in. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM Press, 2004.
- W. Partain. The NoFib benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, editors, *Functional Programming (Proceedings of the 1992 Glasgow workshop on functional programming)*, pages 195–202. Springer-Verlag, 1993.
- L. Pereira. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, pages 203–210. Springer-Verlag, June 1986.
- S. Peyton Jones. Wearing the hair shirt: A retrospective on Haskell. Invited talk at POPL 2003. research.microsoft.com/~simonpj, 2003.
- S. Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, 2002.
- S. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, M. Broy, and R. Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001.
- S. Peyton Jones. Implementing lazy functional languages on stock hardware: The

- Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
- S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, 1986.
- S. Peyton Jones and P. Wadler. Imperative functional programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 71–84. ACM Press, 1993.
- S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, January 1996.
- S. Peyton Jones, M. Jones, and E. Meijer. Type classes: Exploring the design space. In *Proceedings of the 2nd ACM Haskell Workshop*, June 1997.
- S. Peyton Jones, A. Reid, F. Henderson, C. A. R. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36. ACM Press, 1999.
- S. Peyton Jones, P. Hudak, J. Hughes, and P. Wadler. The history of haskell (draft). www.haskell.org/haskellwiki/History_of_Haskell, 2006.
- B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- M. Pil. Dynamic types and type dependent functions. In K. Hammond, A. Davie, and C. Clack, editors, *Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer, 1998.
- R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman, 1993.
- B. Pope. Declarative Debugging with Buddha. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer-Verlag, 2005.

BIBLIOGRAPHY

- B. Pope. Buddha: A declarative debugger for Haskell. Technical Report 98/12, Department of Computer Science and Software Engineering, The University of Melbourne, 1998.
- B. Pope. A tour of the Haskell Prelude. www.haskell.org/bookshelf, 2001.
- B. Pope and L. Naish. A program transformation for debugging Haskell-98. *Australian Computer Science Communications*, 25(1):227–236, 2003a.
- B. Pope and L. Naish. Practical aspects of declarative debugging in Haskell-98. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240. ACM Press, 2003b.
- B. Pope and L. Naish. Specialisation of higher-order functions for debugging. In M. Hanus, editor, *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.
- C. Reinke. Ghood - graphical visualisation and animation of Haskell object observations. In *Proceedings of the 5th ACM SIGPLAN Haskell Workshop*, volume 59 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
- C. Runciman and N. Rojemo. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 6(4):587–620, 1996.
- C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–246, 1993.
- A. Sabry. What is a purely functional language? *Journal of Functional Programming*, 8(1):1–22, 1998.
- P. Samson and S. Peyton Jones. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(1):334–385, 1997.

- M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.
- E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
- F. Shapiro. Etymology of the computer bug: History and folklore. *American Speech*, 62(4):376–378, 1987.
- Z. Somogyi. Idempotent I/O for safe time travel. In *Proceedings of the Fifth International Workshop on Automated Debugging*, pages 13–24, Belgium, September 2003.
- J. Sparud. *Tracing and Debugging Lazy Functional Computations*. PhD thesis, Chalmers University of Technology, Sweden, 1999.
- L. Sterling and E. Shapiro, editors. *The Art of Prolog*. MIT Press, 1986.
- P. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proceedings of the ACM SIGPLAN 2003 Haskell Workshop*, pages 72–83. ACM Press, 2003.
- G. Tremblay. Lenient evaluation is neither strict nor lazy. *Computer Languages*, 26(1):43–66, 2001.
- D. Turner. Miranda: A non-strict language with polymorphic types. In *Symposium on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.
- D. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.
- P. Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architectures, Lecture Notes in Computer Science 201*, pages 113–128. Springer-Verlag, 1985.

BIBLIOGRAPHY

- P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.
- P. Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.
- P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.
- M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: A new Hat. In R. Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, 2001.
- R. Watson. *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, Southern Cross University, New South Wales, Australia, 1997.
- W. Wayt Gibbs. Software’s chronic crisis. *Scientific American*, pages 72–81, September 1994.
- A. Zeller, editor. *Why Does My Program Fail?* Morgan Kaufmann, California, August 2005.

Appendix A

An example transformed program

In this appendix we show the output of `buddha-trans` for the example program in Figure 3.5.

```
module Main_B where
import Prelude_B (fromInteger, fromRational)
import qualified Buddha as B
import Prelude_B
smain
  = B.con Main_B.v5
    (\ v6 ->
      ((#>>) v6 (sputStrLn v6 "Enter a number")
        ((#>>=) v6 (sgetline v6)
          (B.fe1
            (\ snum v7 ->
              (#>>) v6 (sputStrLn v6 "Enter base")
                ((#>>=) v6 (sgetline v6)
                  (B.fe1
                    (\ sbase v8 ->
                      sputStrLn v6 (sconvert v6 (sread v6 sbase)
                                            (sread v6 snum))))))))))

pconvert :: B.D (B.F Int (B.F Int [Char]))
sconvert :: B.D (Int -> Int -> [Char])
pconvert v20 = B.fe2 (\ v22 _ v23 v21 -> sconvert v20 v22 v23)
sconvert v24 sbase snumber
  = B.call Main_B.v19 [B.V sbase, B.V snumber] v24
    (\ v25 ->
      smymap v25 (ptoDigit v25)
        (sreverse v25
          (slastDigits v25 sbase (sprefixes v25 sbase snumber))))
```

```

ptoDigit :: B.D (B.F Int Char)
stoDigit :: B.D (Int -> Char)
ptoDigit v27 = B.fe1 (\ v29 v28 -> stoDigit v27 v29)
stoDigit v30 si
  = B.call Main_B.v26 [B.V si] v30
  (\ v31 ->
    (#!!) v31
      ((#++) v31 (Prelude_B.senumFromTo v31 '0' '9')
        (Prelude_B.senumFromTo v31 'a' 'z')) si)

pprefixes :: B.D (B.F Int (B.F Int [Int]))
sprefixes :: B.D (Int -> Int -> [Int])
pprefixes v33 = B.fe2 (\ v35 _ v36 v34 -> sprefixes v33 v35 v36)
sprefixes v37 sbase sn
  = B.call Main_B.v32 [B.V sbase, B.V sn] v37
  (\ v38 ->
    case (#<=) v38 sn 0 of
      True -> []
      False -> case sotherwise v38 of
        True -> (:) sn (sprefixes v38 sbase (sdiv v38 sn sbase)))

plastDigits :: B.D (B.F Int (B.F [Int] [Int]))
slastDigits :: B.D (Int -> [Int] -> [Int])
plastDigits v40
  = B.fe2 (\ v42 _ v43 v41 -> slastDigits v40 v42 v43)
slastDigits v44 sbase sxs
  = B.call Main_B.v39 [B.V sbase, B.V sxs] v44
  (\ v45 -> smymap v45 (B.fe1 (\ sx v46 -> smod v45 sbase sx)) sxs)

pmymap :: B.D (B.F (B.F a b) (B.F [a] [b]))
smymap :: B.D (B.F a b -> [a] -> [b])
pmymap v49 = B.fe2 (\ v51 _ v52 v50 -> smymap v49 v51 v52)
smymap v53 v3 v4
  = B.call Main_B.v48 [B.V v3, B.V v4] v53
  (\ v54 ->
    case (v3, v4) of
      (sf, []) -> []
      (sf, (sx : sxs)) -> (:) (B.ap sf sx v54) (smymap v54 sf sxs))

v0 = "Main.hs"
v48 = B.identFun "mymap" 28 1 Main_B.v0
v39 = B.identFun "lastDigits" 25 1 Main_B.v0
v32 = B.identFun "prefixes" 20 1 Main_B.v0
v26 = B.identFun "toDigit" 17 1 Main_B.v0
v19 = B.identFun "convert" 10 1 Main_B.v0
v5 = B.identFun "main" 3 1 Main_B.v0

```

Appendix B

Higher-order functions in the intensional style

In this appendix we show an alternative debugging session for the example from Section 3.4. In this version we transform the program to use the intensional style of printing higher-order values, whereas in the previous version we used the extensional style.

There are two main differences between the debugging sessions:

1. The shape of the EDT.
2. The way the arguments of `mymap` are printed.

Nonetheless, `buddha` returns the same buggy node in its diagnosis (in general there could be multiple buggy nodes in the EDT and it is possible that we will find different ones depending on which style of printing higher-order values is used).

To begin with we transform the program, but this time we use the ‘`-t intens`’ command line switch:

```

▷ buddha-trans -t intens Main.hs
buddha-trans 1.2.1: initialising
buddha-trans 1.2.1: transforming: Main.hs
buddha-trans 1.2.1: compiling
Chasing modules from: Main.hs
Compiling Main_B      ( ./Main_B.hs, ./Main_B.o )
Compiling Main        ( Main.hs, Main.o )
Linking ...
buddha-trans 1.2.1: done

```

Then we run the debuggee:

```

▷ ./Buddha/debug

Enter a number
1976
Enter base
10
0aaa

```

After the execution of the debuggee is complete we start debugging:

```

Welcome to buddha, version 1.2.1
A declarative debugger for Haskell
Copyright (C) 2004 - 2006 Bernie Pope
http://www.cs.mu.oz.au/~bjpop/buddha

Type h for help, q to quit

[0] <Main.hs:3:1> main
    result = { 0 -> (8,Right ()) }

```

First, we decide to draw the EDT:

```

buddha: draw edt

```

Figure B.1 contains the output from the draw command, as displayed by *dotty*. Compare this to Figure 3.6. Note that the nodes are numbered in the same way in each EDT. This is to be expected because nodes are numbered according to the order in which reductions take place, and the style of transformation does not alter

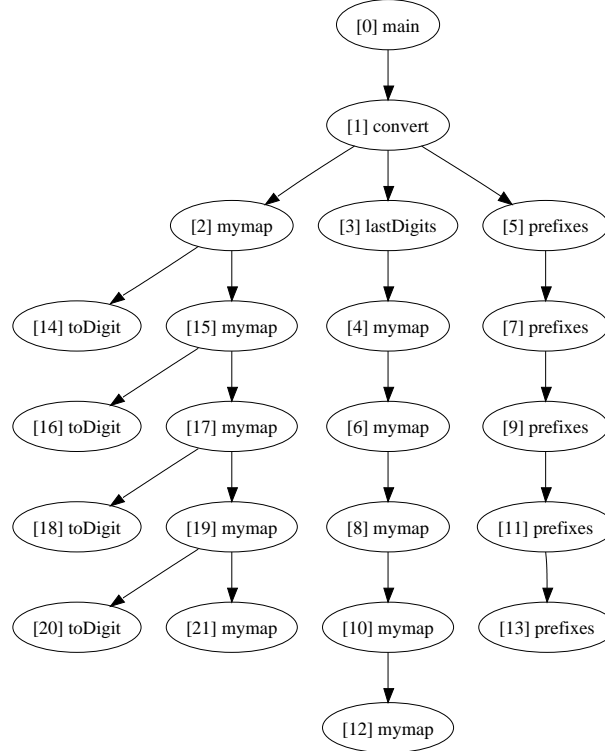


Figure B.1: An example EDT diagram produced by the ‘draw edt’ command.

the evaluation order of the program. Also note the position of the nodes for `toDigit` in each tree. In Figure 3.6, nodes for `toDigit` appear as children of the node for `convert` because higher-order instances of `toDigit` are printed in the extensional style. That means `toDigit` determines its parent at the point where it is mentioned by name, which is in the body of `convert`. In Figure B.1, nodes for `toDigit` appear as children of the left-branch of nodes for `mymap` because higher-order instances of `toDigit` are printed in the intensional style. That means `toDigit` determines its parent at the point where it is saturated, which is inside the body of `mymap`.

We decide to look ahead in the EDT by listing the children of the node for `main`:

buddha: *kids*

Children of node 0:

```
[1] <Main.hs:10:1> convert
    arg 1  = 10
    arg 2  = 1976
    result = ['0','a','a','a']
```

We decide to jump to the node for `convert`:

buddha: *jump 1*

We judge that derivation to be erroneous:

buddha: *erroneous*

Buddha automatically navigates to the first child of `convert`, which is an application of `mymap`:

```
[2] <Main.hs:28:1> mymap
    arg 1  = toDigit
    arg 2  = [0,10,10,10]
    result = ['0','a','a','a']
```

Note that the argument of `mymap` is now printed in the intensional style. We judge this derivation to be correct:

buddha: *correct*

Buddha then moves to the next child of `convert`, which is an application of `lastDigits`:

```
[3] <Main.hs:25:1> lastDigits
    arg 1  = 10
    arg 2  = [1976,197,19,1]
    result = [10,10,10,0]
```

This is erroneous:

buddha: *erroneous*

Buddha automatically moves to the first (and only) child of `lastDigits` which is another application of `mymap`:

```
[4] <Main.hs:28:1> mymap
    arg 1  = (<Main.hs:25:30> lambda)
    arg 2  = [1976,197,19,1]
    result = [10,10,10,0]
```

Note that the first argument to `map` is a lambda function, which `buddha` prints using its source code coordinates. We look it up in the program and find that it is `'\x -> mod base x'`. After some careful consideration we decide that this is correct:

buddha: *correct*

We have found a buggy node, so `buddha` makes a diagnosis:

```
Found a bug:
[3] <Main.hs:25:1> lastDigits
    arg 1  = 10
    arg 2  = [1976,197,19,1]
    result = [10,10,10,0]
```