Hardware-Assisted and Target-Directed Evaluation of Functional Programs

Matthew Francis Naylor

Department of Computer Science
University of York

A dissertation submitted for the degree
Doctor of Philosophy

September 2008

Abstract

A new reduction machine for lazy functional languages called the Reduceron is proposed. The Reduceron exploits wide, parallel memories to increase evaluation speed, and is prototyped in programmable hardware. It is compared with conventional machines by a theoretical analysis and by an experimental comparison. It is shown that the use of wide, parallel memories in a graph reduction machine can lead to a factor of five performance improvement.

Existing approaches to property-based testing are reviewed, and two new ones – Reach and Lazy SmallCheck – are developed. Reach is a program analyser that targets evaluation of expressions left uncovered by existing testing techniques, and Lazy SmallCheck is a lightweight library for a standard lazy functional language. Both are more effective than existing approaches at testing properties that have restrictive antecedents.

Contents

1 Introduction **15**
1.1 Hardware-assisted evaluation 16
1.2 Target-directed evaluation 18

2 Hardware-Assisted Evaluation **20**
2.1 Introduction 20
2.2 Background 21
2.2.1 A core functional language 22
2.2.2 A simpler core functional language 23
2.2.3 Evaluation 24
2.2.4 Graph reduction 26
2.2.5 Implementing graph reduction 27
2.3 Template instantiation or the G-machine? 27
2.4 Syntax 28
2.4.1 Dealing with strict primitives 28
2.4.2 Bytecode 29
2.5 Semantics 31
2.5.1 A primitive evaluator 32
2.5.2 Semantic definition 32
2.6 The Narrow Reduceron 35
2.6.1 Memory 36
2.6.2 Register-transfer notation 37
2.6.3 Register-transfer model 38
2.6.4 Discussion 42
2.7 The Wide Reduceron 43
2.7.1 Memory 43
2.7.2 Register-transfer model 44

2.7.3 Discussion 48
2.8 Implementation on FPGA 48
2.8.1 Word size 49
2.8.2 Block RAMs 49
2.8.3 Constructing large memories from small ones 49
2.8.4 Quad-word memories 51
2.8.5 Memory layout 53
2.8.6 Garbage collection 53
2.8.7 Description language 54
2.8.8 Interface to the FPGA 54
2.8.9 Resource usage 54
2.8.10 Minor variations 55
2.9 Performance 55
2.9.1 Conclusions 56
2.10 Limitations and future work 59
2.10.1 Memory capacity 59
2.10.2 Clock frequency 59
2.10.3 Storage efficiency versus logic efficiency 60
2.10.4 Indirections 60
2.10.5 Template instantiation versus the G-machine 63
2.10.6 Case expressions 64
2.10.7 Memory utilisation 65
2.11 Related work 66
2.11.1 Hardware reduction machines 66
2.11.2 The Big Word Machine 66
2.11.3 Ward's work 67
2.11.4 The statically allocated functional language 68

3 The Essence of Circuit Description 70
3.1 Introduction 70
3.2 VHDL 72
3.2.1 Structural description 72
3.2.2 Behavioural description 73
3.3 Lava: a pure structural language 76
3.4 Recipe: a library for behavioural description 78
3.4.1 Some useful components 79

3.4.2 Monads . . . 81
3.4.3 The Recipe monad . . . 82
3.4.4 Skip and tick . . . 83
3.4.5 Choice . . . 84
3.4.6 Iteration . . . 84
3.4.7 Parallel composition . . . 85
3.4.8 Mutable variables . . . 86
3.4.9 Following recipes . . . 88
3.5 Application 1: a sequential multiplier . . . 89
3.5.1 Numbers in Lava . . . 89
3.5.2 Implementation . . . 89
3.5.3 Correctness . . . 90
3.6 Application 2: a Lego brick-sorter . . . 90
3.6.1 Structure of the brick-sorter . . . 92
3.6.2 Operation of the brick-sorter . . . 92
3.6.3 Implementation . . . 93
3.7 Application 3: a simple stack processor . . . 94
3.7.1 Bytecode . . . 94
3.7.2 Memory . . . 95
3.7.3 Processor . . . 97
3.7.4 Correctness . . . 99
3.8 Discussion . . . 99
3.8.1 Implementation cost . . . 99
3.8.2 Testing . . . 99
3.8.3 Efficiency . . . 100
3.8.4 Static versus dynamic typing . . . 100
3.8.5 Flattening . . . 101
3.8.6 Advanced synthesis . . . 101
3.9 Related work . . . 102
3.9.1 Claessen and Pace . . . 102
3.9.2 Handel-C . . . 102
3.9.3 SPARK . . . 104
3.10 Summary . . . 105

4 Target-Directed Evaluation 106
4.1 Introduction . . . 106

4.1.1 A motivating example . . . 107
4.1.2 Testing with QuickCheck and HPC . . . 108
4.1.3 Program coverage with Reach . . . 109
4.1.4 Direct refutation with Reach . . . 109
4.1.5 Chapter outline . . . 110
4.2 Syntax and semantics . . . 110
4.2.1 Syntax . . . 110
4.2.2 Semantics . . . 111
4.3 Basic Reach . . . 112
4.3.1 Extending the semantics . . . 112
4.3.2 Full normal form . . . 113
4.3.3 Enumerating the domain . . . 114
4.3.4 Bounding recursion . . . 114
4.3.5 Definition of Basic Reach . . . 115
4.4 Forward Reach . . . 115
4.4.1 Extensions to Basic Reach . . . 116
4.4.2 An example derivation . . . 117
4.4.3 Enumerating the domain . . . 119
4.4.4 Definition of Forward Reach . . . 120
4.5 Backward Reach . . . 120
4.5.1 Overview . . . 121
4.5.2 The target context . . . 121
4.5.3 The six rewrite rules . . . 122
4.5.4 Unification . . . 126
4.5.5 Unification assumptions . . . 127
4.5.6 An example derivation . . . 128
4.5.7 Initial inlining . . . 131
4.5.8 Definition of Backward Reach . . . 131
4.6 Implementation . . . 133
4.7 Comparison . . . 133
4.7.1 Binary search trees . . . 137
4.7.2 Red-Black trees . . . 140
4.7.3 A digital multiplexor . . . 143
4.7.4 Huffman compression . . . 145
4.7.5 Turner's abstraction algorithm . . . 147
4.8 Conclusions . . . 149

4.9 Related work . . . 151
4.9.1 Functional-logic programming in Curry . . . 151
4.9.2 Property-directed generation of test-data . . . 152
4.9.3 Specification-based testing of Java programs . . . 153
4.9.4 Static checking and theorem proving . . . 154
4.10 Limitations and future work . . . 156
4.10.1 Higher-order functions . . . 156
4.10.2 Garbage collecting the target . . . 157
4.10.3 Initial inlining . . . 157
4.10.4 Parallel conjunction . . . 158
4.10.5 Arithmetic constraint solvers . . . 158
4.10.6 Efficiency of implementation . . . 159
4.10.7 Generalisations . . . 159

5 A Library for Demand-Driven Testing 160
5.1 Introduction . . . 160
5.1.1 SmallCheck . . . 161
5.1.2 Lazy SmallCheck . . . 162
5.1.3 Structure of this chapter . . . 162
5.2 QuickCheck: a review . . . 162
5.2.1 Arbitrary types and testable properties . . . 162
5.2.2 Generators for user-defined types . . . 163
5.2.3 Conditional properties . . . 164
5.2.4 Counter examples . . . 166
5.3 SmallCheck: a review . . . 166
5.3.1 Small values . . . 166
5.3.2 Serial types . . . 167
5.3.3 Test coverage and counter examples . . . 168
5.3.4 Existential properties . . . 169
5.3.5 Conditional properties . . . 170
5.4 Lazy SmallCheck . . . 170
5.4.1 Implication . . . 171
5.4.2 Laziness is delicate . . . 172
5.4.3 Parallel conjunction . . . 173
5.4.4 Strict properties . . . 174
5.4.5 Serial types . . . 175

5.5 Implementation . . . 176
5.5.1 Partially-defined inputs . . . 176
5.5.2 Answers . . . 177
5.5.3 Refinement . . . 178
5.5.4 Series combinators . . . 179
5.5.5 Refutation algorithm . . . 179
5.5.6 Parallel conjunction . . . 181
5.5.7 Variations . . . 181
5.6 Comparative evaluation . . . 182
5.6.1 RedBlack . . . 182
5.6.2 Huffman . . . 184
5.6.3 Mate . . . 184
5.6.4 Reduceron instantiator . . . 185
5.6.5 Other examples . . . 187
5.6.6 Summary of results . . . 187
5.7 Related work . . . 188
5.7.1 Needed narrowing . . . 188
5.7.2 Residuation . . . 188
5.7.3 Gast . . . 189
5.7.4 EasyCheck . . . 189
5.8 Limitations and future work . . . 189
5.9 Conclusions . . . 190

6 Conclusions **192**

A Implementation of Recipe **194**

B Implementation of Lazy SmallCheck **198**

List of Figures

2.1 Syntax of Yhc Core. 22
2.2 An example series of reductions. 25
2.3 The syntax of nodes in Reduceron bytecode. 29
2.4 Helper functions for bytecode nodes. 29
2.5 Bytecode for the example functions. 30
2.6 Transition rules for the Reduceron. 33
2.7 Definitions of *inst* and *unwind*. 33
2.8 Registers used by the Narrow Reduceron. 36
2.9 Syntax of register-transfer statements. 37
2.10 Clock-accurate model of primitive evaluation (Narrow). . . . 39
2.11 Clock-accurate model of unwinding (Narrow). 40
2.12 Clock-accurate model of unfolding (Narrow). 41
2.13 Clock-accurate model of instantiation (Narrow). 41
2.14 Additional registers used by Wide Reduceron. 43
2.15 A frequently-used abstraction for reading memory. 44
2.16 Clock-accurate model of primitive evaluation (Wide). 45
2.17 Clock-accurate model of unwinding (Wide). 46
2.18 Clock-accurate model of unfolding (Wide). 47
2.19 Cascading block RAMs to form a larger RAM. 50
2.20 Circuit diagram for a quad-word memory. 52
2.21 Benchmark programs. 57
2.22 Example functions after a widening transformation. 66

3.1 Schematic of a four-input OR-tree. 73
3.2 Structural architecture of an OR-tree in VHDL. 74
3.3 Behavioural description of a sequential multiplier in VHDL. . 75
3.4 Operations of Lava's `Bit` ADT. 76

3.5 Schematic of a sequential parity checker. 77
3.6 Lava's `row` pattern. 79
3.7 Delay with input enable and a set-reset latch. 80
3.8 An example of `do`-notation. 83
3.9 Parallel composition. 86
3.10 Variable creation. 88
3.11 Addition of bit-lists in Lava. 91
3.12 Sequential multiplier in Recipe. 91
3.13 End-elevation view of the brick-sorter. 92
3.14 Controller for a Lego brick-sorter in Recipe. 93
3.15 Compiler and machine for evaluating polynomial expressions. 95
3.16 Routines for decoding op-codes. 96
3.17 Interface to memory. 96
3.18 Poly's register file. 97
3.19 Stack processor for evaluating polynomials in Recipe. 98
3.20 Sequential multiplier in Handel-C. 103

4.1 Partial output of HPC on the motivating example. 108
4.2 Syntax of the core functional language. 111
4.3 An example program in abstract syntax. 117
4.4 Non-deterministic substitution. 125
4.5 Definition of *bot*. 125
4.6 Another example program in abstract syntax. 128
4.7 Inlining to reveal a target. 132
4.8 Graphical comparison of Basic and Forward Reach. 135
4.9 Graphical comparison of Forward and Backward Reach. . . . 136
4.10 A faulty insertion function for Red-Black trees. 140
4.11 Turner's abstraction algorithm. 148

5.1 A QuickCheck `Arbitrary` instance for `Prop`. 164
5.2 Lazy SmallCheck's `Series` combinators. 180
5.3 Template instantiation function, from the Reduceron. 186

List of Tables

2.1 Clock-cycles taken by each Reduceron transition. 48
2.2 Memory layout of the Wide Reduceron. 53
2.3 Reduceron synthesis results on the Virtex-II. 54
2.4 Numbers of clock-cycles needed to execute several programs. 57
2.5 Timings of programs running on various implementations. . . 58
2.6 Profiles of programs running on the Wide Reduceron. 59

4.1 Timimgs of Basic and Forward Reach. 134
4.2 Timings of Forward and Backward Reach. 135
4.3 Effect of different conjunct orderings. 138
4.4 Comparing Forward and Backward Reach on RedBlack_2. . . . 142
4.5 Comparing Forward and Backward Reach on Mux_2. 144

5.1 Timings of SmallCheck and Lazy SmallCheck. 183

Acknowledgements

My supervisor, Colin Runciman, has been a great source of encouragement. No matter how depressed I felt going into a supervisor meeting, I always left with a great feeling of enthusiasm! Not only did Colin allow me to work on some of his ideas, but he keenly assisted me when I pursued some of my own. I found Colin's excitement about my work to be very rewarding.

Many in the real-time systems group, where I spent my first year, also helped me. In particular, Neil Audsley supervised me in my first year and sparked my interest in FPGAs. Jack Whitham and Ian Gray always made themselves available to answer my FPGA questions. Andy Wellings ensured my move to the programming group proceeded with minimum fuss and upset.

In the programming group, I felt at home. Neil Mitchell had a fearless attitude to research that made the PhD seem not so scary, and he contributed lots of valuable code and suggestions to my work. Detlef Plump and Malcolm Wallace so often invited me to join them for lunch.

Following advice from Colin, I visited Chalmers University in Gothenburg for a month. Koen Claessen and Mary Sheeran helped arrange my visit, and made me feel very welcome when I arrived. They introduced me to Emil Axelsson and Fredrik Lindblad, who both subsequently made significant contributions to my work, and with whom I really enjoyed working. Emil looked after me, introduced me to his friends and family, and took me to beautiful locations, including two amazing boat trips around the archipelago.

Many others perked me up along the way, particularly Matt Day, Lee McSorley, Rebecca Steinitz, Jan Tobias Mühlberg, Kathrin Staiger, Shailesh Pandey and many at Heslington Church. Always supporting me from afar was my family, especially Mum, Dad, Freda, John, and Suzanne.

Declaration

Chapters 2 to 5 are based on the following published papers, respectively.

2. Matthew Naylor and Colin Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In *IFL '07: Implementation and Application of Functional Languages (Revised Selected Papers)*, pages 129-146. Springer LNCS, 2008.

3. Matthew Naylor. A Recipe for Controlling Lego using Lava. In *The Monad.Reader*, issue 7, edited by Wouter Swierstra, pages 5-21, 2007.

4. Matthew Naylor and Colin Runciman. Finding inputs that reach a target expression. In *SCAM '07: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 133-142, IEEE Computer Society, 2007.

5. Colin Runciman, Matthew Naylor and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *ACM SIGPLAN 2008 Haskell Symposium*, page to appear, ACM, 2008.

Sections 5.1, 5.2, 5.3, and 5.4.3 of this thesis are based on text written mostly by my coauthors. Sections 1.1, 1.2, 5.6, and 5.7, are written mostly by me, but are in part based on text written by my coauthors. All other sections are written entirely by me. My contribution to the work described in Chapters 2 to 5 is as follows.

2. All of the work was conceived and carried out by me.

3. All of the work was conceived and carried out by me.

4. The problem statement and a rough sketch of the backward analysis

were done by Colin. I formalised Colin's sketch, and carried out all of the implementation and comparative work.

5. The idea to apply properties to partially-defined inputs and use the results to prune input-spaces came from me. Fredrik developed the first working prototype, including the idea to encode positions in the undefined parts of the input. The final implementation and comparative work were done mostly by me.

Chapter 1

Introduction

The importance of functional languages has for some time remained largely in academia, where they serve as a strong semantic foundation for studying new ideas [41] and teaching core programming principles [27]. But now they are becoming increasingly important in industry too, not only due to growing commercial use [25], but also because of their influence on other languages both mainstream [71, 38, 23] and new [24, 1, 74].

One of the most attractive features of functional languages is that they allow programs to be written at a very high level of abstraction. This feature is partly due to the fact that functional languages have evolved much more from the desire to allow humans to express algorithms concisely than from the architectural constraints of the machines on which programs must run. The price to pay is that functional programs typically run less efficiently on conventional computers than their more machine-oriented counterparts.

Functional language researchers have studied many aspects of software technology, from profilers and tracers to type systems and theorem provers. But, for some time, there has been one rather notable omission – *software testing* – even though it accounts for over 50% of the total cost of software development. Although functional language researchers have recently made significant progress in this area [20], so much so in fact that they have already influenced how testing can be done in a number of different languages [4, 69, 11, 58, 68, 11, 46, 73], there is much scope for further development. In particular, existing automatic test generators sometimes leave parts of

programs untested, and in such cases programmers must resort to writing test generators by hand.

These observations motivate the two main topics of this thesis: (1) the development of a special-purpose machine to evaluate functional programs efficiently; and (2) a program analyser that targets evaluation of expressions left uncovered by existing automatic testing techniques. Both topics are concerned with *evaluation*: the former with *hardware-assisted* evaluation and the latter with *target-directed* evaluation.

The lazy functional language Haskell [79] is used as a host for many of the ideas presented, but only a basic knowledge of pure functional programming is assumed.

1.1 Hardware-assisted evaluation

Efficient evaluation of high-level functional programs on conventional computers is a big challenge. Sophisticated compilation techniques are needed to exploit architectural features designed for low-level imperative execution. Not only do such techniques result in complex compilers, but they can easily become outdated with architectural advances – some techniques that made functional languages efficient fifteen years ago are now known to be inefficient for modern machines [62]. Most serious of all is that conventional computers have inherent limitations when it comes to running functional programs. In particular, memory bandwidth is limited to *serial* communication in *small units* – the *von Neumann bottleneck* [10]. Functional language evaluators based on *graph reduction* [78] perform intensive construction and deconstruction of expressions in memory. Each such operation requires sequential execution of many machine instructions, not because of any inherent data dependencies, but because of architectural constraints in conventional computers.

All this motivates the idea of building special-purpose computers to support functional languages, just as GPUs have been built to support graphics. This is not a new idea. In the '80s and '90s there was a 15-year ACM conference series *Functional Programming and Computer Architecture* covering the topic, and a major computer manufacturer built a graph-reduction

prototype [85]. But as the process of building exotic new machines was so expensive and time consuming, "much of this architecturally oriented work turned out to be a dead end" [42]. In the penultimate chapter of his thesis about a graph-reduction architecture called TIGRE [53], Koopman gets to the heart of the problem:

> *So, it is probably not worthwhile building a special-purpose CPU to support TIGRE, since current RISC technology will probably have increased in speed enough by the time a TIGRE chip could be designed and fabricated to make the exercise pointless.* [53]

Nowadays, the situation is quite different. *Field programmable gate arrays* (FPGAs) have greatly reduced the effort and expertise needed to develop special-purpose hardware. They contain thousands of parallel logic blocks that can be configured at will by software tools. Memory architectures of all kinds can be constructed from large arrays of independent block RAMs. As conventional PCs appear to have hit a limit in sequential processing speed [56], it is now special-purpose computing – through FPGAs – that can be viewed as the advancing technology. FPGAs can also be viewed as a tool for prototyping designs before they are converted to efficient *application-specific integrated circuits* (ASICs). Both of these views motivate fresh experiments in the design of special-purpose hardware to support functional languages.

Chapter 2 of this thesis presents one such experiment. The effect of *widening* the von Neumann bottleneck for sequential graph reduction is analysed. A special-purpose reduction machine – the *Reduceron* – is proposed, featuring wide, parallel memories. Wide and narrow variants of the Reduceron are prototyped on an FPGA, and compared with each other and with existing functional language implementations running on a PC.

To implement the Reduceron prototype, an unconventional approach to circuit description is taken. Instead of a language supporting both structural and behavioural description, a pure structural language is used, and behaviour is expressed using a small library of higher-order, pure structural components. In Chapter 3, this library – *Recipe* – is presented, and the capabilities and deficiencies of the approach are explored.

1.2 Target-directed evaluation

To argue that a program is correct is to argue that it satisfies some set of *properties*. A property could for example be that one function in the program computes the inverse of another; or that some function preserves the invariant of a data structure; or that an efficient-but-complex function produces the same result as a simple-but-inefficient one. In any case, properties state relationships between the component functions of a program, capturing precisely what it means for the program to be correct.

Claessen and Hughes propose an attractive approach to testing properties of Haskell programs [20], as implemented in their library *QuickCheck*. Properties relating the component functions of a program are specified in Haskell itself. The simplest kind of property is just a boolean-valued function whose arguments are interpreted as universally quantified variables. QuickCheck applies the properties to randomly selected inputs of the appropriate type, and reports any counter-examples found.

Specifying properties in QuickCheck forces programmers to think hard about what makes their programs correct, and to record their conclusions in a precise form. Even this preliminary outcome of exact documentation has value. But the big reward for specifying properties is that they can be *tested automatically*, perhaps revealing bugs.

Despite the success of QuickCheck – resulting in ports to many languages including Erlang [4], Java [69], Lisp [11], ML [58], Perl [68], Python [11], Ruby [46] and Scala [73] – it has limitations. Claessen and Hughes write:

> *The major limitation of QuickCheck is that there is no measurement of test coverage.* [20]

Gill and Runciman propose the use of Haskell Program Coverage (HPC) to address this issue [31]. While QuickCheck applies program properties to randomly chosen inputs, HPC records which expressions in the program have been evaluated, and which have not. After testing, HPC generates a marked-up version of the source code in which any unevaluated expressions are highlighted. Although repeated applications of QuickCheck typically decreases the number of highlighted expressions, some stubborn ones often remain. The uncovered expressions might be unreachable, but they

might also be the result of an insufficient distribution of tests generated by QuickCheck.

Chapter 4 of this thesis presents a program analysis – *Reach* – which targets evaluation of highlighted expressions in functional programs. Reach is presented as a series of incremental modifications to a basic lazy evaluator, resulting in variants based on both forward and backward analysis. The benefit offered by each variant is explored on a range of benchmark properties.

Being a program analyser allows great flexibility as the source code of a program can be processed in any desired way. But implementing a full-strength program analyser is a lot of work, comparable to that of writing a compiler. Chapter 5 presents a more lightweight approach, using the lessons learned in Chapter 4 to develop a new Haskell library for property-based testing called *Lazy SmallCheck*. In-principle and quantitative comparisons with the existing libraries QuickCheck and SmallCheck are given.

Finally, Chapter 6 concludes the thesis, highlighting the main discoveries made in the earlier chapters.

Chapter 2

Hardware-Assisted Evaluation

2.1 Introduction

The processing power of PCs has risen astonishingly over the past few decades, and this trend looks set to continue with the introduction of multi-core CPUs. However, increased processing power does not necessarily mean faster programs! Many programs, particularly *memory intensive* ones, are limited by the rate that data can travel between the CPU and the memory, not by the rate that the CPU can process data. Backus called this limiting factor the *von Neumann bottleneck* [10].

A prime example of a memory intensive application is *graph reduction* [78], the operational basis of standard lazy functional language implementations. The core operation of graph reduction is *function unfolding*, whereby a function application $f\ a_1 \cdots a_n$ is reduced to a fresh copy of f's body with its variables replaced by the arguments $a_1 \cdots a_n$. On a PC, unfolding a single function in this way requires the sequential execution of *many* machine instructions. This sequentialisation is merely a consequence of the PC's von Neumann architecture, *not* of any data dependencies in the reduction process.

This chapter explores the effect of *widening* the von Neumann bottleneck for sequential graph reduction. A special-purpose machine – the Reduc-

eron – is proposed, exploiting wide, parallel memories to increase reduction speed. The Reduceron is prototyped on a field programmable gate array (FPGA). Modern FPGAs contain hundreds of independent memory units called *block RAMs*, each of which can be accessed in parallel. The Reduceron *cascades* these block RAMs to form separate dual-port, quad-word memories for stack, heap and code storage, meaning that up to eight words can be transferred between two memories in a single clock-cycle. Together with vectorised processing logic, the wide, parallel memories allow the Reduceron to rapidly execute the block read-modify-write memory operations that lie at the heart of function unfolding.

This chapter is structured as follows. Section 2.2 introduces background material, including graph reduction and a core functional language. Section 2.3 explains the decision to base the Reduceron on a graph reduction technique called *template instantiation*. Section 2.4 defines the bytecode that the Reduceron executes. Section 2.5 presents a small-step operational semantics of the Reduceron, highlighting the parts of the graph-reduction process that can be executed in parallel. Sections 2.6 and 2.7 refine the semantics to two clock-accurate models of the Reduceron: *Narrow* and *Wide*. The Wide Reduceron exploits wide, parallel memories and the Narrow Reduceron does not. The numbers of clock-cycles consumed by the two models are precisely stated and justified. Section 2.8 outlines the implementation of the clock-accurate models on FPGA. In Section 2.9, the Narrow and Wide Reduceron implementations running on an FPGA are compared with each other and with several existing Haskell implementations running on a PC. Section 2.10 suggests possible improvements to the Reduceron and Section 2.11 discusses related work.

2.2 Background

This section reviews a simple *core functional language* to which Haskell programs can be compiled, and the technique of *graph reduction* to evaluate programs. As running examples of Haskell functions, consider the factorial function

```
fact :: Int -> Int
fact n = if n == 1 then 1 else n * fact (n-1)
```

p	::=	$\vec{d}$	(program)
d	::=	$c\ \vec{v}$ `=` e	(function definition)
e	::=	v	(variable)
	|	c	(constant)
	|	$e\ e$	(application)
	|	`case` e `of` $\vec{a}$	(case expression)
	|	`let` $\vec{b}$ `in` e	(let expression)
a	::=	$c\ \vec{v}\ \mapsto\ e$	(case alternative)
b	::=	v `=` e	(let binding)

Figure 2.1: Syntax of Yhc Core.

and the following list-reversing function which uses an accumulator.

```
data List a = Nil | Cons a (List a)

rev :: List a -> List a -> List a
rev Nil acc = acc
rev (Cons x xs) acc = rev xs (Cons x acc)
```

2.2.1 A core functional language

One of the first steps performed by most Haskell compilers is to translate Haskell into a core functional language. The core language is typically much simpler than Haskell, and makes compilation to low-level machine code a much easier task.

To illustrate, Figure 2.1 defines the syntax of *Yhc Core* [32], the core language used in the York Haskell Compiler [90]. The meta-variable v ranges over variable names, c over constants, e over expressions, d over function definitions, and p over programs. A constant can be a function name, a constructor name, or a primitive literal such as a character or an integer. Lists of meta-variables are denoted with an overhead arrow, for example $\vec{v}$ denotes a list of variable names.

Yhc Core provides *fewer* constructs than full Haskell. Most notably, it does not provide lambda expressions. Lambda expressions can be translated to

function definitions using a technique called *lambda-lifting* [44, 49].

Furthermore, Yhc Core provides *simpler* constructs than full Haskell. Patterns in case alternatives are exactly one constructor deep whereas in Haskell they may be any number of constructors deep. Case expressions are exhaustive whereas in Haskell incomplete matching is allowed. Function arguments and the left-hand-sides of let bindings can only be variables whereas in Haskell they may be arbitrary patterns. For details on how to translate Haskell to a core language like Yhc Core, see [78].

In the abstract syntax of Yhc Core, the running examples look as follows.

```
fact n = case (==) n 1 of
           [False ↦ (*) n (fact ((-) n 1)), True ↦ 1]

rev v acc = case v of
              [Nil ↦ acc, Cons x xs ↦ rev xs (Cons x acc)]
```

Note that infix applications have been made prefix, `if` has been desugared to a `case`, and equational pattern matching has been moved to a `case` expression.

2.2.2 A simpler core functional language

Recently Jansen has observed that one can use a core language even simpler than the one defined in Figure 2.1, and still implement an efficient evaluator for the language [48]. This is achieved by encoding data constructors and case expressions as function definitions and applications respectively. Jansen's method comprises two steps. First, for each constructor c_i of a data type with n constructors $c_1 \cdots c_n$, a function definition

$$c_i\ v_1\ \cdots\ v_{\#c_i}\ w_1\ \cdots\ w_n\ =\ w_i\ v_1\ \cdots\ v_{\#c_i}$$

is introduced, where $\#c$ denotes the number of arguments taken by the constructor c. In other words, each original data constructor c_i is encoded as a function that takes as arguments the $\#c_i$ arguments of c_i *and* n continuations stating how to proceed depending on the constructor's value. To illustrate, the `False` and `True` constructors of the `Bool` data type,

```
data Bool = False | True
```

and the `Nil` and `Cons` constructors of the `List` data type are encoded using the following functions.

```
False f t      = f
True  f t      = t
Nil   n c      = n
Cons  x xs n c = c x xs
```

In the second step of Jansen's method, each case expression of the form

$$\texttt{case}\ e\ \texttt{of}\ [c_1\ v_1\ \cdots\ v_{\#c_1}\ \mapsto\ e_1,\ \cdots,\ c_n\ v_1\ \cdots\ v_{\#c_n}\ \mapsto\ e_n]$$

is transformed to an application

$$e\ (\lambda v_1\ \cdots\ v_{\#c_1}\ \mapsto\ e_1)\ \cdots\ (\lambda v_1\ \cdots\ v_{\#c_n}\ \mapsto\ e_n)$$

Note that the resulting application contains lambda abstractions. These can be removed using a lambda-lifter. To illustrate, after applying Jansen's method, the running examples look as follows.

```
fact n = (==) n 1 ((*) n (fact ((-) n 1))) 1
rev v acc = v acc (rev' acc)
rev' acc x xs = rev xs (Cons x acc)
```

Note that `rev'` has been introduced by the lambda-lifter. No data constructors or case expressions remain.

2.2.3 Evaluation

In general, an expression e can be *reduced* if it contains a sub-expression of the form $f\ e_1 \cdots e_n$ where f is defined by the equation $f\ v_1 \cdots v_n$ = rhs. In such a situation, $f\ e_1 \cdots e_n$ is said to be a reducible expression, or a *redex* for short. To reduce e, the redex $f\ e_1 \cdots e_n$ occurring in e is replaced with the expression obtained by substituting $e_1 \cdots e_n$ for $v_1 \cdots v_n$ in rhs.

To illustrate, Figure 2.2 shows a series of reductions of the expression

```
rev (Cons 0 (Cons 1 Nil)) Nil
```

When an expression contains no redexes, like the final expression Figure 2.2, it is said to be in *normal form*.

```
   rev (Cons 0 (Cons 1 Nil)) Nil
= (Cons 0 (Cons 1 Nil)) Nil (rev' Nil)
= rev' Nil 0 (Cons 1 Nil)
= rev (Cons 1 Nil) (Cons 0 Nil)
= (Cons 1 Nil) (Cons 0 Nil) (rev' (Cons 0 Nil))
= rev' (Cons 0 Nil) 1 Nil
= rev Nil (Cons 1 (Cons 0 Nil))
= Nil (Cons 1 (Cons 0 Nil)) (rev' (Cons 1 (Cons 0 Nil)))
= Cons 1 (Cons 0 Nil)
```

Figure 2.2: An example series of reductions.

The expression under evaluation may contain more than one redex, and the order in which redexes are reduced can have a significant impact on the number of reductions needed to reach a normal form. There are two main strategies for choosing a redex.

1. Choose the innermost redex, giving *strict* evaluation.

2. Choose the outermost redex, giving *non-strict* evaluation.

One of the attractive properties of non-strict evaluation is that it always yields a normal form if strict evaluation does, and sometimes if strict evaluation does not. Indeed, it is sometimes referred to as *normal order reduction* due to this property. To illustrate, suppose that the expression `f 0` e is to be evaluated where e is a computationally expensive redex and `f x y = x`. Here, the outermost redex is `f 0` e and the innermost redex is e. In non-strict evaluation, `f 0` e is evaluated first to give the normal form `0` in just one reduction step. In strict evaluation, e must be evaluated before `f` can be applied, resulting in many more reduction steps to yield the same normal form. Or worse, evaluation of e might not terminate.

On the other hand, one of the attractive properties of strict evaluation is that it never evaluates the same argument more than once. Non-strict evaluators do not necessarily have this property. To illustrate, suppose that the expression `double` e is to be evaluated where e is a computationally expensive redex and `double` is defined by the following equation.

```
double x = (+) x x
```

The outermost redex is `double` e and the innermost redex is e. In non-strict

evaluation `double` e is evaluated first, and reduces to `(+)` e e. Here, there is the danger that e will be computed twice. In strict evaluation there is no such danger because e is reduced to normal form before being passed to `double`.

Nevertheless, there is a way to perform non-strict evaluation such that no function argument is evaluated more than once. It is called *lazy evaluation*, but how can it be implemented?

2.2.4 Graph reduction

One possible representation of an expression is as a *string* of symbols, just as one would write it on paper. Consider an expression containing a redex f e. In string reduction, this expression is reduced by replacing the redex with an *instance* of f. An instance of f is created by

1. creating a copy of the body of f, rearranging it to make gaps for e,
2. copying e into each gap.

If the argument is referred to n times in the body of f, then e is copied n times.

A more efficient representation of an expression is as a *graph*. In graph reduction, an expression is a *pointer* to a node containing sub-expressions. Many expressions can point to the same node. Now an instance of f is created by

1. creating a copy of the body of f,
2. overwriting each argument with (the pointer) e.

The actual structure pointed to by e is *never copied*. Furthermore, if e is ever evaluated, the node it points to can be updated with an indirection to the result, so that e is never evaluated again. This updating gives lazy evaluation.

2.2.5 Implementing graph reduction

In [80], Peyton Jones describes two main approaches to implementing graph-reduction: an interpreter-based approach called *template instantiation*, and a compiler-based approach based on the *G-machine* (invented by Johnsson and Augustsson [49, 7]).

In template instantiation, function definitions are stored in *template memory*, and the expression being reduced in *graph memory*. When a function is applied, the function's body is read from template memory and an instance of it is created in graph memory. Peyton Jones describes this as "the simplest possible implementation of a functional language" [80].

In the G-machine, template memory is not used. Before a program is run, each function definition is translated to a sequence of G-machine instructions which, when executed, create an instance of the function body directly in graph memory. Each G-machine instruction can in turn be translated to a sequence of regular CPU instructions and executed by a PC.

To see the crucial difference between the two approaches, consider the operation of template instantiation from the viewpoint of a conventional PC.

1. Fetch CPU instructions from memory which, when executed,
2. read the function body from template memory, and
3. construct an instance of the function body in graph memory.

The operation of the G-machine is the same *except that step 2 is skipped*. As Peyton Jones writes, "there are no instructions required to traverse the template" [80].

2.3 Template instantiation or the G-machine?

Should the Reduceron be based on template instantiation or the G-machine? Indeed template instantiation implemented on a conventional PC is not very efficient due to interpretive overhead: CPU instructions must be fetched which, when executed, fetch and execute the functions residing in template memory. But if implemented as a special-purpose machine, this interpretive

overhead can be removed: a special-purpose machine can read directly from template memory without having to fetch and execute any CPU instructions.

Taking into account the above observation and that template instantiation is simpler than the G-machine, the Reduceron is based on template instantiation. This decision is reviewed in Section 2.10.5.

2.4 Syntax

This section introduces a syntax for the Reduceron *bytecode*. The bytecode of a program is an encoding of a core functional program as a sequence of words. The core functional language used is Yhc Core, defined in Figure 2.1, with two transformations applied to it. The first transformation is to remove case expressions using Jansen's method (Section 2.2.2). The second is defined below.

2.4.1 Dealing with strict primitives

As well as user-defined data constructors, the core functional language supports, as *primitives*, machine integers and associated arithmetic operators. Under lazy evaluation, primitive functions, such as integer multiplication, need special treatment because their arguments must be fully evaluated *before* they can be applied. Peyton Jones and Jansen solve this problem by making their template instantiators recursively evaluate each argument to a primitive. This is an elegant approach when the template instantiator is written in a programming language like Miranda or C, where the presence of an implicit call stack may be assumed. But special-purpose machines have no such implicit call stack, so an alternative solution must be found.

The solution taken here is to rewrite every two-argument primitive function application by the rule

$$p\ n\ m \ \rightarrow \ m\ (n\ p)$$

where p is a primitive function name and n and m are the expressions (of integer type) to which p is applied. After applying this rule, it is clear that m must be evaluated before n p, but not so clear how a machine integer can be applied to an argument! The idea is that the application of a fully

node	::=	`Start` i j	(1^{st} node of function body: arity and size)
	|	`Int` i	(integer literal)
	|	`Ap` i	(application node: pointer to a node sequence)
	|	`End` *node*	(the final node in a node sequence)
	|	`Prim` p	(primitive function name)
	|	`Fun` i	(pointer to a function body)
	|	`Var` i	(variable representing a function argument)

Figure 2.3: The syntax of nodes in Reduceron bytecode.

$$
\begin{array}{rclrcl}
op\ (\texttt{Prim}\ p) & = & p & unEnd\ (\texttt{End}\ n) & = & n \\
contents\ (\texttt{Ap}\ i) & = & i & isEnd\ (\texttt{End}\ n) & = & True \\
contents\ (\texttt{Fun}\ i) & = & i & isEnd\ n & = & False \\
contents\ (\texttt{Int}\ i) & = & i & arity\ (\texttt{Start}\ i\ j) & = & i \\
contents\ (\texttt{Var}\ i) & = & i & size\ (\texttt{Start}\ i\ j) & = & j
\end{array}
$$

Figure 2.4: Helper functions for bytecode nodes.

evaluated integer i to an arbitrary expression e, that is $i\ e$, can be reduced to $e\ i$. Section 2.5.2 will show that an evaluator can, as a result of this transformation, easily deal with strict primitives. The factorial function is now:

```
fact n = 1 (n (==)) (fact (1 (n (-))) (n (*))) 1
```

The list-reversing function `rev` is not affected by this transformation, as it contains no primitives.

2.4.2 Bytecode

The bytecode of a program is defined to be a sequence of *nodes*, and the syntax of a node is defined in Figure 2.3. In the syntax definition, the meta-variables i and j range over integers, and p ranges over primitive function names. The helper functions defined in Figure 2.4 facilitate deconstruction of bytecode nodes.

An n-ary application node, `Ap` i in Reduceron bytecode, is a pointer i to a sequence of n consecutive nodes in memory whose final node is wrapped in

a (`fact`)	+1	+2	+3
`Start 1 15`	`Int 1`	`Ap 7`	`Ap 5`
+4	+5	+6	+7
`End (Int 1)`	`Prim (==)`	`End (Var 0)`	`Ap 12`
+8	+9	+10	+11
`Ap 10`	`End (Fun` a`)`	`Ap 14`	`End (Int 1)`
+12	+13	+14	+15
`Prim (*)`	`End (Var 0)`	`Prim (-)`	`End (Var 0)`

b (`rev`)	+1	+2	+3
`Start 2 5`	`Ap 4`	`Var 2`	`End (Var 1)`
+4	+5		
`Var 2`	`End (Fun` c`)`		

c (`rev'`)	+1	+2	+3
`Start 3 6`	`Ap 4`	`Var 3`	`End (Fun` b`)`
+4	+5		
`Var 1`	`Var 2`	`End (Fun` d`)`	

d (`Cons`)	+1	+2	+3
`Start 4 3`	`Var 2`	`Var 1`	`End (Var 4)`

Figure 2.5: Bytecode for the example functions.
The bytecode for `fact`, `rev`, `rev'`, and `Cons` appearing relative to addresses a, b, c, and d respectively in code memory.

an `End` marker. To permit sharing in over-saturated applications, the nodes in an application sequence are stored in *reverse* order. For example, $f\ x\ y$ is stored as $y\ x$ (`End` f). If $f\ x$ evaluates to z then the application can simply be updated to y (`End` z) without relocating it in memory.

Figure 2.5 shows the bytecode for the running examples. Each application node in the bytecode is an *offset* address, relative to the address of the first node of the function's bytecode. This first node is always a `Start` node, and defines the arity and size (number of words) of the function's body. The bytecode for a whole program is simply the concatenation of the bytecodes for each individual function. Each `Fun` node is then adjusted to point to the final address of the function in the bytecode.

2.5 Semantics

In this section, a semantics for the Reduceron is defined. There are two reasons for presenting a semantics: first to define precisely how the Reduceron works, and second to highlight the parts of the reduction process that can be assisted by *special-purpose hardware*. The semantics is given as a binary small-step state transition relation, $\Rightarrow$, between triples of the form $\langle h,\ s,\ a\rangle$, where h is the heap, s is the node stack, and a is the address stack. The heap is used for both template memory and graph memory.

In the semantics, the heap and stacks are modelled as *lists*. Reduction rules are expressed with the help of common list-processing functions defined, for example, in the Haskell prelude. In addition, $\#xs$ is written to denote the *length* of the list xs, and $xs[i \mapsto x]$ to denote xs with its i^{th} element replaced by x.

Initially, the heap contains the bytecode of the program, the node stack contains the node `Fun 0`, where 0 is the address of the function `main :: Int`, and the address stack contains the address 0. The final result of a program p is defined to be r where

$$\langle p,\ [\texttt{Fun}\ 0],\ [0]\rangle\ \Rightarrow^{\star}\ \langle \_,\ [\texttt{Int}\ r],\ \_\rangle$$

Notice that the `main` function is a pure (non-monadic) function with no

arguments. Currently, Reduceron programs take no input. Furthermore, their output is a single integer (`main` is of type `Int`).

2.5.1 A primitive evaluator

The semantics assumes a function $\mathcal{P}$ that takes a primitive function name p and two integers, i and j, and returns a *node* representing the value of $p\ i\ j$. For example,

$$\mathcal{P}\ \texttt{(+)}\ 5\ 10\ =\ \texttt{Int}\ 15$$

and

$$\mathcal{P}\ \texttt{(==)}\ 1\ 1\ =\ \texttt{Fun}\ \mathit{TrueAddr}$$

where *TrueAddr* is the address of the function `True` in the bytecode of the program.

2.5.2 Semantic definition

The small-step transition relation $\Rightarrow$ is defined in Figure 2.6 and the helper functions *inst* and *unwind* are defined in Figure 2.7. There is one transition rule for each possible type of node that can appear on top of the stack, as described by the following paragraphs.

Primitives

Recall from Section 2.4.1 that primitive applications of the form $p\ n\ m$, where n and m are unevaluated integers, are transformed to $m\ (n\ p)$. Clearly, to evaluate such an application, m must be evaluated first. Hence the *value* of m, of the form `Int` j, appears on top of the stack. To deal with such a situation, the evaluator simply *swaps* the top two stack elements, resulting in $(n\ p)$ (`Int` j) on the stack – growing from right to left. Further evaluation yields (`Int` i) p (`Int` j) on top of the stack, where `Int` i is the result of evaluating n. Another swap gives p (`Int` i) (`Int` j), which can be evaluated by a straightforward application of $\mathcal{P}$.

Once the result of the primitive application has been computed, it must be written onto the heap, overwriting the contents of the original application

$$
\begin{array}{rcl}
\langle h,\ \texttt{Int}\ i : x : s,\ a\rangle & \Rightarrow & \langle h,\ x : \texttt{Int}\ i : s,\ a\rangle \\
\\
\langle h,\ \texttt{Prim}\ p : \texttt{Int}\ x : \texttt{Int}\ y : s, & & \\
\quad \_ : \_ : r : a\rangle & \Rightarrow & \langle h[r \mapsto \texttt{End}\ z],\ z : s,\ r : a\rangle \\
\quad \textbf{where} & & \\
\quad z & = & \mathcal{P}\ p\ x\ y \\
\\
\langle h,\ \texttt{Ap}\ i : s,\ \_ : a\rangle & \Rightarrow & \mathit{unwind}\ i\ \langle h,\ s,\ a\rangle \\
\\
\langle h,\ \texttt{Fun}\ i : s,\ a\rangle & \Rightarrow & \mathit{unwind}\ \#h\ \langle h',\ s',\ a'\rangle \\
\quad \textbf{where} & & \\
\quad \texttt{Start}\ \mathit{arity}\ \mathit{size} & = & h\ !!\ i \\
\quad \mathit{body} & = & \mathit{take}\ \mathit{size}\ (\mathit{drop}\ (i+1)\ h) \\
\quad s' & = & \mathit{drop}\ \mathit{arity}\ s \\
\quad r : a' & = & \mathit{drop}\ \mathit{arity}\ a \\
\quad h' & = & h[r \mapsto \texttt{End}\ (\texttt{Ap}\ \#h)] \\
& & \quad +\!\!+\ \mathit{map}\ (\mathit{inst}\ s\ \#h)\ \mathit{body}
\end{array}
$$

Figure 2.6: Transition rules for the Reduceron.

$$
\begin{array}{rcl}
\mathit{inst}\ s\ b\ (\texttt{Var}\ i) & = & s\ !!\ i \\
\mathit{inst}\ s\ b\ (\texttt{Ap}\ i) & = & \texttt{Ap}\ (b+i-1) \\
\mathit{inst}\ s\ b\ (\texttt{End}\ n) & = & \texttt{End}\ (\mathit{inst}\ s\ b\ n) \\
\mathit{inst}\ s\ b\ n & = & n \\
\\
\mathit{unwind}\ i\ \langle h,\ s,\ a\rangle & = & \langle h,\ \mathit{reverse}\ ap +\!\!+ s,\ \mathit{reverse}\ as +\!\!+ a\rangle \\
\quad \textbf{where} & & \\
\quad ap & = & \mathit{getAp}\ (\mathit{drop}\ i\ h) \\
\quad as & = & \mathit{map}\ (i+)\ [0 \ldots \#ap - 1] \\
\\
\mathit{getAp}\ (\texttt{End}\ n : ns) & = & [n] \\
\mathit{getAp}\ (n : ns) & = & n : \mathit{getAp}\ ns
\end{array}
$$

Figure 2.7: Definitions of *inst* and *unwind*.

node m (n p), so that other references to it do not repeat the computation. This is possible because, as explained below, a pointer to the original application is sitting on the *address stack*.

Applications

When an application node of the form `Ap` i appears on top of the stack, it is replaced by the `End`-terminated sequence of nodes starting at address i on the heap. Furthermore, the addresses of the nodes in the sequence are pushed on the address stack, to permit updating the sequence after reduction. Peyton Jones describes this process as "*unwinding* the application" [78].

In an implementation of the Reduceron on a standard PC architecture, each node in an application sequence is read, one at a time, from the heap and written, one at a time, to the stack. Furthermore, each node address is computed and written, again one at a time, onto the address stack.

The definition of the *unwind* function in Figure 2.7 highlights the first main opportunities for hardware-assisted graph reduction. First, the uses of *getAp* and $+\!\!+$ illustrate that the nodes being copied are *contiguous*, so the copying can be achieved by block transfers in a machine with a wider data bus. Second, the use of *map* to compute the node addresses indicates that they can be computed in parallel. And third, there is no dependency between writing to the node and address stacks, so the two can be done at the same time in a machine with parallel memories.

Functions

When a node of the form `Fun i` is at the top of the stack, the bytecode starting at address $i + 1$ on the heap is

1. *copied* onto the end of the heap (say at address *base*),
2. with the j^{th} argument on the stack *substituted* for each variable `Var` j,
3. and with each application node, `Ap` k, *relocated* to an absolute address, `Ap` $(base + k - 1)$, on the heap.

Subsequently, n nodes are popped off the node and address stacks, where n is the arity of the function that has just been instantiated. The address r which is n places from the top of the address stack represents the *root* of the redex. The value at address r is overwritten with `End` (`Ap` *base*), so that the reduction is never repeated. Finally, the node sequence beginning at the address *base* is unwound onto the stack. This whole process is termed *function unfolding*.

Just as for unwinding, function unfolding on a standard PC architecture requires execution of many sequential instructions to carry out all the necessary memory manipulations. And again the semantics shows great scope for parallelism. In particular, the use of $+\!\!+$ to copy a potentially large contiguous block of nodes onto the end of heap, and the use of *map* to instantiate each node independently, opens up the possibility for parallelisation on a machine with wide memory and vectorised processing logic. Since instantiation of a node requires access to the stack, a parallel evaluator would need to be able to read the stack and heap at the same time. Further, because the nodes are being copied from one portion of memory to another, sequentialisation can be reduced by separating template memory and graph memory, permitting parallel access.

Notice in the semantics that the `Fun` rule calls *unwind*. Immediately after a function body is instantiated on the heap, the *spine* of that body is unwound from the heap onto the stack. It is more efficient to instantiate the spine of the function body on the heap and the stack *in parallel*. This idea is related to the *spineless* G-machine [15], which can in some circumstances avoid construction of the spine on the heap. Parallel instantiation gives the speed benefit of the spineless G-machine without introducing complexity, but not the space benefit.

2.6 The Narrow Reduceron

The Reduceron semantics is now refined to *two* clock-accurate register-transfer models, one *Narrow* and the other *Wide*. The Wide version exploits wide, parallel memories to aid performance, and the Narrow version does not. The models presented are close to final hardware implementations,

Register	Type	Description
`x`, `y`, `z`	*node*	Temporary
`top`	*node*	Top of stack
`hp`, `sp`	*integer*	Heap and stack pointer
`root`	*integer*	Address of redex root in the heap
`base`	*integer*	Base address of function instantiation
`i`	*integer*	Temporary
`end`	*boolean*	End of application sequence?
`h`	*array of node*	Heap
`c`	*array of node*	Code
`s`	*array of node*	Stack
`a`	*array of integer*	Address-stack

Figure 2.8: Registers used by the Narrow Reduceron.

but integers, bytecode nodes, and arrays are not encoded at the FPGA-level until Section 2.8. The motivation for defining two models is to show exactly what is made possible by widening the von Neumann bottleneck. First, the Narrow Reduceron is considered (this section), and then the Wide Reduceron (Section 2.7).

2.6.1 Memory

Figure 2.8 describes the registers used by the Narrow Reduceron. Although there are four separate arrays for the heap (graph memory), code (template memory), stack, and address-stack, they all reside in a single RAM. In other words, only one location of one array can be accessed in a single clock-cycle. Motivated by implementation concerns (discussed in Section 2.8), it is assumed that two clock-cycles are required to lookup an array. Writing to an array requires just one clock-cycle, however.

The following assumptions are made about the initial state of the machine:

- the first location in the heap, `h`[0], contains the node `End` (`Fun` 0), where 0 is the address of the `main` function,
- the heap pointer is 1,

s	::=	$v \leftarrow e$	(schedules assignment for next clock-cycle)
	|	$v \Leftarrow e$	(schedules assignment for next-but-one clock-cycle)
	|	`skip`	(does nothing)
	|	`tick`	(consumes a clock-cycle)
	|	s_0 `;` s_1	(composes two statements)
	|	`if` e s	(executes a statement if e holds)
	|	`repeat` e s	(repeats a statement e times)
	|	`while` e s	(repeats a statement while e holds)
	|	`do` s `until` e	(repeats a statement until e holds)
	|	`let` v `=` e	(binds a variable to a value)

Figure 2.9: Syntax of register-transfer statements.

- the top of the stack, `top`, contains the node `Ap` 0,
- the stack pointer contains $StackSize - 1$ where $StackSize$ is the number of locations in the array `s`,
- the top of the address-stack, `a`$[StackSize - 1]$, contains the address 0,
- and the array `c` contains bytecode of the program to run.

2.6.2 Register-transfer notation

Clock-accurate models will be expressed using register-transfer statements whose syntax is described in Figure 2.9. The meta-variable v ranges over variables, and e over expressions. The semantics of such statements is discussed informally below. (A precise semantics can be found in Chapter 3.)

A register-transfer statement is quite similar to a statement in an imperative programming language. The main difference is that an assignment in a register-transfer language is associated with a *clock-cycle* in which it executes. The *effect* of an assignment executed in the n^{th} clock-cycle is not visible until clock-cycle $n + 1$. To illustrate, first observe that several statements can be composed executed in a single clock-cycle using '`;`'.

`x` $\leftarrow$ 1 `;` `y` $\leftarrow$ 2 `;` `z` $\leftarrow$ 3

To mark the beginning of a new clock-cycle, the keyword `tick` is used.

$\texttt{x} \leftarrow 1 \texttt{ ; } \underline{\texttt{tick}} \texttt{ ; x} \leftarrow \texttt{x} + 1 \texttt{ ; y} \leftarrow \texttt{x ; } \underline{\texttt{tick}}$

Here, if the first assignment is executed in clock-cycle n then the second and third are executed in clock-cycle $n + 1$. Furthermore, the effect of the first is not visible until cycle $n + 1$, and the effect of the second and third is not visible until cycle $n + 2$. So in cycle $n + 1$, `x` has the value 1, and in cycle $n + 2$, `x` has the value 2 but `y` has the value 1.

As another example, the following statement successfully swaps the values of `x` and `y`, without the need for an intermediate register.

$\texttt{x} \leftarrow \texttt{y ; y} \leftarrow \texttt{x ; } \underline{\texttt{tick}}$

Since array lookups require two clock-cycles to complete, it is convenient to have a second kind of assignment, '$\Leftarrow$', which if executed in cycle n, has no effect until cycle $n + 2$. For example, the following statement increments element 2 of the array `a`.

$\texttt{i} \Leftarrow \texttt{a}[2] \texttt{ ; } \underline{\texttt{tick}} \texttt{ ; } \underline{\texttt{tick}} \texttt{ ; a}[2] \leftarrow \texttt{i} + 1 \texttt{ ; } \underline{\texttt{tick}}$

The operator '`|`' will be used to reduce clutter, highlighting the separation of clock-cycles.

$$s_0 \texttt{ | } s_1 \quad = \quad s_0 \texttt{ ; } \underline{\texttt{tick}} \texttt{ ; } s_1$$

So, an equivalent way to write the statement

$\texttt{x} \leftarrow 1 \texttt{ ; } \underline{\texttt{tick}} \texttt{ ; x} \leftarrow \texttt{x} + 1 \texttt{ ; y} \leftarrow \texttt{x ; } \underline{\texttt{tick}}$

is to write

$\texttt{x} \leftarrow 1 \texttt{ | x} \leftarrow \texttt{x} + 1 \texttt{ ; y} \leftarrow \texttt{x | } \underline{\texttt{skip}}$

By counting the number of '`|`' operators, one can see that this statement takes two clock-cycles to complete.

2.6.3 Register-transfer model

The following paragraphs define clock-accurate register-transfer models of each Reduceron transition rule. As before, the transition rule that fires depends on the top stack element. The number of clock-cycles required by each transition is calculated by summing the number of '`|`' operators seen

```
swap = top ⇐ s[sp + 1]
     | s[sp + 1] ← top
     | skip

prim = x ⇐ s[sp + 1]
     | y ⇐ s[sp + 2]
     | root ⇐ a[sp + 2]
     | top ← P (op top) x y
     | sp ← sp + 2 ; h[root] ← End top
     | skip
```

Figure 2.10: Clock-accurate model of primitive evaluation (Narrow).

(and multiplying appropriately to account for loops). No clock-cycles elapse between individual transitions.

Primitives

Clock-accurate models of the transition rules for primitives (`swap` and `prim`) are given in Figure 2.10. The `swap` statement swaps the top two stack elements. A memory-read is scheduled in its first cycle, and a memory-write in its second. Reading from and writing to memory cannot be parallelised in the Narrow Reduceron. Note that the effect of the assignment to `top` is not visible until the third clock-cycle. In total, two clock-cycles are consumed by `swap`.

In its first three clock-cycles, `prim` schedules three reads from memory: two for the primitive's arguments, and one for the address of the redex-root. In its fourth cycle, the primitive function is applied to the two arguments, freshly available in the registers `x` and `y`, and the result is assigned to `top`. In its fifth cycle, the address of the redex-root, freshly available in the register `root`, is used to update the heap. In total, five cycles are consumed by `prim`.

```
unwind = x ⇐ h[contents top]
       ; i ← contents top
       | do ( a[sp] ← i
            | s[sp] ← unEnd x
            ; top ← unEnd x
            ; end ← isEnd x
            ; i ← i + 1
            | if (not end) (x ⇐ h[i] ; sp ← sp − 1 | skip)
            ) until end
```

Figure 2.11: Clock-accurate model of unwinding (Narrow).

Unwinding

A clock-accurate model of application-unwinding is shown in Figure 2.11. An initial read from the heap consumes one cycle. A loop then copies all nodes in the application sequence onto the stack. Each but the final loop iteration consumes three cycles: one for writing the address of the node onto the address-stack, one for writing the node onto the stack, and one for reading the next node in the application sequence from the heap. The final iteration consumes only two cycles because the next node need not be read from the heap. In total, $1 + 3n - 1$ cycles are consumed by `unwind`, where n is the number of nodes in the application sequence.

Unfolding

A clock-accurate model of function-unfolding is shown in Figure 2.12. In its first three clock-cycles, it schedules three reads from memory. The `Start` node of the function and the first node of the function body are read from code memory, and the address of the redex-root is read from the address-stack. A loop then instantiates each node of the function body on the heap. Typically, each iteration consumes two clock-cycles: one to read the next node from code memory, and one to write a newly-instantiated node to the heap. However, instantiation of a `Var` node requires two additional clock-cycles, because the argument must be read from the stack. Node instantiation (`inst`) is defined separately in Figure 2.13. A further cycle is

```
unfold = x ⇐ c[contents top]
       ; i ← 1 + contents top
       ; base ← hp
       | y ⇐ c[i]
       | root ⇐ a[sp + arity x]
       | repeat (size x)
           ( y ⇐ c[i + 1]
           ; i ← i + 1
           | inst id y
           | skip
           )
       ; sp ← sp + arity x
       ; h[root] ← End (Ap base)
       ; top ← Ap base
       | unwind
```

Figure 2.12: Clock-accurate model of unfolding (Narrow).

```
inst f (End n) = inst End n
inst f (Ap n)  = h[hp] ← f (Ap (s[base + n − 1])) ; hp ← hp + 1
inst f (Var n) = z ⇐ s[sp + n + 1]
               | skip
               | h[hp] ← f z ; hp ← hp + 1
inst f n       = h[hp] ← f n ; hp ← hp + 1
```

Figure 2.13: Clock-accurate model of instantiation (Narrow).

consumed to update the redex-root on the heap. Finally, the spine of the function body that has just been instantiated on the heap is unwound onto the stack. In total, `unfold` consumes

$$3 + 2s + 2v + 1 + 3n$$

clock-cycles, where s is the total number of nodes in the function body, v is the number of `Var` nodes in the function body, and n is the number of nodes in the spine of the function body.

2.6.4 Discussion

Under the memory constraints of the Narrow Reduceron, there is little scope to reduce the number of clock-cycles consumed by each transition. On *almost* every clock-cycle, a necessary memory access is performed. There are only three situations in which this is not the case.

1. In the fourth cycle of `prim`, no memory access is made.
2. In the loop body of `unfold`, the next node to instantiate is read from the heap, and this is not necessary on the final iteration.
3. In `inst`, when a function argument is instantiated on the heap, there is a clock-cycle in which no memory access is made.

However, in each case the Reduceron must wait for an earlier memory request to complete. Even if memory could be better utilised in these cases, the savings would be modest, and may not justify the extra complexity introduced. As an aside, it seems reasonable to conclude that the two-cycle memory-read assumption does not impede performance significantly.

One way to relieve the memory bottleneck would be to store chunks of memory – e.g. the top few stack elements – locally in registers where they can be accessed in parallel and read in a single cycle[1]. Instead however, the next section takes the simpler and more general approach of widening the memory bandwidth. In other words, there are designs lying between the Narrow and Wide extremes, but they are not considered here.

[1]Indeed, the Narrow Reduceron does store the top stack element in the register `top` and not in stack memory.

Register	Type	Description
`todo`	*integer*	Number of nodes left to instantiate
`rootsp`	*integer*	Stack pointer to redex-root
`sbuf`	*list of node*	Stack buffer
`hbuf`	*list of node*	Heap buffer
`cbuf`	*list of node*	Code buffer

Figure 2.14: Additional registers used by Wide Reduceron.

2.7 The Wide Reduceron

This section presents a clock-accurate model of the Wide Reduceron at the register-transfer level. The memory bottleneck is widened in three ways.

1. The heap, code, stack and address-stack arrays are mapped onto separate memories, each of which can be accessed in parallel.
2. Any eight consecutive locations in a memory, beginning at any address, can be read or written in parallel. Furthermore, a wide write may control exactly which of the eight consecutive locations are written to.
3. Memory can be read and written in parallel provided that the same location is being addressed, with the semantics that data is written to memory before it is read.

The widening factor of eight is somewhat arbitrary. In fact, the model below could be parameterised by the width of the memories. The only justification for eight is that it is a power of two sufficiently large to expose the potential of the approach yet small enough to permit implementation in the hardware available.

2.7.1 Memory

In addition to the registers listed in Figure 2.8, five further registers listed in Figure 2.14 are used. The arrays `h`, `s`, `c`, and `a` are generalised such that when read, they yield a list of eight consecutive elements in the array beginning at the given array index. Similarly, up to eight list elements can be written contiguously into the arrays beginning at a given array index. The

```
read top' sp' = top ← top'
              ; sp ← sp'
              ; sbuf ⇐ s[sp']
              ; hbuf ⇐ h[contents top']
              ; cbuf ⇐ c[contents top']
```

Figure 2.15: A frequently-used abstraction for reading memory.

`sbuf`, `hbuf`, and `cbuf` registers are used to buffer lists of eight elements that are read from stack, heap, and code memory respectively. No assumptions beyond those mentioned in Section 2.6.1 are made about the initial state of the machine.

2.7.2 Register-transfer model

Clock-accurate models of each transition in the Wide Reduceron are given below. The numbers of clock-cycles required are compared with those of the Narrow Reduceron. As before, no clock-cycles are consumed *between* each transition.

Reading memory

First, it is convenient to define a frequently-used abstraction. Quite often the stack pointer and the top of stack are modified, followed by a series of fresh memory reads. Specifically, stack memory is often read, indexed by the new stack pointer, and heap and code memories are often read, indexed by the contents of the new top stack element. An abstraction over this common behaviour is shown in Figure 2.15.

The following invariant is maintained by each transition rule of the Wide Reduceron. In the second clock-cycle of any transition, the `sbuf`, `hbuf`, and `cbuf` registers are *up-to-date*. That is, `sbuf` reflects the values in stack memory at the index specified by the register `sp`, and `hbuf` and `cbuf` reflect the values in heap and code memory at the index specified by the register `top`.

```
swap = s[sp] ← top
     | read (sbuf !! 0) sp
     | skip

prim = skip
     | x ← P (op top) (sbuf !! 0) (sbuf !! 1)
     ; root ← a[sp+1]
     | read x (sp+2)
     | h[root] ← End x
```

Figure 2.16: Clock-accurate model of primitive evaluation (Wide).

Primitives

Models of the transition rules for handling primitives are shown in Figure 2.16. Swapping still consumes two clock-cycles, due to maintaining the buffer invariant, but primitive application now takes three rather than five clock-cycles. The two-cycle saving is due to fact that both operands are available on the second cycle rather than the fourth as in the Narrow Reduceron.

Unwinding

A clock-accurate model of unwinding is presented in Figure 2.17. Recall that the function *getAp* is defined as part of the semantics given in Figure 2.7. In the second clock-cycle of `unwind`, the whole application sequence is available in `hbuf` and is immediately written onto the stack. Note that the stack is read from and written to in the same clock-cycle – both the read and the write access the same address. In parallel, the addresses of the nodes in the sequence are block-written onto the address-stack. The end result is that unwinding requires just two clock-cycles.

Here it is assumed that application sequences are limited to a maximum of eight nodes. This limitation is easily hidden by the compiler by transforming long applications to nested, smaller ones. For example, $f\ a\ b\ c\ d$ is equivalent to $(f\ a\ b)\ c\ d$.

```
unwind = skip
       | let ap = reverse (getAp hbuf)
       ; let len = length ap − 1
       ; let addrs = map (+ contents top) [7..0]
       ; read (head ap) (sp − len)
       ; s[sp − len] ← tail ap
       ; a[sp − 8] ← addrs
       | skip
```

Figure 2.17: Clock-accurate model of unwinding (Wide).

Unfolding

Figure 2.18 gives a clock-accurate model of function unfolding. Recall that the function *inst* is defined as part of the semantics in Figure 2.7. In the while loop of `unfold`, eight nodes are read from code memory, instantiated, and written to the heap in a single clock-cycle on each iteration. Just $\lfloor \frac{s}{8} \rfloor$ clock-cycles, where s is the number of nodes in the function body, are needed to fully instantiate a function body on the heap. The first block of eight nodes is processed specially, before the loop is entered, for two reasons.

1. Unlike any other block of eight, it contains the `Start` node stating the arity and size of the function.

2. It contains the spine of the function body which must be instantiated on the stack.

To increase performance, the first block is instantiated on the heap in the same clock-cycle that the spine is unwound onto the stack. In total, $3 + \lfloor \frac{s}{8} \rfloor$ cycles are consumed by `unfold`.

Here it is assumed that functions have a maximum of eight parameters. This is because only eight elements of the stack can be accessed simultaneously while instantiating a function body. Again, the limitation can be hidden by the compiler, either by tupling arguments together or by using an abstraction algorithm [92].

```
unfold = cbuf ⇐ c[contents top + 8]
       ; i ← contents top + 16
       ; base ← hp
       | let numArgs = arity (cbuf !! 0)
       ; let bodyLen = size (cbuf !! 0)
       ; let spineLen = length (getAp cbuf) − 1
       ; let body = map (inst sbuf base) cbuf
       ; sp ← sp + numArgs − spineLen + 1
       ; root ← a[sp + numArgs]
       ; rootsp ← sp + numArgs
       ; todo ← bodyLen − 7
       ; top ← unEnd (body !! spineLen)
       ; hp ← hp + min bodyLen 7
       ; i ← i + 8
       ; s[sp + numArgs − 7] ← reverse (map unEnd (tail body))
       ; h[hp] ← tail body
       ; cbuf ⇐ c[i]
       | let addrs = map (+ base) [7..0]
       ; a[rootsp − 8] ← addrs
       ; while (todo > 0)
           ( h[hp] ← body
           ; cbuf ⇐ c[i]
           ; hp ← hp + min todo 8
           ; todo ← todo − 8
           ; i ← i + 8
           | skip
           )
       ; read top sp
       | h[root] ← End (Ap base)
```

Figure 2.18: Clock-accurate model of unfolding (Wide).

Transition	**Narrow**	**Wide**
Swap	2	2
Primitive	5	3
Unwind	3n	2
Unfold	$4 + 2s + 2v + 3n$	$3 + \lfloor \frac{s}{8} \rfloor$

Notes: n is the number of nodes in an application or in the spine of a function body, s is the number of nodes in a function body, and v is the number of `Var` nodes in a function body. No clock-cycles are consumed between each transition.

Table 2.1: Clock-cycles taken by each Reduceron transition.

2.7.3 Discussion

Section 2.6.4 suggested that the Narrow Reduceron is not significantly impeded by the two-cycle array-lookup assumption. In contrast, the `unwind` transition of the Wide Reduceron wastes its first cycle waiting for a lookup to complete. So the speed of unwinding, a potentially common operation, could be doubled.

The effect of widening the memory bottleneck on the number of clock-cycles required by a graph reduction machine based on template instantiation, the Reduceron, is summarised in Table 2.1.

The clock-accurate models of the Narrow and Wide Reducerons have been simulated on several benchmark programs, but no attempt has been made to prove that the high-level semantics and the lower-level models are consistent.

2.8 Implementation on FPGA

This section refines the clock-accurate Reduceron models (Narrow and Wide) to the FPGA-level. In particular, bytecode nodes are encoded at the bit-level, and arrays of nodes are mapped onto FPGA block RAMs. The memory layout, garbage collector, and synthesis results are also discussed. The specific FPGA chip used for the implementations is the Xilinx Virtex-II `XC2V2000-6BF957`.

2.8.1 Word size

Both the Wide and Narrow Reducerons encode bytecode nodes as eighteen-bit words. The two most significant bits identify the node as an application, function, integer or variable. The third most significant bit is high if the node is an `End` node, and low otherwise. The remaining fifteen bits represent the contents of the node, i.e. an application pointer, function address, integer literal, or a variable identifier.

2.8.2 Block RAMs

The Virtex-II contains 56 independent 18 kilobit dual-port block RAMs. Being "dual-port" means that block RAMs have two address busses, two data busses and two write enable signals. Thus two different locations can be accessed in one clock-cycle. Furthermore, each port has separate busses for data input and data output, so a value may be simultaneously written to and read from a single location on a single port.

Only the Wide version exploits the dual-port and separate data bus features of block RAMs. Both versions configure all block RAMs as 1k by 18-bit RAMs, hence each memory location can store one bytecode node. There are alternative ways to configure block RAMs, so this decision is re-examined in Section 2.10.3.

2.8.3 Constructing large memories from small ones

The ability to cascade block RAMs to form larger memories is important in both Reduceron implementations. This is achieved using a decoder and a multiplexor as shown in Figure 2.19. In particular, to construct a single memory out of m block RAMs the most significant $log_2\ m$ bits of the memory's address are

1. fed to the selector input of the multiplexor, so the output of the addressed block RAM is passed through to the output, and
2. used to inhibit the write signal, so that only the addressed block RAM is written to.

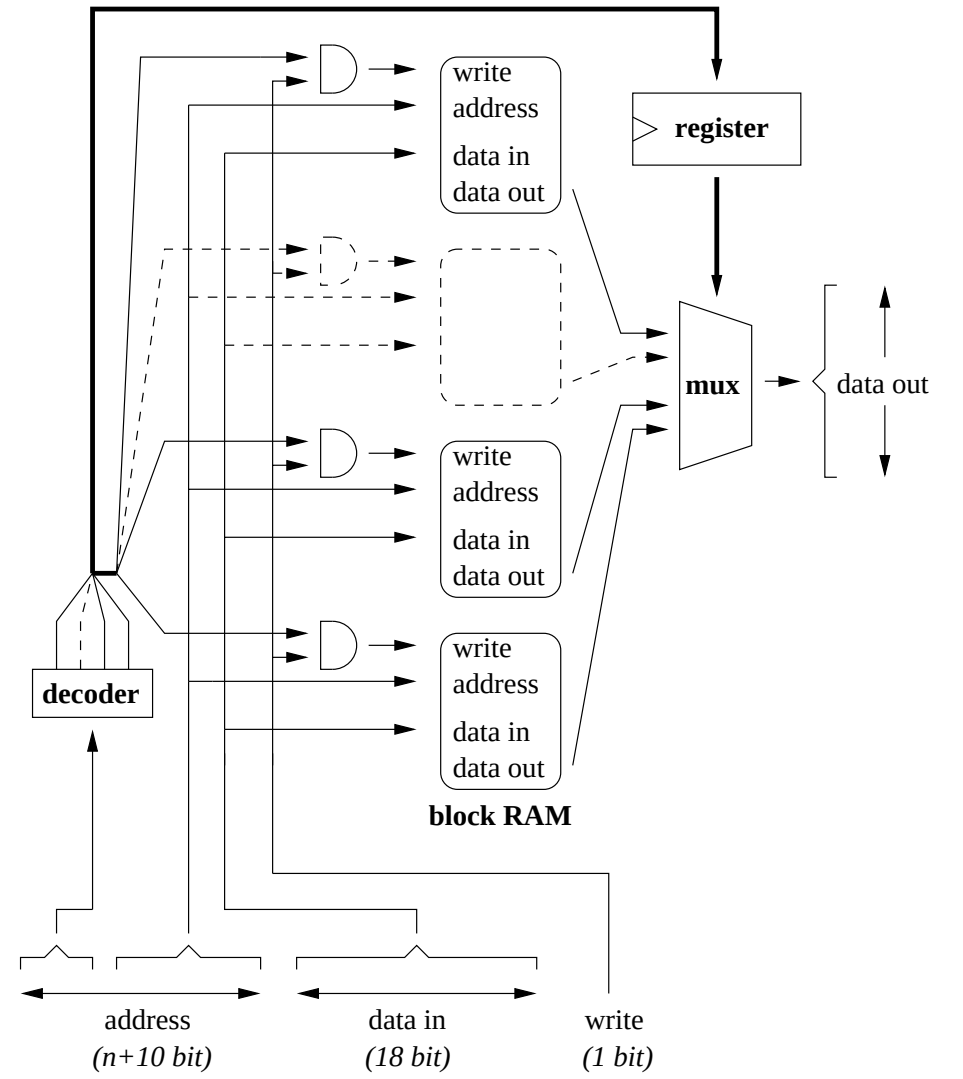

Figure 2.19: Cascading block RAMs to form a larger RAM.

This is a fairly standard way to cascade memories, but an alternative is discussed in Section 2.10.3.

When cascading a large number of block RAMs the multiplexor becomes rather large and its delay becomes significant. To overcome this inefficiency, a register is placed on the output of the multiplexor. This means that two clock-cycles are needed between writing an address to the address-bus and reading the resulting value off the data-bus.

2.8.4 Quad-word memories

To permit wider memory transfers, the Wide Reduceron uses *quad-word* memories. A quad-word memory allows any four consecutive locations to read or written in one clock-cycle. This is *not* the same as simply quadrupling the word size because that would mean that only blocks of words beginning at a four word boundary could be accessed together. Quad-word memories make the Wide Reduceron a natural extension of the Narrow Reduceron – any location can still be directly addressed, but with the optional benefit of accessing the subsequent three locations too. Simply quadrupling the word size would introduce complications, such as word alignment issues, but would require less FPGA logic to implement than quad-word memory. This alternative will be discussed further in section 2.10.3.

Quad-word memories are built out of four separate memories. If each internal memory is numbered i where i is drawn from $[0, 1, 2, 3]$ then memory i is used to store locations $[i, i+4, i+8, \ldots]$ of the quad-word memory. Accessing four consecutive locations beginning at an address a is straightforward if a is a multiple of four, but awkward if it is not. However, each of the four consecutive locations, beginning at *any* address, must be stored in a different internal memory. So the problem is one of rotating the quad-word input and output data busses so they line up with those of the internal memories. The solution used is given in Figure 2.20. The *rotateLeft* and *rotateRight* circuits rotate a given list of inputs by a given number of positions. The *increment* circuit takes an address a and a number n, and produces four copies of a, the first n of which are incremented by one.

Finally, the Wide Reduceron uses quad-word memories that are also *dual-port*, so in fact up to eight consecutive words can be accessed simultaneously.

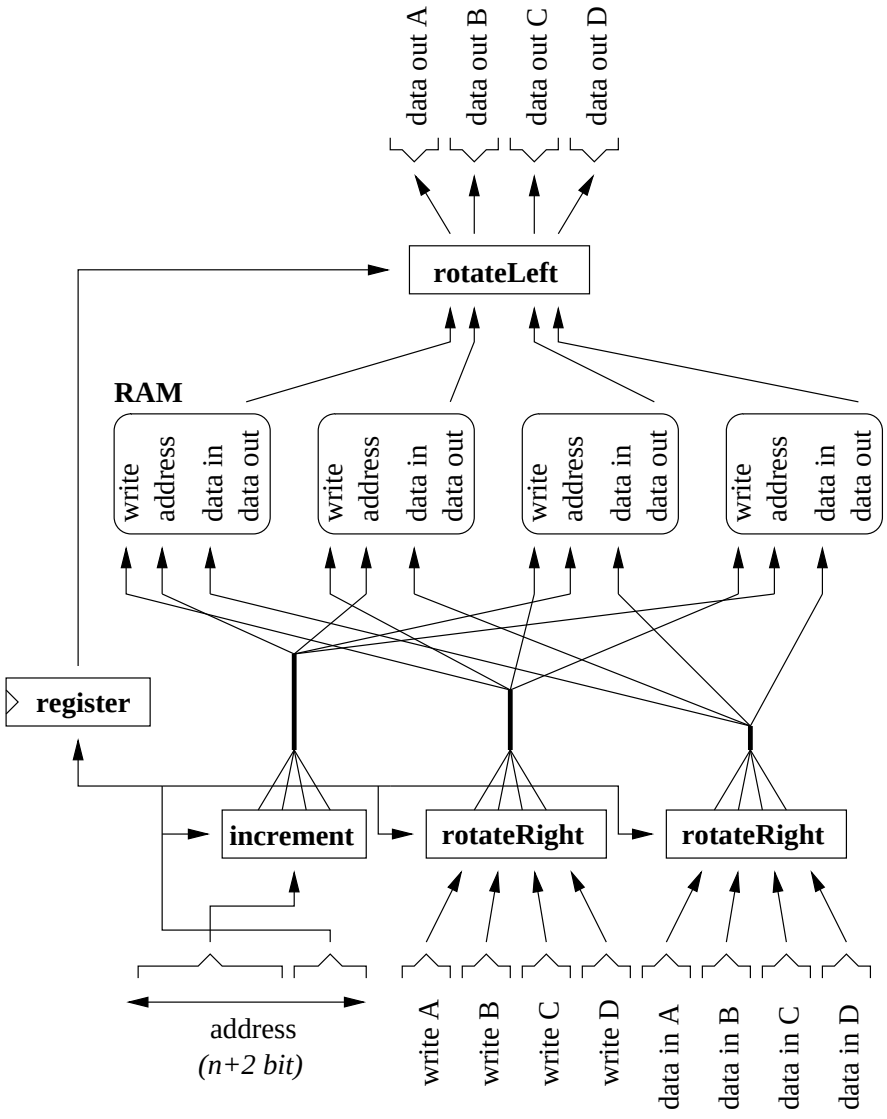

Figure 2.20: Circuit diagram for a quad-word memory.

Memory	**Capacity** (kilowords)
Code	4
Heap	32
Node stack	4
Address stack	4
Garbage collector scratch-pad	12

Table 2.2: The parallel, dual-port, quad-word memories of Wide Reduceron.

2.8.5 Memory layout

As well as widening memory, the Wide Reduceron accesses the stack, address-stack, heap and code arrays in parallel. The Reduceron FPGA implementations also use another array to facilitate garbage collection. Thus, the Wide Reduceron has five separate memories, each of which is shown in Table 2.2 alongside its capacity.

In contrast, the Narrow Reduceron cascades 32 block RAMs to form a single narrow memory, and 16 more to form a temporary storage for garbage collection.

2.8.6 Garbage collection

For any serious computations to be performed in such a small amount of memory, a garbage collector is essential. Both versions of the Reduceron use a simple stop-and-copy two-space garbage collector [26]. In this algorithm, active nodes in the heap are copied to an empty scratch-pad. The scratch-pad, which then contains a compacted copy of the heap, is copied back to the heap again before reduction continues. Although not the cleverest collector, it has the advantage of simplicity. Furthermore, the algorithm is easily defined to be *iterative* so no recursive call stack is needed. The focus here is on optimising the reduction process rather than exploring advanced garbage collectors.

	Narrow	**Wide**	**Maximum**
Slices	1516	4874	10752
Block RAMs	48	56	56
Clock frequency	94.7 MHz	91.5 MHz	(see text)

Table 2.3: Reduceron synthesis results on the Virtex-II (`XC2V2000-6BF957`).

2.8.7 Description language

Both versions of the Reduceron are implemented in Haskell using the Lava library [19]. Lava allows circuits to be described by Haskell functions over structures of booleans, and can turn such functions into VHDL netlists of FPGA components that can be synthesised by the Xilinx tool set. Chapter 3 introduces Lava and explains how the Reduceron is described.

2.8.8 Interface to the FPGA

The bytecode of the program to execute is encoded in block RAM initialisation parameters in the VHDL netlist of the Reduceron, so the program is transferred onto the FPGA as part of the configuration bitstream. The result of the program is displayed on a set of seven-segment-displays. Timing profiles stating the number of clock-cycles consumed by each machine transition can be requested using a set of DIP switches and observed on a set of LEDs.

2.8.9 Resource usage

The results of synthesising each version of the Reduceron on the Virtex-II using Xilinx ISE 9.1 are shown in Table 2.3. Concerning clock frequency: a small, carefully optimised 8-bit processor designed by Xilinx (the PicoBlaze) can be clocked at 173.6 MHz on the same device. For the Reduceron to be clocking within a factor of two is reasonable, but suggests room for improvement. One problem with the existing tool flow is that that there is no traceability from code written in Lava to the generated netlist, so it is hard to identify the critical path in the Lava program.

2.8.10 Minor variations

The implementations of the Narrow and Wide Reducerons follow the clock-accurate models presented in Sections 2.6 and 2.7 respectively, with only the following minor differences. These variations are of no great consequence but are stated anyway for accuracy.

- The Narrow Reduceron consumes three clock-cycles to implement `swap` rather than two. The reason is convenience of implementation. Elements from the stack are always read into a dedicated register, and the extra clock-cycle is used to transfer the contents of this register into the `top` register. In the model, the stack element is read *directly* into `top`.
- For simplicity, the Narrow Reduceron does not avoid the unnecessary memory access in the final iteration of the unwind loop. As a result, `unwind` consumes $3n + 1$ cycles instead of $3n$.
- When evaluating a primitive arithmetic application, it is known that the result is an integer. It is therefore known that the next transition will be a `swap`. The Wide Reduceron exploits this fact by doing the swap inside the `prim` transition, without consuming any additional clock-cycles. This is possible because of the wide stack.

2.9 Performance

In this section, the impact of the wide memory optimisations is measured by comparing the Narrow and Wide Reducerons running a range of Haskell programs. The clock-accurate models are compared in terms of clock-cycle consumption using a PC implementation of the Reduceron. The FPGA implementations are compared in terms of absolute runtime speed against the PC implementation of the Reduceron and against the main Haskell implementations running on a Pentium-4 2.8GHz PC.

Due to the memory and input/output restrictions of the FPGA implementations, the Haskell programs used in the experiments must:

1. have a maximum heap residency and stack size less than 32k words

and 4k words respectively,

2. not take any external input,

3. and produce a single integer as a result.

The programs used are shown in Figure 2.21. The Haskell implementations used are: Hugs (May 2006), GHCi (version 6.6), Yhc (unversioned), Nhc98 (version 1.20), a C implementation of the Reduceron, and the GHC native code compiler (version 6.6) with and without optimisations enabled.

2.9.1 Conclusions

Table 2.4 compares the clock-accurate Reduceron models. Table 2.5 gives absolute run-times of the Reduceron implementations on FPGA and the other Haskell implementations on PC. Table 2.6 shows time profiles of the Wide Reduceron.

On average, the Narrow Reduceron consumes 5.6 times as many clock-cycles as the Wide Reduceron. This difference is reflected by the FPGA implementations, where the average runtime difference is factor of 5.48. On heavily arithmetic programs (Queens and MSS), where much time is spent doing `swap` and `prim` transitions, the factor is between three and five, whereas on heavily applicative programs (Prop and While), where much time is spent unwinding and unfolding, it is around seven. The improvement offered by the Wide Reduceron is significant, but one might have hoped for more considering that eight consecutive locations can be accessed together on each of the five parallel memories. Some suggestions to utilise the parallel memory more fully are given in Section 2.10.

On average, the Wide Reduceron (running at 91.5MHz on FPGA) outperforms the Reduceron, Yhc, and Nhc98 bytecode interpreters (running at 2.8GHz on PC). All of these implementations share a common frontend, so each runs the same core Haskell programs. One of the potential advantages of a bytecode interpreter is that the bytecode can be made sufficiently abstract to have a concise formal semantics, offering hope for a mechanically verified Haskell implementation. However, there is a tension between defining a simple, high-level bytecode and one that is similar enough to the target machine so as to be *efficient*. The Reduceron appears to relax this tension;

1. **Prop**. A program to check the property that insertion into a list preserves ordering for all lists up to n booleans, applied to $n = 11$.
2. **Perm**. A program to find the smallest number in a list of numbers using a permutation sort, applied to the list containing the numbers 9 down to 1.
3. **MSS**. A program to compute the maximum segment sum of a list of integers applied to the list `[-150..150]`.
4. **Queens**. A function to compute the number of queens that can be placed on an n-by-n chess board such that no queen attacks any other queen, applied to $n = 10$.
5. **XO**. An adjudicator for noughts and crosses that determines if one side can force victory given a partially complete board. The adjudicator is applied to the empty board.
6. **Puz**. A solver for general cryptarithmetic puzzles. It is applied to a range of problems and outputs the total number of solutions to all of them. (Division is not supported on the Reduceron so division is implemented by repeated subtraction.)
7. **While**. A structural operational semantics of the *While* language [72] applied to a program that naively computes the number of divisors of 1000. (Divisor testing is implemented by repeated subtraction.)

Figure 2.21: Benchmark programs.

Program	**Narrow**	**Wide**	**Speed-up**
Prop	1109	163	6.77
Perm	360	53	6.79
MSS	562	153	3.66
Queens	649	137	4.72
XO	1307	238	5.48
Puz	734	137	5.32
While	1404	187	7.48

Table 2.4: Clock-cycles (in millions) to execute several programs.

	Prop	**Perm**	**MSS**	**Queens**	**XO**	**Puz**	**While**
Narrow	12.46	4.04	7.04	7.69	12.93	8.58	15.62
Hugs	3.68	2.70	3.85	6.50	14.81	4.11	5.49
GHCi	4.26	2.42	3.24	6.35	7.09	3.39	5.36
Yhc	3.59	1.76	1.22	3.06	3.85	2.51	3.81
PC	3.65	1.16	1.96	2.33	5.00	2.77	4.50
Nhc98	3.60	1.46	1.38	2.32	3.12	2.28	3.21
Wide	1.88	0.58	2.01	1.57	2.70	1.67	2.04
GHC	0.71	0.28	0.38	0.66	0.86	0.47	0.41
GHC -O2	0.57	0.19	0.28	0.09	0.30	0.27	0.34

Key: Narrow: Narrow Reduceron
Wide: Wide Reduceron
PC: PC Reduceron

Table 2.5: Timings (in seconds) of several programs running on various Haskell implementations.

a simple bytecode can be designed without concern for the target machine, and then a machine can be designed to efficiently execute this bytecode.

The PC version of the Reduceron performs well in comparison to Yhc and Nhc98, even though it is based on template instantiation and Yhc and Nhc98 are G-machine variants. This is consistent with recent work by Jansen [48] suggesting that small sets of coarse-grained instructions can compete with large sets of fine-grained ones for interpreting functional programs on modern PCs.

The leading native-code compiler GHC performs many advanced optimisations. For example, GHC spots that the critical `safe` function in Queens is strict, so need not be instantiated on the heap. Such optimisation gives GHC a significant advantage on strict, arithmetic programs. Excluding Queens and XO, which both involve significant integer operations in a critical loop, and which GHC's optimisations speed up by over a factor of two, the Reduceron (at 91.5MHz on FPGA) runs, on average, 4.85 times slower than GHC -O2 (at 2.8GHz on PC).

Program	Unwind	Unfold	Swap	Primitive	GC
Prop	31.4%	64.0%	0.0%	0.0%	4.6%
Perm	33.0%	54.7%	4.7%	3.5%	4.1%
MSS	41.7%	24.1%	5.1%	7.6%	21.5%
Queens	32.1%	27.8%	10.8%	12.2%	17.0%
XO	37.7%	37.8%	7.8%	6.8%	9.9%
Puz	36.8%	37.2%	6.0%	5.2%	14.8%
While	28.9%	55.8%	5.8%	4.6%	4.9%

Table 2.6: Profiles of programs running on the Wide Reduceron.

2.10 Limitations and future work

2.10.1 Memory capacity

One of the main limitations of the Reduceron is that it only has 32k words of heap space. This is enough to make an interesting experiment, but too small for any serious application. However, the limitation might be overcome with improved hardware, *without* affecting the existing design significantly. For example, the Computer Architecture group at York have built the PRESENCE-3 FPGA board [75] containing a Virtex-5 FPGA and five large, fast RAMs. Since these RAMs are all accessible in parallel, a wide heap could be obtained using off-chip storage. Further, the Virtex-5 would offer many more block RAMs, permitting larger on-chip stack and code memories accessed in parallel as in the existing design.

2.10.2 Clock frequency

Another benefit of the Virtex-5 over the Virtex-II is higher performance. The Xilinx synthesis tool clocks the current Wide Reduceron implementation at 160 MHz on the Virtex-5.

Identifying and reducing the critical path might yield further improvements in clock rate. The synthesis tool gives useful information about the critical path, but only at the net-list level. There is currently no facility to trace the critical components in the Lava description. In future, Lava could be

extended to allow descriptive labels to be attached to wires. Wires appearing in the final net-list would then contain information about their origin or purpose, giving meaning to the critical path reported by the synthesis tool.

2.10.3 Storage efficiency versus logic efficiency

There are alternative ways to configure and cascade block RAMs. For example, block RAMs can by configured as 1k by 18-bit, or as 16k by 1-bit. Instead of making a 16k by 18-bit memory by multiplexing the outputs of 16 1k by 18-bit block RAMs, an alternative would be to use 18 16k by 1-bit block RAMs without the need for a multiplexor, simply by concatenating the data busses. The latter approach saves logic at the cost of a few extra block RAMs.

Another implementation choice is whether to use quad-word memories or to quadruple the word-size and word-aligned applications. Quad-word memories have the advantage that nodes can be addressed individually, but they require extra logic on the address and data busses of block RAMs. Quadrupling the word-size requires no such extra logic, but nodes would have to be appropriately aligned on word-boundaries, probably wasting storage due to unused nodes inside words.

In each case, the choice is between efficient storage and efficient logic. The current Reduceron implementations have opted for efficient storage. Aiming for efficient logic could permit single-cycle memory-reads, and hence faster reduction.

2.10.4 Indirections

When a redex is reduced, the location of the redex on the heap is overwritten with an indirection to the result. The main worry when introducing such indirections is that long chains of indirections may build up. The following paragraphs consider this issue in more detail.

Indirection chains

When unfolding a function, the Reduceron instantiates the body of the function on the heap and immediately unwinds the spine onto the stack. For simplicity, the spine is unwound in exactly the same way as an application node of the form `Ap` i appearing on top of the stack. However, unwinding a spine is a special-case, and there is information that can be exploited to make it more efficient.

To illustrate, suppose that a function application `f x` resides at address a on the heap, and that `Ap` a appears on top of the stack. After unwinding `Ap` a, the stack, which grows downwards, looks as follows.

Stack	**Address stack**
`x`	a
`f`	$a + 1$

(Recall that application sequences are stored in reverse order on the heap, so indeed `x` is at address a and `f` is at address $a + 1$.)

Now suppose that `f` is defined as `f x =` `g` e for some unary function `g` and expression e. To apply `f` the spine `g` e is instantiated on the heap, say at address b. After unwinding the spine, the stack looks as follows.

Stack	**Address stack**
e	b
`g`	$b + 1$

To ensure that this reduction is not performed again, the indirection node `End` (`Ap` b) is written to address a on the heap. That is, the value at a is made to point to b.

Just as reduction of `f` leads to an indirection, so too could the reduction of `g`. That is, b could be made to point to some address c where the spine of an instantiated body of `g` is located. In this way, reduction can lead to a *chain of indirections*. If the original application node `Ap` a is evaluated several times, then a potentially long chain of indirections must be followed each time.

This particular form of indirection chain is easily solved: when unwinding a spine, do not overwrite the top element of the address stack. For example,

after unfolding `f` the stack should look as follows.

Stack	**Address stack**
e	a
`g`	$b+1$

Since the spine at address b has only just been created, no node in the graph can point to it. So a can be safely be used to point directly to the result of evaluating the spine, avoiding a chain of indirections.

Implementing this improvement in the PC Reduceron leads to a five percent speed-up on average across the seven benchmark programs.

More indirection chains

Still, indirection chains can build up. Consider the function `f`

```
f n m = if n == 0 then m else f (n-1) m
```

which is compiled down to the following combinator.

```
f n m = 0 (n (==)) m (f (n-1) m)
```

During evaluation, every call to `f n m`, where `n` is non-zero, is overwritten with an indirection to an application `f (n-1) m`. As a result, an indirection chain of length `n` is formed.

In the above example, the indirection chain results from the way in which case expressions are compiled. When matching against a zero-arity data constructor, such as `False`, a case alternative is represented by an expression which may be a redex. Since a redex may be shared, an indirection is needed to record that the original case expression reduces to this case alternative. Had the case alternative been represented by a partially-applied function application (as it would if it matched against a non-zero-arity data constructor), it would not be a redex, and no such indirection would be introduced.

Avoiding indirections altogether

The simplest way to avoid creating indirection chains is to avoid introducing indirections in the first place. This can be done by overwriting a redex

directly with its result, rather than an indirection to its result. The reason why the Reduceron does not do this is because the amount of space occupied by the current redex is both variable and unknown (and would be costly to determine). If the Reduceron used fixed-width, word-aligned applications (discussed in Section 2.10.3), overwriting a redex directly with its result would be much easier to do. Another possibility would be to employ the updating mechanism of the spineless G-machine [15], which evaluates redexes to normal form before updating them.

2.10.5 Template instantiation versus the G-machine

Section 2.2.5 pointed out that the interpretive overhead of template instantiation can be avoided using the compiler-based approach of the G-machine when implementing graph reduction on a PC. Section 2.3 pointed out that this interpretive overhead can also be removed by implementing a special-purpose machine. As template instantiation is somewhat simpler, it was chosen as a basis for the Reduceron. However, according to Peyton Jones the G-machine has another, separate advantage:

> *it turns out that compilation also opens the door to a whole host of short-cuts and optimisations which are simply not available to the template instantiation machine.* [80]

To illustrate, consider the following function `f`.

```
f x = g (h x) x
```

Based on compile-time knowledge about `g` and `h`, here are three optimisations that can be performed by a G-machine implementation.

1. Suppose that `g` is a function of two arguments. Normally, the spine of the body of `f` would be instantiated on the heap and its spine immediately unwound onto the stack. But this is unnecessary: the spine can be built directly on the stack because the body of `f` is known to be a redex. Furthermore, overwriting the root of the redex under evaluation can be avoided because it is not yet in normal form.

2. Suppose that `g` is known to be strict in its first argument. Normally, the application `h x` would be instantiated on the heap and passed as

an argument to `g`. But this is unnecessary: `h x` can be evaluated before entering `g`.

3. Suppose that `g` and `h` are both known to be strict in their arguments. Normally, `x` would be evaluated twice, and on the second occasion `x` would be an indirection to an already-evaluated expression. The second evaluation is unnecessary, and dereferencing the indirection can be avoided.

Most of the G-machine optimisations aim to reduce the number of expressions built on the heap. An avenue for future work is therefore to investigate if such optimisations could be employed in the Reduceron and, if so, how much benefit they might bring to a machine in which constructing and updating expressions is cheap.

2.10.6 Case expressions

Consider again the treatment of case expressions by Jansen's method (Section 2.2.2). A case expression containing n alternatives is transformed into an application of the form

$$e\ (f_1\ v_{1,1}\ \cdots\ v_{1,\star})\ \cdots\ (f_n\ v_{n,1}\ \cdots\ v_{n,\star})$$

where e is the case subject, f_1 to f_n are functions representing each alternative, and $v_{i,1}$ to $v_{i,\star}$ are the free variables referenced in the case alternative i (this set of free variables does not include the variables bound by each case alternative's pattern). One problem with this approach is that some nodes are instantiated on the heap for every case alternative even though only one of them will be chosen. In contrast, the G-machine evaluates the case subject first and only instantiates the chosen case alternative on the heap.

For small values of n, for example when scrutinising lists and pairs, this problem is a minor one. For larger values, the problem is still fairly minor for the Wide Reduceron since instantiation is cheap. But for the Narrow Reduceron, the problem is quite serious.

Another way to deal with a case expression of n alternatives would be to

transform it to an application of the form

$$e\ f_1\ \cdots\ f_n\ v_1\ \cdots\ v_n$$

where e is the case subject, f_1 to f_n are functions representing each alternative, and v_1 to v_n are the free variables referred to all case alternatives. By abstracting out free variables in this way, each one is written onto the heap only *once*. The function corresponding to each case alternative can simply ignore the free variables that it does not refer to. Furthermore, since the sequence of function identifiers f_1 to f_n is constant, it could be stored in code memory rather than written to the heap every time the case expression is to be reduced. So another topic for future work is to explore more efficient ways to compile case expressions.

2.10.7 Memory utilisation

A major limiting factor for memory utilisation is that application nodes are typically only one to five words in size. One possible way to widen such application nodes might be to proceed as follows. Consider the data type for lists

```
data List a = Nil | Cons a (List a)
```

and the following two functions.

```
fromTo n m      = if n > m then Nil else
                      Cons n (fromTo (n+1) m)
sum Nil         = 0
sum (Cons x xs) = x + sum xs
```

Suppose that the definition of the `List` data type can be transformed to the following definition `List2`.

```
data List2 a = Nil | Cons1 a (List a) | Cons2 a a (List a)
```

Such a transformation could be repeated, introducing progressively wider constructors. Now suppose that the functions `fromTo` and `sum` can be transformed, with the help of the law

$$\texttt{Cons2}\ x\ y\ z\ =\ \texttt{Cons1}\ x\ (\texttt{Cons1}\ y\ z)$$

```
fromTo2 n m = if n > m then Nil else
                if n+1 > m then Cons1 n Nil else
                  Cons2 n (n+1) (fromTo2 (n+1+1) m)
sum2 xs     = case xs of
                Nil -> 0
                Cons1 x0 xs -> x0 + sum2 xs
                Cons2 x0 x1 xs -> x0 + x1 + sum2 xs
```

Figure 2.22: Functions `fromTo` and `sum` after a widening transformation.

to `fromTo2` and `sum2`, defined in Figure 2.22. On a special-purpose machine with wide, parallel memories, the resulting functions should be capable of generating and consuming multiple list elements at a time.

Another approach to increasing the size of applications would be to represent applications as 1-level deep trees rather than flat sequences.

2.11 Related work

2.11.1 Hardware reduction machines

In the FPCA series of international conferences held between 1981 and 1995, several papers presented designs of exotic new machines to execute functional programs efficiently. Some special-purpose, sequential graph reduction machines were indeed built, including SKIM [88] and NORMA [85]. Unfortunately, at the time, building such machines was a slow and expensive process, and any performance benefit obtained was wiped out by the next advancement in stock hardware. In contrast, the Reduceron implementation is extremely cheap thanks to FPGAs, and FPGAs are an advancing technology in their own right. Another big difference from the Reduceron is that both SKIM and NORMA were based on Turner's combinators and did not attempt to use wide, parallel memories to increase performance.

2.11.2 The Big Word Machine

A piece of work similar in spirit to the Reduceron is Augustsson's Big Word Machine (BWM) [8], although the Reduceron was designed without knowl-

edge of the BWM. The BWM is a graph reduction machine with a wide word size, specifically four pointers long, allowing wide applications to be quickly built on, and fetched from, the heap. Augustsson likens the BWM to a VLIW (very long instruction word) machine [39], designed for functional languages rather than scientific computing. Like the Reduceron, the BWM has a crossbar switch attached to the stack allowing complex rearrangements to be done in a single clock-cycle. The BWM also encodes constructors and case expressions using functions and applications respectively. Unlike the Reduceron, the BWM works on an explicit, sequential instruction stream rather than by template instantiation. Features of the Reduceron not present in the BWM include

- separate code and heap memories,
- machine integer support,
- less memory wastage as applications need not be aligned on four-pointer boundaries,
- and support for building multiple different function applications on the heap simultaneously.

The BWM was never actually built. Some simulations were performed but Augustsson writes "the absolute performance of the machine is hard to determine at this point" [8].

2.11.3 Ward's work

Ward explores three different ways to implement lazy functional languages on FPGA [97]. A template-instantiation machine written in Handel-C, a compiler to Handel-C, and a new parallel reduction machine are all considered.

To illustrate one of Ward's main ideas, reconsider the three main steps performed by a graph reducer running on a standard PC (originally discussed in Section 2.2.5).

1. Fetch CPU instructions from memory which, when executed,
2. read the function body from template memory, and

3. construct an instance of the function body in graph memory.

The G-machine eliminates step 2, and a special-purpose template instantiation machine, such as the Reduceron, eliminates step 1. Ward proposes, in essence, to eliminate both steps 1 and 2 by compiling functions directly into a hardware which, when executed, instantiates the function body in graph memory. In other words, instructions are stored directly in registers on the FPGA. One of the potential problems with this approach is that every function definition requires its own connection to graph memory. As the number of functions in a program grows, so too does the number of sites from which graph memory is accessed. It may prove difficult to implement this efficiently. Another potential problem is scalability: a lot of logic would be required to store programs in registers.

Ward writes that "since the testing of the machine was not satisfactorily finished, no serious attempt was made to implement it in hardware" [97]. Ward does discuss the use of parallel memories in his parallel reduction machine, but the performance of this machine is not determined.

2.11.4 The statically allocated functional language

A quite different approach to running functional programs on FPGA is taken by Sharp and Mycroft [70]. They present SAFL, the statically allocated functional language, and show how it can be compiled to FPGA. "Statically allocated" means that functions are restricted to be first-order, strict and tail-recursive (or not recursive at all). The motivation for compiling functional programs to hardware is that they contain a lot of fine-grained implicit parallelism. To illustrate, consider the following SAFL program:

```
fun mult(x, y, acc) =
  if (x = 0 or y = 0) then acc
  else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)
```

Here `x` and `y` are the inputs to be multiplied, `acc` is an accumulator, and the return value is the result of the multiplication. Each parameter in the recursive call to `mult` is evaluated in parallel and so too is each side of the `or` expression.

The static-allocation restrictions are quite prohibitive for general-purpose

programming. However, Sharp demonstrates that pure SAFL extended with channels and mutable arrays is useful by implementing a DES encryption/decryption circuit [86]. Frankau extends SAFL with similar features, but in a purely functional manner through lazy lists and linear-typed arrays [29]. Both static allocation and purity are enforced using a linear type system.

Chapter 3

The Essence of Circuit Description

The previous chapter detailed how the Reduceron works, but little was said about the precise method used to implement it on an FPGA. This chapter presents the *functional programming* approach used to describe the Reduceron. Indeed, circuit description is a well-studied application of functional programming [87], but the Reduceron is a somewhat larger, more stateful, and less regular circuit than many of the examples considered in the literature. As a result, new issues are encountered that have not been dealt with in detail before.

3.1 Introduction

There are two quite different yet complementary ways to describe a digital circuit. One is to say what components are present and how they are connected together (*structural* description), and the other is to write a program which is compiled down to a circuit (*behavioural* description).

Standard hardware description languages (HDLs), such as VHDL [6] and Verilog [89], support both styles of description. They provide language constructs for structural description, such as component instantiation and replication, and also for behavioural description, such as event-driven processes

and signal assignment. These constructs can be freely mixed, allowing flexible combinations of the two styles.

This chapter explores an alternative approach using the *pure structural*, *higher-order* language Lava [19]. "Pure structural" means that Lava does not provide any built-in constructs for behavioural description, and "higher-order" means that Lava components can take components as inputs, and produce components as outputs. The approach taken here differs from that taken by standard HDLs in that behavioural constructs are *not built in* to the language, but are provided by a *small library* of pure structural, higher-order components.

The distinction made by circuit designers between structural and behavioural descriptions is similar to that made by functional programmers between *pure* and *impure* functions. Just as functional programmers can neatly simulate a range of impure features using pure abstractions, so circuit designers can neatly simulate a range of behavioural features using structural abstractions. Wadler refers to the former capability as "the essence of functional programming" [94], hence the titling of this chapter to capture the latter capability.

This chapter is primarily concerned with circuit descriptions that can be automatically synthesised to digital circuits of primitive components such as logic gates, flip-flops, and RAMs. All circuit descriptions presented run on an FPGA. However, no knowledge of FPGAs is assumed.

This chapter is structured as follows. Section 3.2 reviews the standard hardware description language *VHDL*, and illustrates some of the constructs it provides for structural and behavioural description. Section 3.3 reviews the pure structural language Lava, and illustrates some commonly-used *Lava* features such as higher-order components. Section 3.4 presents a small Lava library for behavioural description called Recipe – no language extensions to a pure structural language are required. Sections 3.5, 3.6 and 3.7 apply Lava and Recipe to three applications, the third of which is a stack processor (and associated compiler and abstract machine) that is highly-illustrative of the Reduceron. Section 3.8 discusses the strengths and weaknesses of the approach. Section 3.9 discusses related work and Section 3.10 summarises.

3.2 VHDL

Before exploring an *alternative* approach to hardware description, it is first useful to consider existing approaches. In this section, VHDL [6], a standard, widely-used HDL, is introduced. VHDL supports circuit description at a variety of abstraction levels, but this section focuses on the *register-transfer level* (RTL) subset [6] that can be synthesised by most VHDL tools.

3.2.1 Structural description

VHDL's unit of abstraction is the *component*. Every component typically has both an `entity` and an `architecture`. The entity lists the *ports* of the component (its inputs and outputs) and their types. For example, the entity for a 2-input NOR gate can be defined as

```
entity norGate is
  port (o : out std_logic; i1, i2 : in std_logic);
end norGate;
```

The architecture describes the relation between the input and output ports of a component, and can be defined *structurally* or *behaviourally*. Assuming a library providing the primitive components `inv` and `or2`, the following structural architecture for `norGate` can be defined.

```
architecture structural of norGate is
  signal tmp : std_logic;
begin
  c0 : or2 port map (tmp, i1, i2);
  c1 : inv port map (o, tmp);
end structural;
```

Here, two components are *instantiated* using a `port map` and labelled `c0` and `c1`. Signals are connected to the ports of a component by parameter passing. The `tmp` signal represents an intermediate wire, used to connect the output of `or2` to the input of `inv`.

VHDL also provides constructs for *conditional* and *repetitive* instantiation of components using the `if/generate` and `for/generate` statements. Component instantiations may even be recursive, as advocated by Ashenden in [5]. To illustrate, consider describing an OR-tree. A four-input instance of

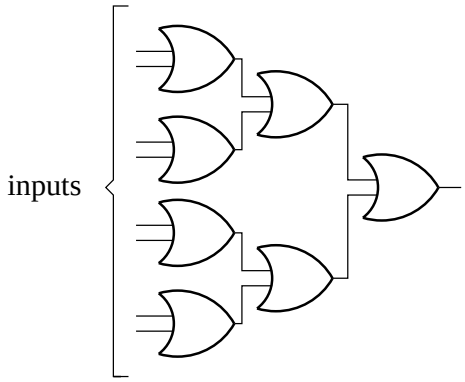

Figure 3.1: Schematic of a four-input OR-tree.

an OR-tree is depicted in Figure 3.1. A suitable entity declaration is

```
entity orTree is
  generic (width : positive);
  port    (o     : out std_logic;
           ins   : in  std_logic_vector(width-1 downto 0));
end entity orTree;
```

The `orTree` component takes a *generic* parameter `width` – a compile-time value specifying how many inputs the OR-tree operates on. It also takes an *array* of inputs – `width` elements long – and produces a single output representing the logical disjunction of all the inputs. Using a combination of conditional and recursive instantiation, a structural architecture for `orTree` is defined in Figure 3.2. Note that the assignment operator `<=` is used to connect two wires together.

An important point is that a structural description is essentially a *circuit generator*, that is, a program that when executed generates a representation of a circuit. In other words, the conditional statements and recursion are expanded out by the VHDL tool at compile time, leaving a *netlist* of primitive components and their connections.

3.2.2 Behavioural description

As well as structural architectures, *behavioural* architectures are also possible in VHDL. The main behavioural abstraction provided is the `process`,

```
architecture structural of orTree is
  signal lout, rout : std_logic;
begin
  baseCase: if width = 1 generate
    o <= ins(0);
  end generate;
  recursiveCase: if width > 1 generate
    left  : entity orTree
              generic map (width/2)
              port map (lout, ins(width/2-1 downto 0));
    right : entity orTree
              generic map ((width+1)/2)
              port map (rout, ins(width-1 downto width/2));
    join  : or2
              port map (o, lout, rout);
  end generate;
end structural;
```

Figure 3.2: Structural architecture of an OR-tree in VHDL.

which consists of a *sequence of statements* and a *sensitivity list*. Whenever a signal in a process's sensitivity list changes, that process is marked as *runnable*. Behavioural architectures are executed in a step-wise fashion. In each step, all runnable processes are executed in parallel. Signals which change during that execution, or as a result of external stimuli, decide the runnable processes for the next step. Any signal assignments (using the `<=` operator) that occur in one step become observable in the next step. Typically, sensitivity lists contain the clock input, so processes run on every clock-cycle; assignments become observable in the next clock-cycle.

To illustrate, consider describing a circuit to multiply two numbers together using an adder, a left-shifter and a right-shifter. The circuit takes two bit-vectors as input, representing the two numbers to multiply, and produces an output bit-vector representing the product of the two numbers. In addition, it takes a `start` bit signifying when the inputs are ready and that the circuit should begin computing the result, and it produces a `finished` bit signifying when the computation has finished and that the result is ready. The intended pattern of use is that the user of the multiplier will pulse the `start` bit (by setting it high for one clock-cycle), wait a number of clock-cycles until the `finished` bit becomes high, and then read the result.

```
entity mult is
  generic (width    : positive);
  port    (clock    : in std_logic;
           a, b     : in std_logic_vector(width-1 downto 0);
           start    : in std_logic;
           result   : out std_logic_vector(width-1 downto 0);
           finished : out std_logic);
end mult;

architecture behavioural of mult is
  signal regA, regB, acc : std_logic_vector(width-1 downto 0);
  signal currentlyRunning : std_logic;
begin
  -- Connect a few wires together
  finished <= not currentlyRunning;
  result <= acc;

  -- The following process runs every time the clock changes
  shiftAndAdd : process(clock) is begin
    if rising_edge(clock) then
      if start = '1' then -- Initialise!
        regA <= a;
        regB <= b;
        acc <= (others => '0');
        currentlyRunning <= '1';
      elsif currentlyRunning = '1' then
        if regB = 0 then
          currentlyRunning <= '0'; -- Finished!
        else
          regA <= regA(width-2 downto 0) & '0'; -- Shift left
          regB <= '0' & regB(width-1 downto 1); -- Shift right
          if regB(0) = '1' then acc <= acc + regA; end if;
        end if;
      end if;
    end if;
  end process;
end behavioural;
```

Figure 3.3: Behavioural description of a sequential multiplier in VHDL.

```
low   :: Bit                  -- Logic 0
high  :: Bit                  -- Logic 1
inv   :: Bit -> Bit           -- Inverter
(<&>) :: Bit -> Bit -> Bit    -- AND-gate
(<|>) :: Bit -> Bit -> Bit    -- OR-gate
(<#>) :: Bit -> Bit -> Bit    -- XOR-gate
delay :: Bit -> Bit -> Bit    -- Flip-flop
```

Figure 3.4: Operations of Lava's `Bit` ADT.

A VHDL description capturing this circuit's behaviour is shown in Figure 3.3. The most interesting part of the description is the `shiftAndAdd` process. Its sensitivity list contains the clock signal, ensuring that the process runs every time the clock changes. A conditional statement within the process additionally ensures that it only runs on the rising edge of the clock, that is, once per clock-cycle. The process contains a simple state machine. If it is 'currently running', the multiplicand is shifted right, the multiplier is shifted left, and if the multiplicand is odd the multiplier is added to the accumulator. When the multiplicand becomes zero, the 'currently running' flag is disabled. When not 'currently running' and a start signal is received, the process initialises the shift registers and the accumulator, and enters the 'currently running' state.

3.3 Lava: a pure structural language

A rather different approach to circuit description is taken by Lava [19], a domain-specific language embedded in Haskell. In Lava, components are modelled as Haskell *functions* from *inputs* to *outputs*. Inputs and outputs are Haskell *data structures* containing values of type `Bit`[1]. The `Bit` type is an abstract data type (ADT) providing the operations shown in Figure 3.4.

In Lava, components are connected together by writing applicative expressions. To illustrate, `norGate` can be defined as

```
norGate :: Bit -> Bit -> Bit
norGate a b = inv (a <|> b)
```

[1] Actually they have type `Signal Bool`, but `type Bit = Signal Bool` is assumed here.

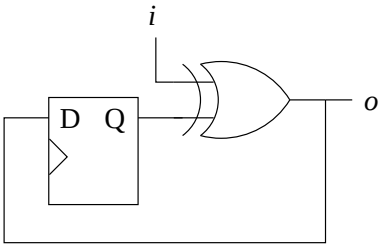

Figure 3.5: Schematic of a sequential parity checker.

Whereas VHDL provides arrays to represent sequences of bits, Lava provides inductively-defined *lists*. For example, in Lava the `orTree` circuit takes a list of bits and returns back a single bit.

```
orTree :: [Bit] -> Bit
orTree xs = tree (<|>) xs
```

It is defined in terms of `tree`, which can reduce a given list of bits using any given binary operator in a tree structure.

```
tree :: (a -> a -> a) -> [a] -> a
tree f [x] = x
tree f (x:y:ys) = tree f (ys ++ [f x y])
```

Like in the VHDL version, conditionals (now in the form of pattern matching) and recursion are used, and are expanded out at circuit-generation time to leave a netlist containing only instances of the primitive components in Figure 3.4 and their connections.

Unlike in the VHDL version, a *higher-order*, *polymorphic* component `tree` is defined. It is parameterised by the binary operator applied at each internal node of the tree, and this operator can take inputs of any type, not just bits. Such higher-order components are commonly used in Lava descriptions [87].

Another difference between Lava and VHDL is that Lava does not provide any built-in constructs for behavioural description. This is not to say that Lava cannot express stateful circuits like the sequential multiplier. For example, consider the simpler stateful circuit `oddParity`, depicted in Figure 3.5, that outputs high if the number times it has received a high input is odd, and outputs low otherwise.

```
oddParity :: Bit -> Bit
oddParity i = o
  where o = i <#> delay low o
```

(Here, `delay` denotes a D-type flip-flop; its first argument is the initial state of the flip-flop, and its second is the D-input.)

The problem is that as the numbers of stateful delay components and feedback loops increase, it becomes more and more difficult to understand structural descriptions. So although it would be possible to structurally express the sequential multiplier, it would be somewhat awkward to do so. Behavioural description, such as that provided by VHDL, abstracts away from flip-flops and feedback loops through mutable assignment and iteratively-executed processes.

3.4 Recipe: a library for behavioural description

In summary so far, VHDL supports structural and behavioural description, whereas Lava only supports structural description. On the other hand, Lava offers powerful abstraction capabilities through higher-order components. The aim of this section is to show that behavioural constructs can be neatly captured in a pure structural language by a small set of higher-order Lava components. This set of components is defined as a Haskell module named Recipe.

The Recipe library follows the approach to behavioural description taken by Page and Luk in their hardware variant of Occam [77]. This gives a behavioural style of description similar to VHDL, but at a slightly higher level of abstraction.

Before presenting the Recipe library, it is helpful to introduce a few building-blocks that will later prove useful. Section 3.4.1 defines some useful Lava components and Section 3.4.2 briefly reviews *monads* [95], used to structure the Recipe library.

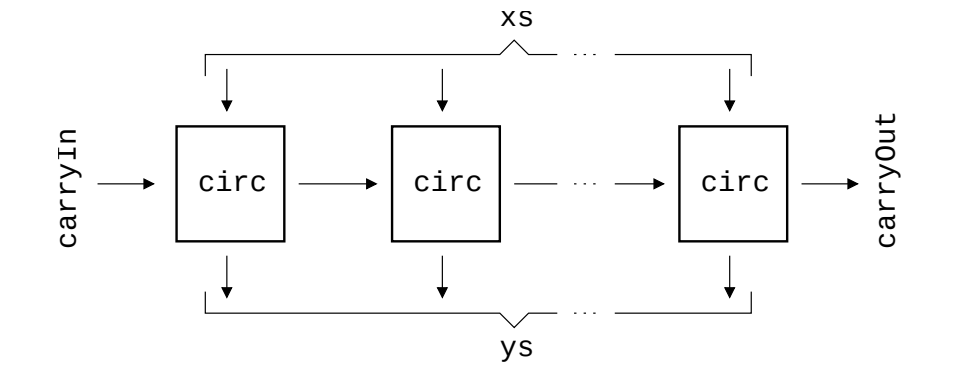

Figure 3.6: Lava's `row` pattern.

3.4.1 Some useful components

A multiplexor The `pick` component takes a list of bit/bit-list pairs and returns the bit-list that is paired with `high`. It is assumed as a pre-condition that *exactly one* bit-list is paired with `high` in the input list.

```
pick :: [(Bit, [Bit])] -> [Bit]
pick xs = map (tree (<|>)) (transpose ys)
  where ys = [map (b <&>) x | (b, x) <- xs]
```

To illustrate, the expression

```
pick [(low, [low, high]), (high, [high, low])]
```

evaluates to `[high, low]`.

A row A `row` is an often used Lava description, and indeed is provided by the Lava prelude. It captures the pattern shown Figure 3.6 and is defined as follows.

```
row :: ((a, b) -> (c, a)) -> (a, [b]) -> ([c], a)
row circ (carryIn, []) = ([], carryIn)
row circ (carryIn, x:xs) = (y:ys, carryOut)
  where (y, carry) = circ (carryIn, x)
        (ys, carryOut) = row circ (carry, xs)
```

A register with input-enable The `delayEn` component is like `delay`, but takes an input-enable line `en` that decides whether the D-input should

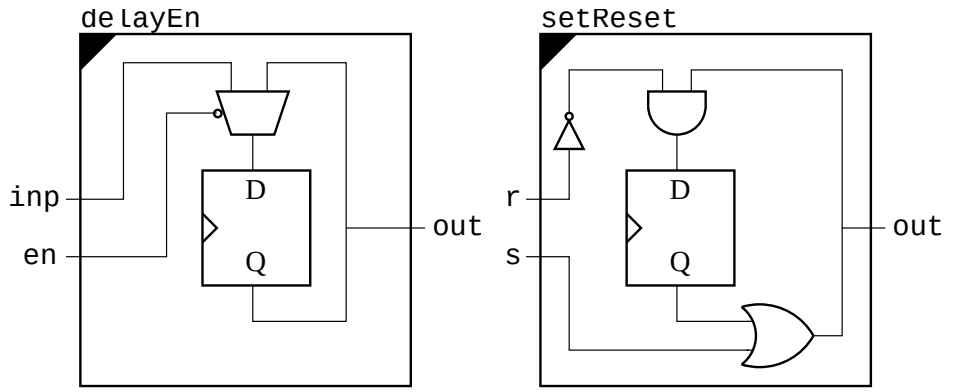

Figure 3.7: Delay with input enable (left) and a set-reset latch (right).

be latched into the flip-flop, or whether the flip-flop should retain its current state.

```
delayEn :: Bit -> Bit -> Bit -> Bit
delayEn init en inp = out
  where out = delay init (mux2 (inv en) inp out)


mux2 :: Bit -> Bit -> Bit -> Bit
mux2 sel a b = (inv sel <&> a) <|> (sel <&> b)
```

The structure of `delayEn` is depicted on the left in Figure 3.7. A register with input-enable can be formed from a series of `delayEn` components.

```
regEn :: [Bit] -> Bit -> [Bit] -> [Bit]
regEn [] en inps = []
regEn (x:xs) en inps = delayEn x en y : regEn xs en ys
  where y:ys = inps
```

A set-reset latch The `setReset` component has an internal state, initially low, and takes two inputs: set and reset. If the set line is high then the latch outputs high, otherwise it outputs its internal state. If the set line is high and the reset line is low then the internal state is set to high on the next clock cycle. If the reset line is high, then the internal state is set to low on the next clock-cycle. If neither line is high, the latch does not change state.

```
setReset :: Bit -> Bit -> Bit -> Bit
setReset init s r = out
  where q   = delay init (out <&> inv r)
        out = s <|> q
```

The structure of `setReset` is depicted on the right in Figure 3.7.

3.4.2 Monads

In Haskell, a monad [95] is an abstract data type, parametrised by some other type, that is a member of the following type class.

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

An expression of the form `return a` denotes a computation that simply returns `a` without performing any side-effect. And one of the form `c >>= f` denotes a computation that sequences the two computations `c` and `f a`, where `a` is the value returned by `c`. Sometimes it is useful to ignore the result of the first computation. This can be done using the `>>` combinator.

```
(>>) :: Monad m => m a -> m b -> m b
c0 >> c1 = c0 >>= \_ -> c1
```

Any implementation of `return` and `>>=` must satisfy the following three monad laws which state that `>>=` is associative and `return` is its unit.

$$\begin{aligned} \texttt{return}\ x\ \texttt{>>=}\ f &= f\ x \\ m\ \texttt{>>=}\ \texttt{return} &= m \\ (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g &= m\ \texttt{>>=}\ (\lambda x.\ f\ x\ \texttt{>>=}\ g) \end{aligned}$$

To be more concrete, the following section considers a specific instance of a monad that captures the side-effect of *state*.

The state monad

One kind of side-effect that is useful for some computations to have is *state*, whereby a value is implicitly threaded through a sequence of computations, and each individual computation can read, modify or ignore it. A stateful

computation can be represented as a transition function from the current state to a pair containing the next state and the return value of the computation [95].

```
data State s a = State (s -> (s, a))
```

Here, the type state of state contained by a stateful computation is parameterised by the type variable `s`. Using a helper function to extract the transition function from a stateful computation,

```
run (State f) = f
```

the type `State s` can be made a monad, for any `s`.

```
instance Monad (State s) where
  return a = State (\s -> (s, a))
  c >>= f  = State (\s -> case run c s of
                            (s', a) -> run (f a) s')
```

To abstract away from the internal representation of `State`, the operators `get` and `set` can be used.

```
get :: State s s
get = State (\s -> (s, s))

set :: s -> State s ()
set s = State (\_ -> (s, ()))
```

To illustrate, Figure 3.8 defines the stateful computation `example`. The key point is that the internal state, in this case a value of type `Int`, is *implicit* and need not be passed around manually. This mimics global state as found in an impure language. The expression `run example 0` where `0` is the initial state, evaluates to `(11, False)`.

3.4.3 The Recipe monad

Like in Page and Luk's hardware variant of Occam [77], all behavioural descriptions written using the Recipe library, referred to as *recipes*, take a *start* signal (a single bit) and produce a *finish* signal (also a single bit). A single-cycle pulse on the start signal indicates that the recipe should start executing. Similarly, when a recipe finishes executing, it should produce a single-cycle pulse on the finish signal. Recall that such start and finish

```
example :: State Int Bool
example =
  set 10    >>
  get       >>= \x ->
  set (x+1) >>
  get       >>= \y ->
  return (y < 10)
```

```
example :: State Int Bool
example =
  do set 10
     x <- get
     set (x+1)
     y <- get
     return (y < 10)
```

Figure 3.8: An example monadic computation (left) and the same example written using `do`-notation (right).

signals were expressed *explicitly* in the VHDL description of the sequential shift-and-add multiplier (Figure 3.3). The Recipe library abstracts over this common setup by making the start and finish signals *implicit* in *every* description. As a result, all recipes can be composed in a variety of useful ways, including sequential and parallel composition, iteration and choice.

The possibility of representing a recipe as a circuit taking a start signal and producing a finish signal naturally leads to its view as a stateful computation.

```
type Recipe a = State (Bit, Env) a
```

In this type synonym, the `Bit` represents the start/finish signal. An *environment*, of type `Env`, also forms part of the implicit state; its purpose is explained in Section 3.4.8.

Apart from the missing definition of `Env`, this completes the definition of the first two behavioural constructs provided by Recipe: the unit recipe (`return`) and sequential composition (`>>=`). The following sections define further constructs, inspired by Page and Luk [77].

3.4.4 Skip and tick

Sometimes the unit description is referred to as *skip*.

```
skip :: Recipe ()
skip = return ()
```

Another construct, similar to `skip` but which takes one clock-cycle to execute, is `tick`. This is implemented by simply passing the start signal through a flip-flop to produce the finish signal.

```
tick :: Recipe ()
tick = State (\(start, env) -> ((delay low start, env), ()))
```

New recipes can easily be defined in terms of existing ones. For example, the following construct is like `tick` but delays for any number of clock-cycles.

```
tickN :: Int -> Recipe ()
tickN 0 = skip
tickN n = tick >> tickN (n-1)
```

3.4.5 Choice

An if-then-else construct over recipes is defined as follows.

```
cond :: Bit -> Recipe () -> Recipe () -> Recipe ()
cond cond p q = State (\(start, env) ->
  let ((fin0, env0), _) = run p (start <&> cond, env)
      ((fin1, env1), _) = run q (start <&> inv cond, env0)
  in  ((fin0 <|> fin1, env1), ()))
```

It takes a condition bit `cond`, and two recipes, `p` (the *then*-branch) and `q` (the *else*-branch). Notice that `cond` is of type `Bit`, meaning that a pure Lava expression can be used to describe the condition. The conjunction of the start signal and `cond` is used to trigger `p`, and the conjunction of the start signal and the inverse of `cond` is used to trigger `q`. The final finish signal is the disjunction of the finish signals of `p` and `q`.

A related construct is the *guarded* recipe, which executes a recipe only if the guard holds.

```
(|>) :: Bit -> Recipe () -> Recipe ()
guard |> p = cond guard p skip
```

3.4.6 Iteration

A while-loop construct over recipes is defined as follows.

```
while :: Bit -> Recipe () -> Recipe ()
while cond p = State (\(start, env) ->
  let ((fin, env'), _) = run p (cond <&> ready, env)
      ready            = start <|> fin
  in  ((inv cond <&> ready, env'), ()))
```

The loop body `p` is said to be *ready* when the start signal of the loop is active, or when its own finish signal is active. However, it is only triggered when it is both ready and the loop-condition holds. If it is ready, and the loop-condition does not hold, then the overall finish signal is triggered.

Another kind of loop is `doUntil` which repeatedly executes the loop body until a condition becomes true. Unlike `while`, `doUntil` tests the condition at the *end* of each iteration.

```
doUntil :: Bit -> Recipe () -> Recipe ()
doUntil cond p = State (\(start, env) ->
  let ((fin, env'), b) = run p (ready, env)
      ready = start <|> (fin <&> inv cond)
  in  ((fin <&> cond, env'), b))
```

When using loops, the programmer must take care not to make a loop-body that takes no clock-cycles to complete – this would result in a combinatorial loop in the circuit! The `tick` construct plays an important role here by allowing the programmer to ensure that loop bodies take at least one clock-cycle.

Again, other useful constructs can be defined in terms of existing ones.

```
forever :: Recipe () -> Recipe ()
forever p = while high p
```

3.4.7 Parallel composition

A list of recipes can be composed in parallel using the `par` operator, defined in Figure 3.9. Parallel composition has a fork/join semantics: an expression of the form `par ps` starts every recipe in the list `ps` at the same time, and finishes only when *all* recipes in `ps` have finished. The joining behaviour is achieved by feeding the finish signal of each recipe into the set-line of a set-reset latch, and AND-ing the outputs of the latches to produce the overall finish signal. The overall finish signal is fed back into the reset-line of each latch, so that the `par` block is ready to be executed again if, for example, it occurs in the body of a loop.

```
par :: [Recipe ()] -> Recipe ()
par ps = State (\(start, env) ->
  let (fins, env') = row circ (env, ps)
      circ (env, p) = fst (run p (start, env))
      fin = tree (<&>) (map (\s -> setReset low s fin) fins)
  in  ((fin, env'), ()))
```

Figure 3.9: Parallel composition.

3.4.8 Mutable variables

In this section, operations for creating, reading, and writing variables are defined. Every mutable variable has a unique identifier and outputs a value of type `[Bit]`, as defined by the following data types.

```
data Var   = Var { varId :: VarId, val :: [Bit] }
type VarId = Int
```

Variables can be assigned to new values. Such assignments hold two pieces of information: a single bit that is pulsed on the clock-cycle in which the assignment should occur, and a list of bits representing the value to be assigned to the variable.

```
type Assignment = (Bit, [Bit])
```

Mutable variables come in two varieties: *signals* and *registers*. Each one attaches a different meaning to assignment.

- When a *register variable* is assigned a new value on a particular clock-cycle, its output does not reflect the new value until the next clock-cycle.
- When a *signal variable* is assigned a new value on a particular clock-cycle, its output immediately reflects the new value in that clock-cycle.
- Whereas the effect of the register assignment is visible in subsequent clock-cycles, the effect of the signal assignment is only visible during the clock-cycle in which the assignment is made. In clock-cycles where no assignment is made, a signal variable simply outputs zero.

Given a list of assignments that have been made to a signal variable, the output of the variable is defined by the following equation.

```
assignSig :: [Assignment] -> [Bit]
assignSig = pick
```

(Recall that `pick` is defined in Section 3.4.1.) A similar equation defines the semantics of register assignment.

```
assignReg :: Int -> [Assignment] -> [Bit]
assignReg width ass = regEn (replicate width low) en (pick ass)
  where en = tree (<|>) (map fst ass)
```

(Recall that `regEn` is defined in Section 3.4.1.) In this case, the assign-pulses of each assignment are OR-ed together and fed into the input-enable line of a register. In addition, the *width* of the register must be known for reasons explained in Section 3.4.9.

So far, little has been said about the *environment* that is implicitly threaded between recipes. The purpose of the environment is to provide a fresh-name supply for variables and to record the assignments made to each variable.

```
data Env = Env { freshId  :: VarId
               , readEnv  :: [(VarId, Assignment)]
               , writeEnv :: [(VarId, Assignment)] }
```

Here, there are *two* assignment mappings rather than one. The `writeEnv` mapping is *written to* as new assignments are made. The `readEnv` mapping is *read from* to determine the list of assignments made to a given variable. In Section 3.4.9, the two mappings will be tied together with a recursive knot [13]. Using this technique, it can be assumed that the `readEnv` mapping contains *all* the assignments made by an entire recipe.

An assignment of the form `v <== x` adds the pair `(varId v,(start,x))` to the list of assignments held in the `writeEnv` mapping.

```
(<==) :: Reg -> [Bit] -> Recipe ()
v <== x = State (\(start, env) ->
  let wenv = (varId v, (start, x)) : writeEnv env
  in  ((start, env { writeEnv = wenv }), ()))
```

A slight variant, `<=|`, does the same but consumes a clock-cycle.

```
(<=|) :: Reg -> [Bit] -> Recipe ()
r <=| x = r <== x >> tick
```

To create a variable, a fresh name is extracted from the environment, looked

```
newSig :: Int -> Recipe Var
newSig width = newVar assignSig

newReg :: Int -> Recipe Var
newReg width = newVar (assignReg width)

newVar :: ([Assignment] -> [Bit]) -> Recipe Var
newVar f = State (\(start, env) ->
  let v   = freshId env
      ass = [i | (w, i) <- readEnv env, v == w]
  in  ((start, env  freshId = v+1 ), Var v (f ass)))
```

Figure 3.10: Variable creation.

up in the `readEnv` mapping, and the resulting list of assignments to that variable is passed to `assignSig` or `assignReg` appropriately. This behaviour is captured by the functions `newSig` and `newReg` in Figure 3.10.

3.4.9 Following recipes

All that remains is to define how to turn a start bit and a recipe, of type `Recipe a`, into a finish bit and a value of type `a`. That is, how to follow a recipe.

```
follow :: Bit -> Recipe a -> (Bit, a)
follow start r = (fin, a)
  where ((fin, env), a) = run r (start, initialEnv)
        initialEnv      = Env 0 (writeEnv env) []
```

The crucial connection here is the feedback loop from the `writeEnv` mapping of the final environment to the `readEnv` mapping of the initial environment. This feedback loop is not surprising considering that a variable (say `v`) can be assigned to its own output, e.g. by the statement `v <== map inv (val v)`. In hardware, this statement corresponds to a circuit with a cycle. The possibility of such cycles occurring is the reason why register variables must have a pre-defined width, so that the spine of the list representing the output of a register does not form an unresolvable mutual dependency.

The following sections present three applications of the Recipe library.

3.5 Application 1: a sequential multiplier

On some FPGA devices, single-cycle multiplication is an expensive operation in terms of both critical-path delay and resource usage. It is therefore handy to have a *sequential* multiplier available – one that takes several clock-cycles to complete, but which has a much smaller combinatorial delay and requires less FPGA resources. This section defines such a multiplier using Recipe.

3.5.1 Numbers in Lava

In Lava, numbers are typically represented as a lists of bits, with the least significant bit coming first. More formally, a non-negative integer `n` is encoded as a list of `m` bits by the following function.

```
ofWidth :: Int -> Int -> [Bit]
0 `ofWidth` m = replicate m low
n `ofWidth` m = b : (n `div` 2) `ofWidth` (m-1)
  where b = if odd n then high else low
```

Like the earlier VHDL description (Figure 3.3), the multiplier will be implemented using the shift-and-add algorithm. The following ingredients are needed first: functions for adding two bit-lists together and shifting a bit-list left and right. The addition function is defined in Figure 3.11. The left and right shifting operations are defined as follows.

```
shl :: [Bit] -> Int -> [Bit]
a `shl` n = drop n a ++ replicate n low

shr :: [Bit] -> Int -> [Bit]
a `shr` n = reverse (reverse a `shl` n)
```

Again it is assumed that non-negative numbers are being manipulated – numbers are just padded with zeroes rather than doing sign-extension as would be required in a two's complement encoding.

3.5.2 Implementation

Quite often descriptions in Recipe contain expressions of the form f `(`g `(`h x`))`, for some functions f, g, and h. To simplify such expressions, the following

left-associative operator for reverse function application is used.

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

Now such expressions are written $x\texttt{!}h\texttt{!}g\texttt{!}f$.

Figure 3.12 defines the sequential multiplier in Recipe. The main difference from the VHDL version is that the start and finish signals are implicit and the code for initialisation can simply be sequentially composed with the code for shifting and adding. As a result, no explicit state machine is needed.

The multiplier requires one clock-cycle for initialisation and n clock-cycles for the loop in the worst case, where n is the length in bits of the second argument.

3.5.3 Correctness

If the multiplier is correctly defined, the following function should return true for all argument values

```
prop_mult :: Int -> Int -> Bool
prop_mult x y = x >= 0 && y >= 0 ==> x*y == multSim n x y
  where n = 2 * (1 + log2 (max x y))
```

where `multSim n x y` uses Lava to simulate `mult` (until it asserts its finish signal) on `n`-element bit-vectors representing integers `x` and `y`, and converts the resulting bit-vector to an integer. To avoid overflow, the result bit-vector must contain at least twice the number bits needed to store the maximum of `x` and `y`.

3.6 Application 2: a Lego brick-sorter

In "Reactive Systems Design 2007", a taught-course module at the University of York[2], the students were asked to write a program to control a Lego brick-sorter. This section presents a solution using Recipe. As programs were to run on a Lego Mindstorms RCX micro-controller, a C back-end for

[2]Organised by Gerald Lüttgen and Jan Tobias Mühlberg.

```
(/+/) :: [Bit] -> [Bit] -> [Bit]
a /+/ b = sum
  where (sum, carryOut) = row fullAdd (low, zip a b)

halfAdd (a, b) = (sum, carry)
  where sum   = a <#> b
        carry = a <&> b

fullAdd (carryIn, (a, b)) = (sum, carryOut)
  where (sum1, carry1) = halfAdd (a, b)
        (sum, carry2)  = halfAdd (carryIn, sum1)
        carryOut       = carry1 <#> carry2
```

Figure 3.11: Addition of bit-lists in Lava.

```
mult :: [Bit] -> [Bit] -> Recipe [Bit]
mult a b =
  do regA <- newReg (length a)
     regB <- newReg (length b)
     acc  <- newReg (length a)

     regA <== a
     regB <== b
     acc  <== 0 `ofWidth` length a
     tick

     while (tree (<|>) (regB!val))
       (do regA <== regA!val `shr` 1
           regB <== regB!val `shl` 1
           regB!val!head |> acc <== acc!val /+/ regA!val
           tick)

     return (acc!val)
```

Figure 3.12: Sequential multiplier in Recipe.

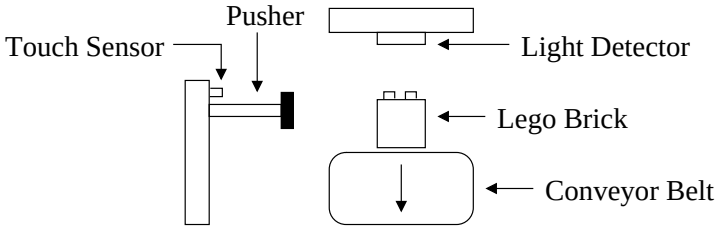

Figure 3.13: End-elevation view of the brick-sorter (the brick is moving towards you).

Lava was developed. The solution described below indeed runs correctly on the RCX.

3.6.1 Structure of the brick-sorter

The brick-sorter is composed of a four main components.

1. A conveyor belt, on which light or dark-coloured bricks may be placed.
2. A pusher arm, capable of pushing a brick off the conveyor belt when it reaches the *push point*.
3. A light-sensor sitting at the push point, capable of detecting light-coloured bricks.
4. A touch-sensor reporting whether or not the pusher arm is in its *resting position* or not. The resting position is the position in which the pusher arm is not blocking possible bricks on the conveyor belt.

Figure 3.13 gives an end-elevation view of the brick-sorter.

The brick-sorter should push all light-coloured bricks off the conveyor belt, and leave dark-coloured bricks untouched.

3.6.2 Operation of the brick-sorter

The intended operation of the brick-sorter is as follows.

1. *Initialise*. The pusher arm is moved into its resting position, where it touches the touch sensor. Note that the pusher arm is either *moving* or *not moving* – there is no notion of direction. Mechanically, the pusher

```
sorter :: (Bit, Bit) -> Recipe (Bit, Bit)
sorter (touch, light) =
  do belt <- newReg 1
     push <- newSig 1

     while (inv touch) (push <=| [high])

     belt <=| [high]
     forever (do while (inv light) tick
                 while touch (push <=| [high])
                 while (inv touch) (push <=| [high]))

     return (belt!val!head, push!val!head)
```

Figure 3.14: Controller for a Lego brick-sorter in Recipe.

arm will automatically retract towards the resting position when it reaches as far as it can stretch.

2. *Turn on belt.* Once the pusher arm is resting, the conveyor belt is turned on.

3. *Wait and push.* When the light-sensor detects a light-coloured brick, the pusher arm is enabled, pushing the brick off the conveyor belt. Once it returns to the resting position, the pusher arm is disabled again. Note that there is no need to stop the conveyor belt when a light-coloured brick is detected – the pusher arm is fast enough to hit the brick before it passes the push point.

3.6.3 Implementation

Figure 3.14 describes a controller for the brick-sorter using Recipe. As inputs, the description takes active-high bits from the touch and light sensors. As outputs, it returns active-high bits to drive the conveyor belt and pusher arm motors. The belt-enable bit is represented by a register variable and the push-enable bit by a signal variable; whereas the belt is enabled permanently after initialisation, the pusher is only enabled in clock-cycles in which the push-enable bit is assigned to `[high]`.

3.7 Application 3: a simple stack processor

The largest application of Recipe is, by some margin, the Reduceron graph reduction machine described in Chapter 2. This application is slightly too large to present in detail here. Instead, a simple 8-bit stack processor called Poly for Xilinx FPGAs is presented. Poly is designed to evaluate polynomial expressions containing possibly-many references to a *single* variable, written in the following language.

```
data Expr = X | N Int | Expr :+: Expr | Expr :*: Expr
```

Here, `X` represents the variable. The semantics of the language is as follows.

```
eval X           n = n
eval (N n)       _ = n
eval (e1 :+: e2) n = eval e1 n + eval e2 n
eval (e1 :*: e2) n = eval e1 n * eval e2 n
```

Poly implements the following six instructions.

```
data Instr = LIT Int | DUP | REV | ADD | MUL | HALT
```

The equations in Figure 3.15 define a compiler (`compile`) from expressions to instruction sequences and an abstract machine (`exec`) for executing such instruction sequences with the help of a stack. These definitions were originally developed by Colin Runciman as an exercise in proving the following theorem using a mechanised theorem prover.

$$\forall e, n.\ \texttt{exec (compile}\ e\texttt{) [}n\texttt{]} = \texttt{eval}\ e\ n$$

The example was subsequently re-used by Coquand et al. in a tutorial for a dependently-typed language [22].

The implementation of Poly, presented below, is highly illustrative of the implementation of the Reduceron.

3.7.1 Bytecode

To implement the six instructions, it is first necessary to define how they are encoded as sequences of bits. The following, simple encoding is used.

```
compile :: Expr -> [Instr]
compile e = comp e ++ [HALT]
  where comp X           = []
        comp (N n)       = [LIT n]
        comp (e1 :+: e2) = DUP:comp e2++[REV]++comp e1++[ADD]
        comp (e1 :*: e2) = DUP:comp e2++[REV]++comp e1++[MUL]

type Stack = [Int]

exec :: [Instr] -> Stack -> Int
exec (LIT m : c) (n:s)   = exec c (m:s)
exec (DUP   : c) (m:s)   = exec c (m:m:s)
exec (REV   : c) (m:n:s) = exec c (n:m:s)
exec (ADD   : c) (m:n:s) = exec c (m+n:s)
exec (MUL   : c) (m:n:s) = exec c (m*n:s)
exec (HALT  : c) (n:s)   = n
```

Figure 3.15: Compiler and machine for evaluating polynomial expressions.

```
encode :: Instr -> Int
encode i = case i of
             LIT n -> 1 + 2*n ; ADD  -> 8
             DUP   -> 2       ; MUL  -> 16
             REV   -> 4       ; HALT -> 32
```

If the least-significant bit of an instruction is high, then it is a `LIT n` instruction where `n` is defined by the remaining bits. Each remaining instruction is represented by a one-hot bit sequence whose least-significant bit is low.

In Lava, an instruction is represented more directly as a bit-sequence.

```
type Opcode = [Bit]
```

The Lava operations defined in Figure 3.16 allow decoding of such op-codes.

3.7.2 Memory

This section presents an abstraction over *memories* consisting of an abstract data type `Mem` with two operations `mread` and `mwrite` for reading from and writing to memory. A Xilinx block RAM is used to implement memory. The Lava component `bram8` represents a block RAM.

```
isLIT, isDUP, isREV, isADD, isMUL, isHALT :: Opcode -> Bit
isLIT   i = i !! 0
isDUP   i = inv (isLIT i) <&> i !! 1
isREV   i = inv (isLIT i) <&> i !! 2
isADD   i = inv (isLIT i) <&> i !! 3
isMUL   i = inv (isLIT i) <&> i !! 4
isHALT  i = inv (isLIT i) <&> i !! 5

getLIT :: Opcode -> [Bit]
getLIT i = drop 1 i ++ [low]
```

Figure 3.16: Routines for decoding op-codes.

```
data Mem = Mem { addr :: Var, inp :: Var
               , we   :: Var, out :: [Bit] }

newMem :: [Int] -> Recipe Mem
newMem init =
  do addrSig <- newSig 8
     inpSig  <- newSig 8
     weSig   <- newSig 1
     let out = bram8 init ( val addrSig
                          , val inpSig
                          , head (val weSig) )
     return (Mem addrSig inpSig weSig out)
```

Figure 3.17: Interface to memory.

```
bram8 :: [Int] -> ([Bit], [Bit], Bit) -> [Bit]
```

As inputs, it takes a list of 256 integers specifying the initial state of the block RAM, an 8-bit address bus, an 8-bit data-write bus, and a 1-bit write-enable signal respectively. As output, it produces an 8-bit data-read bus.

The abstract data type `Mem` is defined in Figure 3.17. The `newMem` function creates a value of type `Mem` containing signal variables that are connected to the input and output busses of a `bram8` instance.

To read the value at address `a` in memory `m`, the address `a` is placed on the address bus of `m`.

```
mread :: Mem -> [Bit] -> Recipe ()
mread m a = m!addr <== a
```

```
data RegFile = RegFile { mem    :: Mem
                       , pc     :: Var, sp  :: Var
                       , opcode :: Var, top :: Var }

newRegFile :: [Int] -> Recipe RegFile
newRegFile init =
  do m      <- newMem init
     pcReg  <- newReg 8
     spReg  <- newReg 8
     ocReg  <- newReg 8
     topReg <- newReg 8
     return (RegFile m pcReg spReg ocReg topReg)
```

Figure 3.18: Poly's register file.

To write a value `x` to address `a` in memory `m`, the address `a` is placed on the address bus of `m`, the value `x` on the data-write bus of `m`, and the write-enable signal of `m` is set.

```
mwrite :: (Mem, [Bit]) -> [Bit] -> Recipe ()
mwrite (m, a) x = do m!addr <== a
                     m!inp  <== x
                     m!we   <== [high]
```

3.7.3 Processor

The processor's register-file is defined in Figure 3.18. It contains a program counter (`pc`), a stack pointer (`sp`), an op-code register (`opcode`), and a top-of-stack register (`top`). It also contains an interface to memory (`mem`).

The processor is defined in Figure 3.19. It is parameterised by the initial state of memory. It reads the program bytcode starting at address `0`, and the stack grows downwards from address `255`. At the beginning of execution, address `255` contains the value of the variable referenced by the polynomial expression, and at the end of execution, it contains the result of evaluating the expression. The following two operators are used for incrementing and decrementing an eight bit list.

```
inc, dec :: [Bit] -> [Bit]
inc a = a /+/ (1 `ofWidth` 8)
dec a = a /+/ (255 `ofWidth` 8)
```

```
cpu :: [Int] -> Int -> Recipe [Bit]
cpu code x =
  do s     <- newRegFile code
     s!top <== x `ofWidth` 8        -- Initialise top of stack
     s!sp  <== 255 `ofWidth` 8      -- Initialise stack ptr
     tick

     let i = s!opcode!val           -- Shorter name
     doUntil (isHALT i) (do         -- Iterate until HALT
       mread (s!mem) (s!pc!val)     -- Fetch instruction
       s!pc <== s!pc!val!inc        -- Increment prog counter
       tick

       mread (s!mem) (s!sp!val!inc) -- Read top-but-one
       s!opcode <== s!mem!out       -- Save instruction
       tick

       -- Execute instruction
       par [ isLIT i |> s!top <== getLIT i
           , isDUP i |>
               do mwrite (s!mem, s!sp!val) (s!top!val)
                  s!sp <== s!sp!val!dec
           , isREV i |>
               do mwrite (s!mem, s!sp!val!inc) (s!top!val)
                  s!top <== s!mem!out
           , isADD i |>
               do let result = s!top!val /+/ s!mem!out
                  s!top <== result
                  s!sp <== s!sp!val!inc
           , isMUL i |>
               do result <- mult (s!top!val) (s!mem!out)
                  s!top <== result
                  s!sp <== s!sp!val!inc
           ]

       tick)

     return (s!top!val)
```

Figure 3.19: Stack processor for evaluating polynomials in Recipe.

3.7.4 Correctness

If the processor is correctly defined, the following function should return true for all argument values

```
prop_cpu :: Expr -> Int -> Bool
prop_cpu e x = eval e x == cpuSim bytecode x
  where bytecode = map encode (compile e)
```

where `cpuSim` uses Lava to simulate `cpu` (until it asserts its finish signal), and converts the resulting bit-vector to an integer.

3.8 Discussion

This section discusses the properties of the Recipe approach to circuit description presented in this chapter, including strengths, weaknesses, and areas for further work.

3.8.1 Implementation cost

The approach taken here has a low implementation cost: Recipe is a mere 170-line Lava module, fully-defined in the text of this chapter. Lava is itself a small Haskell library. Implementation is therefore cheap compared to building an autonomous language from scratch with support for both structural and behavioural description.

3.8.2 Testing

The fact that both Lava and Recipe are implemented in Haskell opens up the possibility of testing low-level circuit descriptions against high-level Haskell specifications. For example, Sections 3.5.3 and 3.7.4 stated properties relating circuit-level multiplier and processor descriptions to high-level Haskell functions. The next two chapters of this thesis are concerned with how such properties can, in general, be tested automatically. The two abovementioned properties will be considered again as examples in Section 5.6.5.

3.8.3 Efficiency

One Recipe construct of questionable efficiency is `par`. For example, the `cpu` description in Figure 3.19 uses `par` to perform a parallel multi-way case analysis. Even though only one case can be taken – because the conditions are non-overlapping – several set-reset latches are constructed, combining the finish signals to implement the join semantics. An alternative approach would be to provide a new construct like `par` but which combines finish signals with an OR-gate. The problem is that such a construct has confusing behaviour if cases *do* overlap – multiple finish pulses are potentially generated.

One possible way to improve efficiency in general would be to make the Recipe combinators build up an abstract syntax tree. Optimisations could then be applied to the abstract syntax before compilation to a circuit.

3.8.4 Static versus dynamic typing

Representing bit-sequences as inductively-defined lists is convenient, but the number of bits in the sequence does not form part of its type. This means that some circuit descriptions can be written that do not have a sensible meaning. For example, the assignment operator `<==` assigns a list of bits to a register, but the constraint that the width of register must be equal to the length of the list is not specified.

One possible solution is to use *number-parameterised types* [52] to represent bit-sequences as sized-vectors. However, for circuit-generators, a more lightweight solution is to use *dynamic* typing. Since the circuit-generator runs at "compile-time", before the circuit is actually implemented, a dynamic type-error is in many cases just as useful as a static one. Indeed, Recipe is easily extended to produce *size mismatch exceptions* for invalid assignments. Unfortunately, these messages only indicate the presence of an error and not its whereabouts. This is an instance of a more general problem faced by Haskell programmers (locating the origin of exceptions), and a solution would be of great value to Recipe. One promising-looking solution that has recently been proposed is *contract checking* [40].

3.8.5 Flattening

Lava descriptions can be flattened to netlists of primitive components and used, for example, to configure an FPGA. Unfortunately, after flattening, the call-graph structure of the original description and all the identifiers used in it are lost – they are simply evaluated away by the Haskell implementation. This is problematic for two reasons. Firstly, netlists become large as every call to a particular function in the description is expanded to a separate list of gates. Secondly, and more worryingly, synthesis reports typically refer to names *in the netlist*, and there is no way to trace these back to names *in the original description*. For example, the Xilinx synthesis tool gives the critical path as a sequence of netlist labels.

These problems appear to be an inevitable consequence of using meta-programming for circuit description. Yet it is meta-programming that gives the approach its power in the first place. If call-traces were available during circuit generation, the name problem could possibly be solved. It is not clear how to preserve the structure of the description without resorting to the use of an explicit – and less convenient – component abstraction instead of plain Haskell functions.

3.8.6 Advanced synthesis

Standard HDLs like VHDL provide a wealth of pre-defined components. In contrast, Lava does not even provide a subtractor in its arithmetic module! VHDL synthesis tools are also quite advanced. For example, the efficient composition of block RAMs to form a larger memory can be inferred simply from the width and capacity of a VHDL array. Similarly, VHDL tools look to extract efficient state machines, multiplexors, decoders, and shift registers from behavioural descriptions. In Lava as it stands, all of these structures have to be captured explicitly. A Lava module providing a range of commonly-used circuits would be a great help here.

The downside of advanced synthesis of VHDL descriptions is that it is a black-box. It is difficult for the programmer to know the hardware generated for a given description. On this matter, Wakerly writes:

> *For the foreseeable future, digital designers who use synthesis*

> *tools will need to pay reasonably close attention to their coding style in order to obtain good results. And for the moment, the definition of "good coding style" depends somewhat on both the synthesis tool and the target technology.* [96]

In contrast, Lava and Recipe descriptions have clearly-defined and easy-to-understand mappings to hardware.

3.9 Related work

3.9.1 Claessen and Pace

In [21], Claessen and Pace introduce two Lava libraries for behavioural description: one for matching boolean signals using regular expressions and another capturing a small subset of Esterel [12]. Claessen and Pace advocate Lava as a *framework* for defining behavioural languages. They show how to implement a behavioural language by introducing a Haskell data type capturing the abstract syntax of the language and an evaluation function assigning a circuit-level meaning to the syntax. In particular, they emphasise the possibility of introducing multiple behavioural languages, and the ability to connect them together so that different languages can be used for different parts of the description.

This chapter supplements Claessen and Pace's work in two main ways. Firstly, from a technical point of view, the language presented here supports mutable variables. Of the all features provided, this is perhaps the trickiest to support. Secondly, from a broader perspective, the approach has been applied to a number of examples, in particular the Reduceron. Although for reasons of space the full Reduceron description has not been presented here, the stack processor Poly and its associated compiler and abstract machine are highly illustrative of it.

3.9.2 Handel-C

Handel-C is a hardware compiler developed by Celoxica [16], and is also inspired by Page and Luk's compilation scheme from Occam to FPGAs [77]. It

```
macro proc mult(a, b, acc) {
  unsigned (width(a)) regA;
  unsigned (width(b)) regB;

  par {
    regA = a;
    regB = b;
    acc  = 0;
  }

  while (regB != 0) {
    par {
      regA = regA << 1;
      regB = regB >> 1;
      if (regB[0]) acc = acc + regA;
    }
  }
}
```

Figure 3.20: Sequential multiplier in Handel-C.

shares several features with Recipe but it is implemented as an autonomous behavioural language rather than a library for a structural language. Figure 3.20 shows the sequential multiplier example written in Handel-C. Handel-C's assignment operator `=` is like `<=|` in Recipe, as it consumes a clock-cycle. As a result, `par` blocks must be used to make assignments happen in parallel. Another minor difference is that Handel-C variable names can be used directly in an expression context, such as in the `while` condition and right-hand-side of `=`, whereas in Recipe, the `val` function must be used to obtain the value of a variable.

Handel-C provides many features that Recipe does not. Channels, for example, support synchronous communication between parallel blocks, and `prialt` allows blocking on multiple channels. Arrays are also supported, and can be implemented using flip-flops, lookup-tables or block RAMs. Each function in the program is mapped onto a single piece of hardware, and multiple function calls represent shared access to the hardware. Other features include a `switch` statement, pointers, and interfacing to foreign components and off-chip memories.

Handel-C provides a higher-order untyped functional language, in the form of *macro expressions*, for structural description. However there are two limitations. First, the macro language is call-by-name (not call-by-need) so named intermediate subexpressions are not shared – not even those bound by a `let` expression. Handel-C does allow a *shared expression* to be declared, but this serves a different purpose: to share a parameterised expression, *not* the result of evaluating an expression. The second limitation is that expressions in Handel-C always execute within a clock-cycle, so only combinatorial logic can be expressed. The end result of these two limitations is that only tree-shaped (not graph-shaped) circuits can be expressed using macro expressions.

3.9.3 SPARK

A different approach to behavioural description is taken by SPARK [34]. SPARK is a *high-level synthesis* [65, 30] system capable of compiling a subset of ANSI C to hardware. The restrictions forbid the use of pointers, recursion and `goto` statements, and force all function calls to be inlined.

Being a high-level synthesis system means that the SPARK programmer does not specify the consumption of clock-cycles by statements, like in Recipe and Handel-C. Instead, a timing model for the program is determined automatically. Each ANSI-C operator is considered to be a resource, and the programmer specifies how many of each resource are available. SPARK then determines which operations should execute in which clock-cycle, attempting to perform as much work as possible in each clock-cycle without breaking the resource constraints. This process is known as *resource-constrained scheduling*. The performance of SPARK relies heavily on being able to determine a large number of operations in the program that can be executed concurrently in hardware. For this, a variety of parallelising *code motion* transformations are proposed.

SPARK is a prototype tool. Commercially-sold high-level synthesis systems are scarcely available, if at all. This situation is perhaps understandable. In general, an efficient hardware algorithm is often very different to an efficient software algorithm, and it is perhaps unreasonable to expect a computer to be able to derive the one from the other [33]. However, it would be

interesting to see if high-level synthesis techniques could be successfully applied to the Reduceron to allow an efficient machine to be produced from a higher-level description.

3.10 Summary

This chapter has explored an alternative approach to hardware description in which a behavioural language is expressed simply as a library in a pure structural language. In this case, the structural language is itself a library for the functional language Haskell. This library-based approach is in contrast to the conventional approach of implementing an autonomous language with built-in support for both styles of description. The main advantage of the library-based approach is that it is extremely simple to implement.

Despite the disadvantages discussed in Sections 3.8 and 3.9, the Recipe library is already useful. Three example applications were captured by short, clear descriptions. In particular, implementing the stack processor Poly reflected the approach taken to implement the Reduceron, a substantial application. The source code for the Reduceron is available online[3].

[3] `http://www.cs.york.ac.uk/fp/reduceron/reduceron-thesis.tar.gz`

Chapter 4

Target-Directed Evaluation

The previous chapter was concerned with circuit descriptions written in a functional language. This chapter is concerned with trying to find errors in such descriptions and indeed errors in functional programs in general.

4.1 Introduction

A desirable outcome of software testing is that every reachable expression in a program contributes to at least one correct execution of that program. With this goal, Gill and Runciman propose the combination of QuickCheck [20] and Haskell Program Coverage (HPC) [31] to test Haskell programs. QuickCheck applies program properties to randomly chosen inputs, and HPC marks-up the source code, highlighting any unevaluated expressions in a suitably prominent colour. Repeated runs of QuickCheck and HPC typically decrease the number of coloured expressions until only a handful of stubborn ones remain. This chapter presents *Reach*, a program analyser that aims to determine how to cause evaluation of these hard-to-reach expressions. More precisely, Reach solves the following problem.

> **GIVEN** a program with a top-level function marked as a *source* and an expression marked as a *target*,
>
> **FIND** *applications* of the source function that entail evaluation of the target.

The Reach analyser is *conservative*: a target-reaching application may exist even if no solution is returned. The degree of effort with which the analyser attempts to find a solution is controlled by a bound on the domain of the source function, or a bound on the depth of function calls. The analyser operates on programs written in a first-order subset of Haskell. Extending it to work on higher-order programs is discussed in Section 4.10.1.

4.1.1 A motivating example

Suppose that the deletion operation of a binary search tree data structure is to be implemented and tested. A binary search tree is either empty or is a node containing an element and two subtrees.

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

It has the invariant that for every node `Node a l r`, the elements in `l` are no greater than `a`, and the elements in `r` are no less than `a`. This invariant (`ord`) is defined in Figure 4.1, along with the deletion function (`del`). The deletion function operates as follows.

- If the given tree is empty then `del` returns an empty tree.
- If the element to be deleted is less than or greater than the value at the root of the tree, then it is recursively deleted from the tree's left or right subtree respectively.
- If the element to be deleted is equal to the value at the root, then the left subtree is returned, with its rightmost leaf replaced with the right subtree. This replacement operation is defined by the `ext` function.

If the implementation of `del` is correct, it should preserve the `ord` invariant. That is, for all trees t and values a, if t is ordered then so is the tree that results from deleting a from t. This property is captured by the function `prop_ordDel` defined in Figure 4.1, where the arguments of `prop_ordDel` represent universally quantified variables. The `==>` operator denotes implication.

```
22  del a Empty = Empty
23  del a (Node b t0 t1)
24    | a < b = Node b (del a t0) t1
25    | a > b = Node b t0 (del a t1)
26    | a == b = ext t0 t1
27
28  ext Empty t = t
29  ext (Node a t0 t1) t = Node a t0 (ext t1 t)
30
31  allLe x Empty = True
32  allLe x (Node a t0 t1) = a <= x && allLe x t0 && allLe x t1
33
34  allGe x Empty = True
35  allGe x (Node a t0 t1) = a >= x && allGe x t0 && allGe x t1
36
37  ord Empty = True
38  ord (Node a t0 t1) = allLe a t0 && allGe a t1
39
40  prop_ordDel :: (Nat, Tree Nat) -> Property
41  prop_ordDel (a, t) = ord t ==> ord (del a t)
42
43  main = quickCheck prop_ordDel
```

Figure 4.1: Partial output of HPC on the motivating example.

4.1.2 Testing with QuickCheck and HPC

A popular method of testing program properties is to use the QuickCheck [20] library, which applies properties to randomly generated inputs. Compiling and running the program in Figure 4.1 often yields:

```
OK, passed 100 tests.
```

All one hundred of these test cases contain an ordered tree – QuickCheck does *not* count randomly-generated inputs that falsify the property's antecedent as passed tests.

If a program is compiled and run with Haskell Program Coverage (HPC) [31] enabled then it produces a `.hpc` file which can be used to generate a marked-up version of the program's source code in which unevaluated expressions are highlighted in yellow. It is common to find the second equation of the `ext` function unevaluated, as highlighted on line 29 in Figure 4.1, even after a dozen batches of a hundred random tests. HPC also highlights boolean expressions that always evaluate to true in green. The highlighted expression on line 26 always evaluates to true because for integer values `a` and `b`, if `a`

is not less than `b` and `a` is not greater than `b`, then `a` must equal `b`.

4.1.3 Program coverage with Reach

A suitable place, then, to insert a target is in the second equation of `ext`.

```
ext Empty t = t
ext (Node a t0 t1) t = target (Node a t0 (ext t1 t))
```

The target expression is marked using the primitive unary function `target` recognised by Reach. Applying Reach to the modified program, with the source function set as `prop_ordDel`, yields several target-reaching applications. Interestingly, the 90^{th} such application evaluates to `False`.

```
prop_ordDel (1, Node 1 (Node 1 Empty (Node 0 Empty Empty))
                       (Node 1 Empty Empty))
```

The property does not hold because the programmer forgot to make `ord` recursively call itself on each subtree! It is not surprising that one of the applications returned by Reach refutes the property, because to do so, one must at least delete an element from a tree containing that element, and this requires that `ext` is called.

4.1.4 Direct refutation with Reach

Since a target can be placed anywhere, it is possible to set Reach the goal of refuting properties directly. For example, with the useful auxiliary

```
refute True  = True
refute False = target False
```

refutation of `prop_ordDel` can be attempted as follows.

```
prop_ordDel (a, t) = refute (ord t ==> ord (del a t))
```

Passing the problem to Reach yields a series of refutations. The first is:

```
prop_ordDel (0, (Node 0 Empty (Node 1 Empty
                  (Node 0 Empty Empty))))
```

4.1.5 Chapter outline

Section 4.2 defines the syntax of the core language that Reach operates on, and a basic lazy evaluator for the language. Sections 4.3, 4.4 and 4.5 make a series of incremental modifications to the basic lazy evaluator resulting in three variants of the Reach analyser: *Basic*, *Forward* and *Backward*. Section 4.6 discusses implementation details and Section 4.7 applies each Reach variant to a range of example programs, and compares the results. Section 4.8 lists the conclusions of the comparison. Section 4.9 discusses related work and Section 4.10 states a number of limitations of Reach and suggests how these limitations might be overcome.

4.2 Syntax and semantics

The Reach analyser operates on a *core*, *first-order* functional language. Extension of Reach to a higher-order language is discussed in Section 4.10.1. Programs written in a first-order subset of Haskell can be transformed to programs in the core language using the York Haskell Compiler [90]. This section defines a syntax and a basic lazy evaluator (call-by-need semantics) for the core language.

4.2.1 Syntax

The syntax is defined in Figure 4.2. The meta-variable v ranges over variable names, f over function names, c over data constructor names, d over function definitions and p over programs. Lists of meta-variables are denoted with an overhead arrow, for example $\vec{e}$ denotes a list of expressions.

Patterns in case alternatives are of the form $c\ v_1 \cdots v_n$ where n is the arity of the constructor c. Case expressions are *exhaustive*, that is, an alternative is present for every constructor that the case subject can evaluate to.

To help reduce clutter in the definitions presented in this chapter, the core syntax omits let expressions. *Acyclic* let bindings in Haskell are desugared to function applications in the core language by the rewrite rule

$$\texttt{let}\ [v_1 \texttt{ = } e_1,\ \cdots,\ v_n \texttt{ = } e_n]\ \texttt{in}\ e \quad \longrightarrow \quad f\ \vec{w}\ e_1\ \cdots\ e_n$$

p	::=	$\vec{d}$	(program)
d	::=	$f\ \vec{v}$ `=` e	(function definition)
e	::=	v	(variable)
	|	$f\ \ \vec{e}$	(function application)
	|	$c\ \ \vec{e}$	(data construction)
	|	`case` e `of` $\vec{a}$	(case expression)
	|	$\bullet$	(target)
a	::=	$c\ \vec{v}\ \mapsto\ e$	(case alternative)

Figure 4.2: Syntax of the core functional language.

where $f\ \vec{w}\ v_1\ \cdots\ v_n$ `=` e and $\vec{w}$ is the list of free variables in e. *Cyclic* let bindings are not supported, but are not believed to cause any difficulty.

4.2.2 Semantics

A lazy evaluator for the core language is defined as a big-step binary relation, $\rightarrow$, between initial and final *configurations*. A configuration is a pair of the form $\langle s, e\rangle$ containing a state s and an expression e. The value of a variable v in state s is denoted $s(v)$, assuming that v is bound, that is, $v \in \mathit{domain}(s)$. The state $s[v := e]$ is just like s, except with v bound to e. Similarly, $s[\vec{v} := \vec{e}]$ is just like s, except with each variable in $\vec{v}$ bound to the expression at the corresponding position in $\vec{e}$.

The semantics is similar to Launchbury's *A natural semantics for lazy evaluation* [57] but for a first-order language with explicit data constructors and case expressions.

The $\rightarrow$ relation evaluates a given expression to *head normal form*, that is, a data constructor applied to unevaluated arguments. The following reduction rule expresses that data constructions are already in head normal form.

$$\langle s, c\ \vec{e}\rangle\ \rightarrow\ \langle s, c\ \vec{e}\rangle \qquad \text{(Con)}$$

To evaluate a variable, the expression bound to that variable, $s(v)$, is evaluated. Crucial to lazy evaluation is that once $s(v)$ is evaluated, v is rebound to the result so that the computation is never repeated.

$$\frac{\langle s, s(v)\rangle \;\rightarrow\; \langle s', e\rangle}{\langle s, v\rangle \;\rightarrow\; \langle s'[v := e], e\rangle} \text{ if } v \in \mathit{domain}(s) \qquad (\text{Var}_0)$$

To evaluate an application, a function $fresh(s, f)$ is used to obtain a new function definition, just like f's except that all variables in it are replaced with fresh, unbound variables with respect to the state s.

$$\frac{\langle s[\vec{v} := \vec{e}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, f\ \vec{e}\rangle \;\rightarrow\; \langle s', e'\rangle} \text{ if } (f\ \vec{v}\ \texttt{=}\ e) = fresh(s, f) \qquad (\text{App})$$

To evaluate a case expression, the subject is evaluated, and passed to the matching relation, $\rightarrow_M$, which takes a triple of the form $\langle s, e, \vec{a}\rangle$ where e is the evaluated case subject and $\vec{a}$ is the list of case alternatives.

$$\frac{\langle s, e\rangle \;\rightarrow\; \langle s', e'\rangle, \quad \langle s', e', \vec{a}\rangle \;\rightarrow_M \langle s'', e''\rangle}{\langle s, \texttt{case}\ e\ \texttt{of}\ \vec{a}\rangle \;\rightarrow\; \langle s'', e''\rangle} \qquad (\text{Case})$$

If the case subject is a constructor, then the appropriate case alternative is selected and evaluated.

$$\frac{\langle s[\vec{v} := \vec{e}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, c\ \vec{e}, \vec{a}\rangle \;\rightarrow_M\; \langle s', e'\rangle} \text{ if } (c\ \vec{v} \mapsto e) \in \vec{a} \qquad (\text{Match})$$

This completes the semantics of the core functional language.

4.3 Basic Reach

This section defines Basic Reach, a simple version of Reach that applies the source function exhaustively to each individual value in a bounded domain, and detects if the target is evaluated.

4.3.1 Extending the semantics

Basic Reach extends the semantics of Section 4.2.2 with two new rules. First, a target expression is already in head normal form.

$$\langle s, \bullet\rangle \;\rightarrow\; \langle s, \bullet\rangle \qquad (\text{Targ}_0)$$

Second, if a case expression contains a case subject that evaluates to the target, then the whole case expression also evaluates to the target.

$$\langle s, \bullet, \vec{a} \rangle \ \rightarrow_M \ \langle s, \bullet \rangle \qquad (\text{Targ}_1)$$

4.3.2 Full normal form

The $\rightarrow$ relation reduces an expression to *head normal form*, that is, a target or a constructor with unevaluated arguments. Now it is necessary to recursively apply $\rightarrow$ to unevaluated constructor arguments, as these expressions could also lead to evaluation of the target. A new big-step relation, $\twoheadrightarrow$, is defined between configuration pairs of the form $\langle s, e \rangle$ and *answers*. An answer is either YES s', indicating that evaluation of e from state s reaches the target and results in state s', or NO s' if it does not. The first rule defining $\twoheadrightarrow$ is as follows.

$$\langle s, \bullet \rangle \ \twoheadrightarrow \ \text{YES } s \qquad (\text{Full}_0)$$

A data construction reaches the target if any of its arguments does.

$$\frac{\langle s, \vec{e} \rangle \ \twoheadrightarrow^\star \ ans}{\langle s, c \ \vec{e} \rangle \ \twoheadrightarrow \ ans} \qquad (\text{Full}_1)$$

The meta-variable *ans* represents answers. The auxiliary relation $\twoheadrightarrow^\star$ determines whether any of a list of expressions reaches the target. It is defined as follows, assuming that [] denotes the empty list, and $x : \vec{x}$ denotes the list with an initial element x and a tail $\vec{x}$.

$$\langle s, [] \rangle \ \twoheadrightarrow^\star \ \text{NO } s \qquad (\text{Any}_0)$$

$$\langle s, e : \vec{e} \rangle \ \twoheadrightarrow^\star \ \text{YES } s' \ \text{ if } \langle s, e \rangle \twoheadrightarrow \text{YES } s' \qquad (\text{Any}_1)$$

$$\frac{\langle s', \vec{e} \rangle \ \twoheadrightarrow^\star \ ans}{\langle s, e : \vec{e} \rangle \ \twoheadrightarrow^\star \ ans} \ \text{ if } \langle s, e \rangle \twoheadrightarrow \text{NO } s' \qquad (\text{Any}_2)$$

Any other expression reaches the target if its head normal form does.

$$\frac{\langle s, e \rangle \ \rightarrow \ \langle s', e' \rangle, \ \ \langle s', e' \rangle \ \twoheadrightarrow \ ans}{\langle s, e \rangle \ \twoheadrightarrow \ ans} \ \text{ if } \neg normal(e) \qquad (\text{Full}_2)$$

The predicate $normal(e)$ holds if e is in head normal form.

4.3.3 Enumerating the domain

The source function is applied to all inputs in a bounded domain to determine if the target is reachable. The domain is bounded by limiting the *depth* of data constructions that occur in it, where depth is defined as

$$depth(c\ \vec{e}) \quad = \quad \begin{cases} 0 & \text{if } \ \vec{e} = [] \\ 1 + maximum([depth(e) \mid e \ \in \ \vec{e}]) & \text{otherwise} \end{cases}$$

The domain, say of type t, is enumerated with the help of a function $disjuncts(t)$ that returns the constructors of t, each paired with a list containing the types of the constructor's arguments. To enumerate the domain, the type t must of course be monomorphic. To illustrate, if `Natural` is defined as

```
data Natural = Z | S Natural
```

then $disjuncts($`Natural`$)$ returns $[\langle$`Z`$, []\rangle, \langle$`S`$, [$`Natural`$]\rangle]$. Now a relation $\mathbf{enum}_d$ is defined such that $t\ \mathbf{enum}_d\ e$ maps the type t to each value e of type t up to depth d.

$$t\ \mathbf{enum}_d\ c\ [] \ \text{ if } \ d = 0 \ \wedge \ \langle c, [] \rangle \ \in \ disjuncts(t) \tag{Val$_0$}$$

$$\frac{\vec{t}\ \mathbf{enum}^{\star}_{d-1}\ \vec{e}}{t\ \mathbf{enum}_d\ c\ \vec{e}} \ \text{ if } d > 0 \ \wedge \ \langle c, \vec{t} \rangle \ \in \ disjuncts(t) \tag{Val$_1$}$$

Notice that $\mathbf{enum}_d$ is non-deterministic, mapping a single type to possibly many values, because a type may have many disjuncts. The auxiliary relation $\mathbf{enum}^{\star}_d$ maps lists of types to lists of values.

$$[]\ \mathbf{enum}^{\star}_d\ [] \tag{Val$_2$}$$

$$\frac{t\ \mathbf{enum}_d\ e, \ \vec{t}\ \mathbf{enum}^{\star}_d\ \vec{e}}{(t : \vec{t})\ \mathbf{enum}^{\star}_d\ (e : \vec{e})} \tag{Val$_3$}$$

4.3.4 Bounding recursion

In many cases, bounding the construction depth of values in the domain of the source function is enough to make Basic Reach terminate. But this is

not always the case: a program may loop without demanding any external input at all. For this reason, Basic Reach also allows the depth of function calls to be bounded.

To bound the depth of function calls, every function symbol f is tagged with a natural number b. When a function symbol f_b is applied, every function symbol in the body of f is tagged with $b-1$, unless $b=0$ in which case the application fails. More formally, the App rule is redefined as

$$\frac{\langle s[\vec{v} := \vec{e}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, f_b\ \vec{e}\rangle \;\rightarrow\; \langle s', e'\rangle} \text{ if } (f\ \vec{v} = e) = fresh_b(s, f) \qquad \text{(App)}$$

where $fresh_n(s, f)$ is like $fresh(s, f)$ except that every function symbol in the body of f is tagged with $n-1$, unless $n=0$ in which case the function body returned is $\bot$. There are no reduction rules that deal with $\bot$; evaluation of $\bot$ fails.

4.3.5 Definition of Basic Reach

Basic Reach is defined as

$$\frac{types(f)\ \mathbf{enum}^{\star}_{d}\ \vec{e},\ \langle [], f_b\ \vec{e}\rangle \;\twoheadrightarrow\; \textsc{Yes}\ s'}{f\ \mathbf{reach}\ \vec{e}} \qquad \text{(Basic Reach)}$$

where f is the source function, $types(f)$ yields the types of the arguments of f, d is the construction depth bound, and b is the call-depth bound.

4.4 Forward Reach

Forward Reach extends Basic Reach by allowing the source function to be applied to *unbound variables*. The intuition for an unbound variable is that it represents *any possible value*. If, during evaluation, the value of an unbound variable is required for evaluation to proceed, then that variable is non-deterministically *bound* to a data constructor of the right type applied to fresh unbound variables. In other words, unbound variables are instantiated during evaluation, only when required, and only by the amount needed for evaluation to proceed. Compared to Basic Reach, there are two main differences.

1. The bounded domain of the source function need not necessarily be enumerated in full. If the value of an unbound variable is not demanded, due to lazy evaluation, then the portion of the domain represented by that variable is never enumerated.

2. Some evaluation is shared between different inputs with common substructures. This is not the case in Basic Reach, where the source function is applied to each possible input in turn, and evaluated from scratch each time.

These two differences will be illustrated by example in Section 4.4.2. The idea to evaluate functions on unbound variables is inspired by *needed narrowing* [2], an evaluation strategy used by some functional-logic languages, most notably Curry [35].

4.4.1 Extensions to Basic Reach

The possibility of encountering an unbound variable during evaluation gives rise to a new normal form.

$$\langle s, v\rangle \;\rightarrow\; \langle s, v\rangle \;\text{ if } v \notin \mathit{domain}(s) \qquad (\mathrm{Var}_1)$$

The interesting case is when a *case subject* evaluates to an unbound variable. In this case, a case alternative is picked non-deterministically, and the unbound variable is bound to that alternative's pattern.

$$\frac{\langle s[v := c\ \vec{v}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, v, \vec{a}\rangle \;\rightarrow_M\; \langle s', e'\rangle} \;\text{ if } (c\ \vec{v} \mapsto e) \in \vec{a} \qquad (\mathrm{Narrow})$$

This rule is non-deterministic because c is not constrained to be a particular constructor, and thus matches any case alternative.

Considering now full normal form, if an unbound variable occurs in the result of the source function, then evaluation of that variable cannot lead to evaluation of the target.

$$\langle s, v\rangle \;\twoheadrightarrow\; \textsc{No}\ s \;\text{ if } v \notin \mathit{domain}(s) \qquad (\mathrm{Full}_3)$$

```
a0 && b0 = case a0 of [False ↦ False, True ↦ b0]
and xs0 = case xs0 of [[] ↦ True, x0:xs1 ↦ x0 && and xs1]
example ys = case and ys of [False ↦ False, True ↦ •]
```

Figure 4.3: An example program in abstract syntax.

4.4.2 An example derivation

Figure 4.3 contains an example program, in core syntax. Part of the derivation tree of the modified rules applied to `example v` is given below, where `v` is an unbound fresh variable. The first step in the derivation is

```
[]                                                          1
example v                                                   (App)
```

Here the elements of the initial configuration pair $\langle [], \texttt{example v} \rangle$ have been written on the left of separate lines. On the right is a step number and the name of an applicable rule. The derivation proceeds as follows.

```
[ys := v]                                                   2
case and ys of [False ↦ False, True ↦ •]                    (Case)
```

Reduction of the case expression first requires reduction of the case subject. This sub-derivation is indented.

```
  [ys := v]                                                 2.1
  and ys                                                    (App)

  [ys := v,  xs0 := ys]                                     2.2
  case xs0 of [[] ↦ True, x0:xs1 ↦ x0 && and xs1]           (Case)

    [ys := v,  xs0 := ys]                                   2.2.1
    xs0                                                     (Var0)

    [ys := v,  xs0 := ys]                                   2.2.2
    ys                                                      (Var0)

    [ys := v,  xs0 := v]                                    2.2.3
    v                                                       (Var1)
```

The subject of the case expression first considered in step 2.2 is now in normal form, and is passed to the matching relation $\rightarrow_M$, which recall takes a triple containing the state, the evaluated case subject, and the case alternatives.

$[\texttt{ys} := \texttt{v},\ \texttt{xs}_0 := \texttt{v}]$ 2.3
v
$[\texttt{[]} \mapsto \texttt{True},\ \texttt{x}_0\texttt{:xs}_1 \mapsto \texttt{x}_0\ \texttt{\&\& and xs}_1]$ (Narrow)

Since the Narrow rule is non-deterministic, the derivation forks at this point. First, the derivation branch that binds `v` to `[]` is considered.

$[\texttt{ys} := \texttt{v},\ \texttt{xs}_0 := \texttt{v},\ \texttt{v} := \texttt{[]}]$ 2.4
True (Con)

The subject of the case expression, `and ys`, first considered in step 2 is now in normal form.

$[\texttt{ys} := \texttt{v},\ \texttt{xs}_0 := \texttt{v},\ \texttt{v} := \texttt{[]}]$ 3
True
$[\texttt{False} \mapsto \texttt{False},\ \texttt{True} \mapsto \bullet]$ (Match)

$[\texttt{ys} := \texttt{v},\ \texttt{xs}_0 := \texttt{v},\ \texttt{v} := \texttt{[]}]$ 4
$\bullet$ (Targ_0)

A normal form is reached, which also happens to be the target expression, so it can be inferred that `example []` reaches the target. Returning to the fork point in step 2.3, the derivation continues, this time binding `v` to $\texttt{x}_0 : \texttt{xs}_1$.

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1]$ 2.4
$\texttt{x}_0$ `&& and` $\texttt{xs}_1$ (App)

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1,\ \texttt{a}_0 := \texttt{x}_0,\ \texttt{b}_0 := \texttt{and xs}_1]$ 2.5
`case` $\texttt{a}_0$ `of` $[\texttt{False} \mapsto \texttt{False},\ \texttt{True} \mapsto \texttt{b}_0]$ (Case)

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1,\ \texttt{a}_0 := \texttt{x}_0,\ \texttt{b}_0 := \texttt{and xs}_1]$ 2.5.1
$\texttt{a}_0$ (Var_0)

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1,\ \texttt{a}_0 := \texttt{x}_0,\ \texttt{b}_0 := \texttt{and xs}_1]$ 2.5.2
$\texttt{x}_0$ (Var_1)

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1,\ \texttt{a}_0 := \texttt{x}_0,\ \texttt{b}_0 := \texttt{and xs}_1]$ 2.6
$\texttt{x}_0$
$[\texttt{False} \mapsto \texttt{False},\ \texttt{True} \mapsto \texttt{b}_0]$ (Narrow)

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1,\ \texttt{a}_0 := \texttt{x}_0,\ \texttt{b}_0 := \texttt{and xs}_1,\ \texttt{x}_0 := \texttt{False}]$ 2.7
False (Con)

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1,\ \texttt{a}_0 := \texttt{x}_0,\ \texttt{b}_0 := \texttt{and xs}_1,\ \texttt{x}_0 := \texttt{False}]$ 3
False
$[\texttt{False} \mapsto \texttt{False},\ \texttt{True} \mapsto \bullet]$ (Match)

$[\texttt{ys} := \texttt{v},\ \texttt{v} := \texttt{x}_0\texttt{:xs}_1,\ \texttt{a}_0 := \texttt{x}_0,\ \texttt{b}_0 := \texttt{and xs}_1,\ \texttt{x}_0 := \texttt{False}]$ 4
`False` (Con)

Now a normal form has been reached which is not the target. It can be inferred that `example` ($\texttt{False:xs}_1$) does not reach the target for any value of $\texttt{xs}_1$. The derivation continues at the fork point in step 2.6, this time binding the first element of the input list, $\texttt{x}_0$, to `True`, but details will not be elaborated here.

The two main differences of Forward Reach compared to Basic Reach stated in Section 4.4 can now be illustrated. First, all the inputs represented by $\texttt{False:xs}_1$, for any value $\texttt{xs}_1$, are not enumerated by Forward Reach, because all such inputs are known not to reach the target. In contrast, Basic Reach enumerates all input lists, regardless of whether or not they begin with `False`. Second, all reductions up to the fork point in step 2.3 are shared between the two forked sub-derivations elaborated above. In contrast, Basic Reach repeats these reductions when reducing, for example, `example []` and `example [False]`.

4.4.3 Enumerating the domain

Due to the presence of unbound variables, it is no longer necessary to explicitly enumerate the domain of the source function, but it is still desirable to *bound* it. This is achieved by extending the evaluation state to associate each unbound variable with its *depth*.

An expression of the form $s_\downarrow(v)$ denotes a natural number representing the depth of a variable v in state s, and $s[\vec{v} :=_\downarrow n]$ denotes the state s with the depth of each variable in $\vec{v}$ set to n. When instantiating an unbound variable v of depth n to a construction $c\ \vec{v}$, the variables in $\vec{v}$ are assigned depth $n-1$, provided $n > 0$ or $\vec{v}$ is empty. More precisely, the Narrow rule is redefined as follows.

$$\frac{\langle s[v := c\ \vec{v}][\vec{v} :=_\downarrow s_\downarrow(v) - 1], e\rangle\ \rightarrow\ \langle s', e'\rangle}{\langle s, v, \vec{a}\rangle\ \rightarrow_M\ \langle s', e'\rangle} \quad \text{(Narrow)}$$
$$\text{if } (c\ \vec{v} \mapsto e) \in \vec{a}\ \wedge\ (s_\downarrow(v) > 0\ \vee\ \vec{v} = [])$$

4.4.4 Definition of Forward Reach

Forward Reach is defined as

$$\frac{\langle [\vec{v} :=_{\downarrow} d], f_b\ \vec{v} \rangle \ \twoheadrightarrow\ \textsc{Yes}\ s',\ \vec{v}\ \mathbf{deref}^{\star}_{s'}\ \vec{e}}{f\ \mathbf{reach}\ \vec{e}} \qquad \text{(Forward Reach)}$$

where f is the source function, $\vec{v}$ is a list of fresh, unbound variables, d is the construction depth bound and b is the call-depth bound. The auxiliary $\mathbf{deref}^{\star}_{s}$ relation determines the values of a list of variables in state s,

$$[]\ \mathbf{deref}^{\star}_{s}\ [] \qquad (\text{Deref}^{\star}_{0})$$

$$\frac{v\ \mathbf{deref}_s\ e,\ \vec{v}\ \mathbf{deref}^{\star}_{s}\ \vec{e}}{v : \vec{v}\ \mathbf{deref}^{\star}_{s}\ e : \vec{e}} \qquad (\text{Deref}^{\star}_{1})$$

and $\mathbf{deref}_s$ finds the value of a single variable in state s.

$$\frac{s(v)\ \mathbf{deref}_s\ e}{v\ \mathbf{deref}_s\ e}\ \text{ if } v\ \in \mathit{domain}(s) \qquad (\text{Deref}_0)$$

$$v\ \mathbf{deref}_s\ v\ \text{ if } v\ \notin \mathit{domain}(s) \qquad (\text{Deref}_1)$$

$$\frac{\vec{v}\ \mathbf{deref}^{\star}_{s}\ \vec{e}}{c\ \vec{v}\ \mathbf{deref}_s\ c\ \vec{e}} \qquad (\text{Deref}_2)$$

Forward Reach may produce applications of the source function which contain unbound variables. Any values of appropriate type can be substituted for such variables. If totally-instantiated inputs are desired, Deref_1 can be redefined as

$$\frac{\mathit{type}(v)\ \mathbf{enum}_{s_{\downarrow}(v)}\ e}{v\ \mathbf{deref}_s\ e}\ \text{ if } v \notin \mathit{domain}(s) \qquad (\text{Deref}_1)$$

where $\mathit{type}(v)$ represents the type of the variable v.

4.5 Backward Reach

Basic Reach and Forward Reach both begin at the source function and apply reduction rules that aim to expose the target. It is also reasonable to proceed in the opposite direction, starting at the function containing the target and applying reduction rules that aim to expose the source. This is

what Backward Reach does. Compared to Forward Reach, there are two main differences.

1. Expressions close to the target are evaluated first. If a restrictive constraint surrounds the target then evaluating it first will narrow the search space early in the analysis.

2. Backward Reach does not continue searching for a target when there is no reference to the target or to a target-reaching function symbol in the expression graph under evaluation. This is not the case in Forward Reach, which will reduce the expression to full normal form even if there are no such references to the target.

These two differences will be illustrated in Section 4.5.6.

4.5.1 Overview

Backward Reach repeatedly applies a set of six rewrite rules to an expression containing the target (Section 4.5.3). The initial expression is obtained by inlining function-calls in the body of the source function until the target is revealed (Section 4.5.7). Each rewrite rule matches the target in a particular expression context and lifts it one step closer to the source. Notably, one of the rewrite rules is only applicable if an equational constraint of the form $e_0 = e_1$ can be solved where e_0 is contains only constructors and variables, and e_1 is an arbitrary expression. Such constraints are solved by a combination of unification and lazy symbolic evaluation of e_1 (Section 4.5.4).

The following invariant is maintained throughout: *the expression under analysis contains exactly one target.*

4.5.2 The target context

Motivated by the six rewrite rules to be presented in Section 4.5.3, a new syntactic construct for expressions, $\{e\}$, called a *target context* is introduced. A *valid target context* is a target context $\{e\}$ where e is either a target or a non-zero arity data construction. If it is a non-zero arity data construction then exactly one of the arguments to the constructor must itself be a valid

target context. For example, the following are valid target contexts provided that e_1 and e_3 are not target contexts.

$$\{\bullet\}$$
$$\{c\ \{\bullet\}\}$$
$$\{c\ e_1\ \{\bullet\}\ e_3\}$$
$$\{c_0\ e_1\ \{c_1\ \{\bullet\}\}\ e_3\}$$

Note that any valid target context contains exactly one target.

4.5.3 The six rewrite rules

The rewrite rules are defined by a small-step transition relation, $\Rightarrow$, between state-expression pairs. The purpose of the first rewrite rule is to wrap the target in a target context.

$$\langle s, \bullet \rangle \ \Rightarrow\ \langle s, \{\bullet\} \rangle \qquad (\text{Init}^{\uparrow})$$

Rules are not applicable *inside* target contexts, so the above rule, for example, is not infinitely applicable. At most one rewrite rule is applicable to any expression, assuming it contains exactly one target context.

The aim of each of the five remaining rewrite rules is to lift the target context outside the syntactic construct in which it occurs. When the target context appears at the outermost position in an expression, the analysis is complete. There is one rewrite rule for each possible position in which the target context may occur.

Data constructions To lift the target context outside a data construction, the data construction is simply wrapped in a new target context.

$$\langle s, c\ e_1 \cdots \{e_i\} \cdots e_n \rangle \ \Rightarrow\ \langle s, \{c\ e_1 \cdots \{e_i\} \cdots e_n\} \rangle \qquad (\text{Con}^{\uparrow})$$

Function applications Consider an expression containing a function application of the form

$$f\ e_1 \cdots \{e_i\} \cdots e_n$$

The aim of Backward Reach here is to apply f to its arguments without losing the target and without replicating it, to maintain the invariant that the expression being rewritten contains exactly one target. There are three cases to consider.

1. If f is defined by $f\ \vec{v} = e$ and v_i does not occur in e then the rewrite rule fails because the target is no longer referenced.

2. Alternatively, if v_i occurs exactly once in e, then $\{e_i\}$ is substituted for v_i in e to give a new expression e'. The variables in $\vec{v}$ are bound to the arguments in $\vec{e}$ in the usual way. The target is not replicated because there are no references to v_i in e' – it does not matter what v_i is bound to.

3. Alternatively, if v_i occurs more than once in e then $\{e_i\}$ is *non-deterministically substituted* for some individual reference to v_i to give a new expression e'. Non-deterministic substitution is defined in Figure 4.4. To illustrate, if the body of f is $g\ v_i\ (h\ v_i)$ where g and h are function symbols then

   $$(g\ v_i\ (h\ v_i))\ [\![\{e_i\}/v_i]\!]$$

   reduces to

   $$g\ \{e_i\}\ (h\ v_i)$$

   and also, non-deterministically, to

   $$g\ v_i\ (h\ \{e_i\})$$

   The variables $\vec{v}$ are bound to the arguments $\vec{e}$ in the usual way. However, now the target is replicated because there are still references to v_i in e'. This situation is resolved by binding v_i to $bot(e_i)$ where *bot* is defined in Figure 4.5. In particular, *bot* takes an expression and removes all target contexts and replaces the target with $\perp$. To illustrate, *bot* applied to the target context

   $$\{c_0\ e_1\ \{c_1\ \{\bullet\}\}\ e_3\}$$

reduces to

$$c_0\ e_1\ (c_1\ \bot)\ e_3$$

To summarise: if a function body contains $n > 1$ occurrences of an argument v which is to be bound to the target context, then the analyser splits non-deterministically into to n sub-analysers. Each sub-analyser substitutes the target context for a different occurrence of v. All remaining occurrences of v are replaced by a version of the target context in which all target sub-contexts are removed and the target is replaced with $\bot$. A sub-analyser fails if it demands the value of $\bot$, but such a failure implies that another sub-analyser, in which this instance of $\bot$ is replaced with $\bullet$, proceeds.

The rewrite rule capturing these three cases is defined below.

$$\begin{array}{ll} \langle s, f_b\ e_1 \cdots \{e_i\} \cdots e_n \rangle & \\ \Downarrow & \text{if } (f\ \vec{v}\ \texttt{=}\ e) = fresh'_b(f)\ \ (\text{App}^{\uparrow}) \\ \langle s[\vec{v} := \vec{e}][v_i := bot(e_i)], e[\![\{e_i\}/v_i]\!] \rangle & \end{array}$$

The function $fresh'_b(s, f)$ is like $fresh_b(s, f)$ except that the body of the returned function definition has any targets replaced with $\bot$. This maintains the invariant that exactly one target is present in the expression under analysis.

Case expressions If a target context containing the target appears in the subject of a case expression, then it can be lifted out directly.

$$\langle s, \texttt{case}\ \{\bullet\}\ \texttt{of}\ \vec{a} \rangle\ \Rightarrow\ \langle s, \{\bullet\} \rangle \qquad (\text{Case}_0^{\uparrow})$$

If a target context containing a data constructor applied to a list of arguments appears in the subject of a case expression, then the argument holding the sub-context is non-deterministically substituted into the matching case alternative, much in the same way as the App$^{\uparrow}$ rule.

$$\begin{array}{ll} \langle s, \texttt{case}\ \{c\ e_1 \cdots \{e_i\} \cdots e_n\}\ \texttt{of}\ \vec{a} \rangle & \\ \Downarrow & \text{if } (c\ \vec{v} \mapsto e) \in \vec{a} \qquad (\text{Case}_1^{\uparrow}) \\ \langle s[\vec{v} := \vec{e}][v_i := bot(e_i)], e[\![\{e_i\}/v_i]\!] \rangle & \end{array}$$

$$
\begin{array}{rcl}
v\ [\![e/v]\!] & = & e \\
f\ \vec{e}\ [\![e/v]\!] & = & f\ (\vec{e}\ [\![e/v]\!]^\star) \\
c\ \vec{e}\ [\![e/v]\!] & = & c\ (\vec{e}\ [\![e/v]\!]^\star) \\
\texttt{case}\ e_0\ \texttt{of}\ \vec{a}\ [\![e/v]\!] & = & \texttt{case}\ e_0\ [\![e/v]\!]\ \texttt{of}\ \vec{a} \\
\texttt{case}\ e_0\ \texttt{of}\ \vec{a}\ [\![e/v]\!] & = & \texttt{case}\ e_0\ \texttt{of}\ \vec{a}\ [\![e/v]\!]^A \\
 & & \\
e_0 : \vec{e}\ [\![e/v]\!]^\star & = & e_0\ [\![e/v]\!] : \vec{e} \\
e_0 : \vec{e}\ [\![e/v]\!]^\star & = & e_0 : (\vec{e}\ [\![e/v]\!]^\star) \\
 & & \\
c\ \vec{v} \mapsto e_0 : \vec{a}\ [\![e/v]\!]^A & = & c\ \vec{v} \mapsto e_0\ [\![e/v]\!] : \vec{a} \\
c\ \vec{v} \mapsto e_0 : \vec{a}\ [\![e/v]\!]^A & = & c\ \vec{v} \mapsto e_0 : (\vec{a}\ [\![e/v]\!]^A)
\end{array}
$$

Figure 4.4: Non-deterministic substitution.

$$
bot(e) \quad = \quad \begin{cases}
\bot & \text{if } e = \bullet \\
bot(e_0) & \text{if } e = \{e_0\} \\
c\ [\ bot(v)\ |\ v \in \vec{v}\ ] & \text{if } e = c\ \vec{v} \\
e & \text{otherwise}
\end{cases}
$$

Figure 4.5: Definition of *bot*.

If the target context occurs as the right-hand-side in a case alternative a, then the target can be lifted outside the case expression provided that the case subject can be *unified* with the pattern of a.

$$\begin{array}{l} \langle s, \texttt{case}\ e\ \texttt{of}\ \vec{a}\rangle \\ \qquad \Downarrow \qquad\qquad \text{if } (c\ \vec{v} \mapsto \{e_t\}) \in \vec{a}\ \wedge\ \langle s, c\ \vec{v}, e\rangle\ \textbf{unify}\ s' \qquad (\text{Case}_2^{\uparrow}) \\ \langle s', \{e_t\}\rangle \end{array}$$

Unification is defined in the next section.

4.5.4 Unification

The aim of the **unify** relation is to bind values to variables in order to make two expressions equal. It relates a triple of the form $\langle s, e_0, e_1\rangle$ to s', where e_0 and e_1 are the expressions to unify, s is the initial mapping of variables to values and s' is a mapping in which e_0 and e_1 are equal. Whereas e_1 is an arbitrary expression representing a case subject, two assumptions are made about e_0.

1. It is assumed that e_0 is a *partial construction*. A partial construction is either an unbound variable, or a data constructor applied to several expressions each of which is itself a partial construction, or a variable which is bound to a partial construction.

2. It is assumed that no unbound variables in the partial construction e_0 occur in e_1, eliminating the need for an occurs check [81] to avoid construction of cyclic terms.

These two assumptions always hold when applying the $\text{Case}_2^{\uparrow}$ rule, and are discharged in Section 4.5.5.

To unify two expressions, it is first convenient to find their *roots*. The root of an expression e in state s, denoted $root_s(e)$, is e if e is not a bound variable and $root_s(s(e))$ if it is.

$$\begin{array}{lcl} root_s(v) & = & \begin{cases} root_s(s(v)) & \text{if } v \in domain(s) \\ v & \text{otherwise} \end{cases} \\ root_s(e) & = & e \ \text{ if } \neg var(e) \end{array}$$

The **unify** relation is defined in terms of the helper relation **uni**, which may assume that the roots of the expressions to unify are given.

$$\frac{\langle s, root_s(e_0), root_s(e_1)\rangle \ \mathbf{uni}\ s'}{\langle s, e_0, e_1\rangle \ \mathbf{unify}\ s'} \tag{Unify}$$

The $\mathbf{unify}^\star$ relation is used to unify lists of expressions.

$$\langle s, [], []\rangle \ \mathbf{unify}^\star\ s \tag{Unify}_0^\star$$

$$\frac{\langle s, e_0, e_1\rangle \ \mathbf{unify}\ s',\ \langle s', \vec{e}_0, \vec{e}_1\rangle \ \mathbf{unify}^\star\ s''}{\langle s, e_0 : \vec{e}_0, e_1 : \vec{e}_1\rangle \ \mathbf{unify}^\star\ s''} \tag{Unify}_1^\star$$

An unbound variable can be unified with an arbitrary expression by a straightforward binding.

$$\langle s, v, e\rangle \ \mathbf{uni}\ s[v := e] \tag{Uni}_0$$

$$\langle s, c\ \vec{e}, v\rangle \ \mathbf{uni}\ s[v := c\ \vec{e}] \tag{Uni}_1$$

Two data constructions with the same constructor can be unified if each of their arguments can.

$$\frac{\langle s, \vec{e}_0, \vec{e}_1\rangle \ \mathbf{unify}^\star\ s'}{\langle s, c\ \vec{e}_0, c\ \vec{e}_1\rangle \ \mathbf{uni}\ s'} \tag{Uni}_2$$

The only remaining case to consider is when the first expression to unify is a data construction and the second is an arbitrary expression. In this case, the second expression is evaluated to head normal form, and unification is attempted again.

$$\frac{\langle s, e_1\rangle \rightarrow \langle s', e_1'\rangle,\ \langle s', e_0, e_1'\rangle \ \mathbf{uni}\ s''}{\langle s, e_0, e_1\rangle \ \mathbf{uni}\ s''} \ \text{if } \neg var(e_0) \wedge \neg normal(e_1) \tag{Uni}_3$$

Evaluation is performed by the Forward Reach's $\rightarrow$ relation.

4.5.5 Unification assumptions

This section discharges the two assumptions made in the previous section about the inputs to the **unify** relation. Recall that the $\text{Case}_2^\uparrow$ rule can only be applied to an expression of the form

$$\texttt{case}\ e\ \texttt{of}\ [\ \cdots,\ c\ \vec{v}\ \mapsto\ \{e_t\},\ \cdots\ ]$$

```
a0 && b0 = case a0 of [False ↦ False, True ↦ b0]
c0 || d0 = case c0 of [False ↦ d0, True ↦ True]
and ws0 = case ws0 of [[] ↦ True, w0:ws1 ↦ w0 && and ws1]
or xs0 = case xs0 of [[] ↦ False, x0:xs1 ↦ x0 || or xs1]
example2 ys = case or ys of
  [False ↦ e0, True ↦ case and ys of [False ↦ e1, True ↦ •]]
```

Figure 4.6: Another example program in abstract syntax.

if $\langle s, c\ \vec{v}, e\rangle$ **unify** s' succeeds.

It was assumed that $c\ \vec{v}$ is a partial construction. By definition, $c\ \vec{v}$ is a partial construction if all the variables in $\vec{v}$ are also partial constructions. Before Backward Reach begins, the variables in $\vec{v}$ are unbound. The only way for Backward Reach to bind variables in $\vec{v}$ to values before application of $\text{Case}_2^{\uparrow}$ is by application of the Uni_3 rule, and consequently, the Narrow rule. The Narrow rule always binds variables to constructors which are themselves applied to fresh unbound variables. Therefore the variables in $\vec{v}$ must be partial constructions.

It was also assumed that the variables in $\vec{v}$ do not occur in the case subject e. Indeed, this is the case because the variables in $\vec{v}$ are not in scope in the case subject e.

4.5.6 An example derivation

Figure 4.6 contains an example program, in abstract syntax. Part of the derivation of Backward Reach applied to `example2 ys zs` is given below, where `ys` and `zs` are unbound variables. The initial state-expression configuration pair is

```
[]                                                        1
case or ys of [False ↦ e0, True ↦
  case and ys of [False ↦ e1, True ↦ •]]          (Init↑)
```

As before, the elements of configuration tuples are written on the left of separate lines. On the right is a step number and the name of an applicable rule. The derivation proceeds as follows.

[] 2
`case or ys of` [`False` $\mapsto$ e_0, `True` $\mapsto$
  `case and ys of` [`False` $\mapsto$ e_1, `True` $\mapsto$ {$\bullet$}]] ($\text{Case}_2^{\uparrow}$)

To apply $\text{Case}_2^{\uparrow}$, `True` must be unified with `and ys`. The sub-derivation representing this unification is indented. Recall that the **unify** relation takes a configuration triple.

  [] 2.1
  `True`
  `and ys` (Unify, Uni_3)

To apply Uni_3, `and ys` must be symbolically evaluated to head normal form. Some of this derivation is elided as the evaluation rules have already been illustrated in Section 4.4.2.

    [] 2.1.1
    `and ys` (App, $\cdots$)

    [`ys` := `[]`] 2.1.2
    `True` (Con)

Symbolically evaluating `and ys` for an unbound `ys` leads to a number of results; the first is `True` with `ys` bound to `[]`. Unification proceeds:

  [`ys` := `[]`] 2.2
  `True`
  `True` (Uni_2)

[`ys` := `[]`] 3
`case or ys of` [`False` $\mapsto$ e_0, `True` $\mapsto$ {$\bullet$}] ($\text{Case}_2^{\uparrow}$)

  [`ys` := `[]`] 3.1
  `True`
  `or ys` (Unify, Uni_3)

    [`ys` := `[]`] 3.1.1
    `or ys` (App, $\cdots$)

    [`ys` := `[]`] 3.1.2
    `False` (Con)

  [`ys` := `[]`] 3.2
  `True`
  `False` –

Here, there is no applicable rule in the **unify** relation since `True` and `False` are not equal. The analyser backtracks to the evaluation of `and ys` in step 2.1.1, this time resulting in `False` and binding `ys` to `False:ys1` where `ys1` is unbound.

[ys := False:ys1]	2.1.2
False	(Con)
[ys := False:ys1]	2.2
True	
False	–

Again unification fails and the analyser backtracks to step 2.1.1.

[ys := True:[]]	2.1.2
True	(Con)
[ys := True:[]]	2.2
True	
True	(Uni_2)
[ys := True:[]]	3
case or ys of [False $\mapsto e_0$, True $\mapsto$ {•}]	($\text{Case}_2^{\uparrow}$)
[ys := True:[]]	3.1
True	
or ys	(Unify, Uni_3)
[ys := True:[]]	3.1.1
or ys	(App, $\cdots$)
[ys := True:[]]	3.1.2
True	(Con)
[ys := True:[]]	3.2
True	
True	(Uni_2)
[ys := True:[]]	3.3
{•}	

The analyser succeeds when the target context reaches the outermost position in the expression being rewritten. Backward Reach infers that `example2` applied to `[True]` causes evaluation of the target, and backtracks to step 2.1.1 to find further solutions.

The two main differences between Forward and Backward Reach stated in Section 4.5 can now be illustrated. First, the orders in which constraints are solved by the two are not necessarily the same: Backward Reach solves `and ys` before `or ys` whereas Forward Reach would solve `or ys` before `and ys`; the latter is of course worse because it will result in many solutions to `or ys` that do not satisfy `and ys`. The second difference is that some unnecessary evaluation is avoided by Backward Reach. In particular, it never evaluates e_0 or e_1 (see Figure 4.6). Forward Reach evaluates e_0 for all `ys` that falsify `or ys`, and likewise e_1 for all `ys` that satisfy `or ys` but falsify `and ys`; yet evaluation of e_0 and e_1 is unnecessary to reach the target.

4.5.7 Initial inlining

The six rewrite rules of Backward Reach operate on expressions, not programs. To apply the rewrite rules to a program, the program is converted to an initial expression containing exactly one target. This is achieved by inlining the source function until the target is exposed, as defined by the **inline** relation in Figure 4.7. In the definition of **inline**, the expression $holds_T(f)$ holds if the body of f contains the target and $on_T(f)$ holds if the function f is on a call-path to a target-containing function. Since many paths to the target may exist, there are many possible initial expressions. This is captured by non-determinism in the **inline** relation. The inline relation terminates for all finite expressions due to the call-depth bound.

Any expression produced by the **inline** relation contains exactly one target, assuming that the original program contains exactly one target. The only rules that can reveal a target are those which apply a function, namely Inline_0 and Inline_1, and as the former produces a normal form and the latter removes any targets, only an expression containing at most one target can be produced. Furthermore, since the former rule produces the *only* normal form, only an expression containing at least one target can be produced.

4.5.8 Definition of Backward Reach

Backward Reach is defined as

$$\langle s, f_b\ \vec{e}\rangle\ \textbf{inline}\ \langle s[\vec{v} := \vec{e}], e\rangle \text{ if } (f\ \vec{v}\ \texttt{=}\ e) = fresh_b(f) \wedge holds_T(f) \quad (\text{Inline}_0)$$

$$\frac{\langle s[\vec{v} := \vec{e}], e\rangle\ \textbf{inline}\ \langle s', e'\rangle}{\langle s, f_b\ \vec{e}\rangle\ \textbf{inline}\ \langle s', e'\rangle} \text{ if } (f\ \vec{v}\ \texttt{=}\ e) = fresh'_b(f) \wedge on_T(f) \quad (\text{Inline}_1)$$

$$\frac{\langle s, \vec{e}\rangle\ \textbf{inline}^\star\ \langle s', \vec{e}'\rangle}{\langle s, f\ \vec{e}\rangle\ \textbf{inline}\ \langle s', f\ \vec{e}'\rangle} \quad (\text{Inline}_2)$$

$$\frac{\langle s, \vec{e}\rangle\ \textbf{inline}^\star\ \langle s', \vec{e}'\rangle}{\langle s, c\ \vec{e}\rangle\ \textbf{inline}\ \langle s', c\ \vec{e}'\rangle} \quad (\text{Inline}_3)$$

$$\frac{\langle s, e\rangle\ \textbf{inline}\ \langle s', e'\rangle}{\langle s, \texttt{case}\ e\ \texttt{of}\ \vec{a}\rangle\ \textbf{inline}\ \langle s', \texttt{case}\ e'\ \texttt{of}\ \vec{a}\rangle} \quad (\text{Inline}_4)$$

$$\frac{\langle s, \vec{a}\rangle\ \textbf{inline}^A\ \langle s', \vec{a}'\rangle}{\langle s, \texttt{case}\ e\ \texttt{of}\ \vec{a}\rangle\ \textbf{inline}\ \langle s', \texttt{case}\ e\ \texttt{of}\ \vec{a}'\rangle} \quad (\text{Inline}_5)$$

$$\frac{\langle s, \vec{e}\rangle\ \textbf{inline}^\star\ \langle s', \vec{e}'\rangle}{\langle s, e : \vec{e}\rangle\ \textbf{inline}^\star\ \langle s', e : \vec{e}'\rangle} \quad (\text{Inline}^\star_0)$$

$$\frac{\langle s, e\rangle\ \textbf{inline}\ \langle s', e'\rangle}{\langle s, e : \vec{e}\rangle\ \textbf{inline}^\star\ \langle s', e' : \vec{e}\rangle} \quad (\text{Inline}^\star_1)$$

$$\frac{\langle s, \vec{a}\rangle\ \textbf{inline}^A\ \langle s', \vec{a}'\rangle}{\langle s, a : \vec{a}\rangle\ \textbf{inline}^A\ \langle s', a : \vec{a}'\rangle} \quad (\text{Inline}^A_0)$$

$$\frac{\langle s, e\rangle\ \textbf{inline}\ \langle s', e'\rangle}{\langle s, c\ \vec{v} \mapsto e : \vec{a}\rangle\ \textbf{inline}^A\ \langle s', c\ \vec{v} \mapsto e' : \vec{a}\rangle} \quad (\text{Inline}^A_1)$$

Figure 4.7: Inlining to reveal a target.

$$\frac{f_b\ \vec{v}\ \textbf{inline}\ e,\ \langle s, e\rangle\ \Rightarrow^\star\ \langle s', \{e'\}\rangle,\ \vec{v}\ \textbf{deref}^\star_{s'}\ \vec{e}}{f\ \textbf{reach}\ \vec{e}} \quad \text{(Backward Reach)}$$

where f is the source function, $\vec{v}$ is a list of fresh, unbound variables, and b is the call-depth bound. Backward Reach does not support a construction depth bound. The $\Rightarrow^\star$ relation holds between two state-expression pairs if the first can be rewritten using one or more $\Rightarrow$ rewrites to the second.

4.6 Implementation

The definitions presented in previous sections are essentially logic programs. They have been implemented directly in Haskell, using lazy lists to express non-determinism [93]. The implementations take as input core programs produced by the York Haskell Compiler [32]. Features of Haskell rejected by the implementations include higher-order functions and primitive data types – only algebraic data types are supported. A library for natural numbers is available and can be imported by any program. This library encodes natural numbers in unary using an algebraic data type, and provides several common operations over such numbers.

4.7 Comparison

Tables 4.1 and 4.2 present performance figures for each Reach variant applied to a range of benchmark properties. In particular, Table 4.1 compares Basic Reach and Forward Reach using a construction-depth bound, and Table 4.2 compares Forward Reach and Backward Reach using a call-depth bound (recall that Backward Reach does not support a construction-depth bound). Each Reach variant returns exactly the same set of solutions when applied to each problem.

Some of the benchmark problems referred to in Table 4.1 are direct property-refutation problems. Forward Reach and Backward Reach perform identically on such problems, so the results are not repeated in Table 4.2.

Graphical comparisons are shown in Figures 4.8 and 4.9 respectively. Each graph compares the times taken by two Reach variants to solve each problem

Problem		**Construction depth**					
		3	**4**	**5**	**6**	**7**	**8**
Tree_1	Bs	0.02	⋆0.18	⋆819.1	×		
	Fo	0.02	⋆0.07	⋆15.6	×		
Tree_2	Bs	⋆0.03	⋆0.21	⋆466.7	×		
	Fo	⋆0.02	⋆0.11	⋆10.3	×		
RedBlack_1	Bs	0.09	10.90	×			
	Fo	0.07	0.71	114.4	$^{+}$215.1		
RedBlack_2	Bs	0.09	⋆9.88	×			
	Fo	0.06	⋆0.70	⋆112.4	×		
Mux_1	Bs	0.03	0.09	1.9	718.0	×	
	Fo	0.03	0.04	0.1	0.1	0.2	0.5
Mux_2	Bs	0.03	0.08	1.6	490.8	×	
	Fo	0.03	0.04	0.1	0.1	0.2	0.5
Huffman_1	Bs	0.04	0.11	0.7	6.0	66.9	1137.5
	Fo	0.05	0.13	0.8	6.8	73.1	1077.6
Huffman_2	Bs	0.04	0.10	4.97	×		
	Fo	0.04	0.07	0.52	11.9	1538.9	×
Turner_1	Bs	0.04	0.98	×			
	Fo	0.03	0.06	1.43	×		
Turner_2	Bs	0.05	8.31	×			
	Fo	0.03	0.05	⋆4.78	×		

Key: Bs Basic Reach; Fo Forward Reach; ⋆ Solution(s) found; $^{+}$ Time to find 1^{st} solution; × Longer than 40 minutes

Table 4.1: Time taken (seconds) to find all solutions to a range of benchmark problems using Basic and Forward Reach.

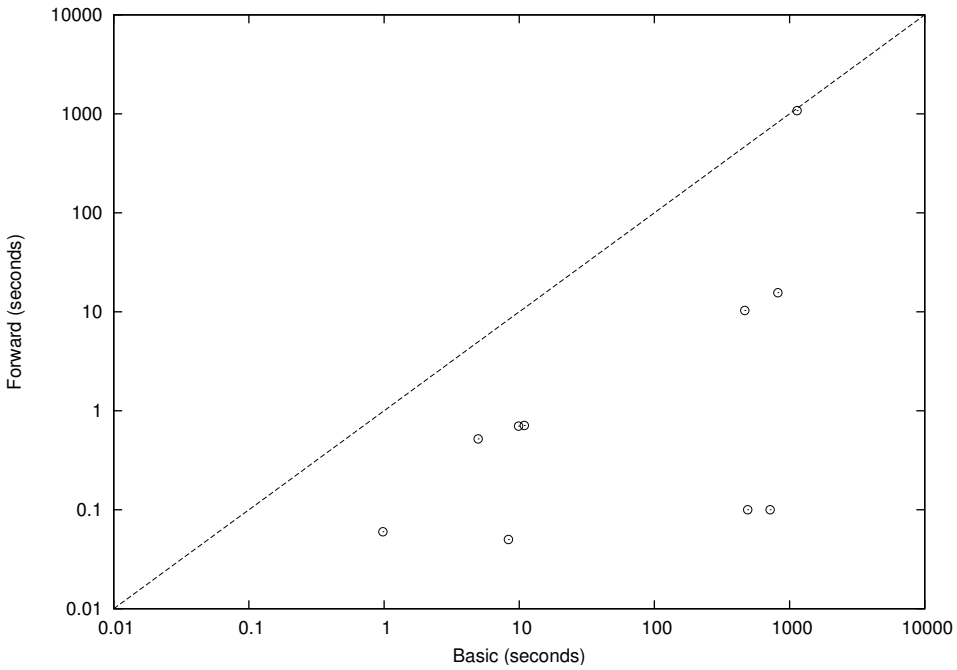

Figure 4.8: Graphical comparison of Basic and Forward Reach.

Problem		**Function call-depth**					
		7	**8**	**9**	**10**	**11**	**12**
$Tree_2$	Fo	⋆0.08	⋆16.29	×			
	Bk	⋆0.05	⋆4.53	×			
$RedBlack_2$	Fo	0.03	0.09	⋆23.18	×		
	Bk	0.04	0.07	⋆20.33	×		
Mux_2	Fo	0.03	0.04	0.06	0.10	0.20	0.4
	Bk	0.07	0.24	1.24	4.90	57.93	290.9
$Turner_2$	Fo	0.03	0.03	0.03	0.07	0.49	⋆34.98
	Bk	0.03	0.03	0.03	0.03	0.04	⋆1.22

Key: Fo Forward Reach ⋆ Solution(s) found
Bk Backward Reach × Longer than 40 minutes

Table 4.2: Time taken (seconds) to find all solutions to a range of benchmark problems using Forward and Backward Reach.

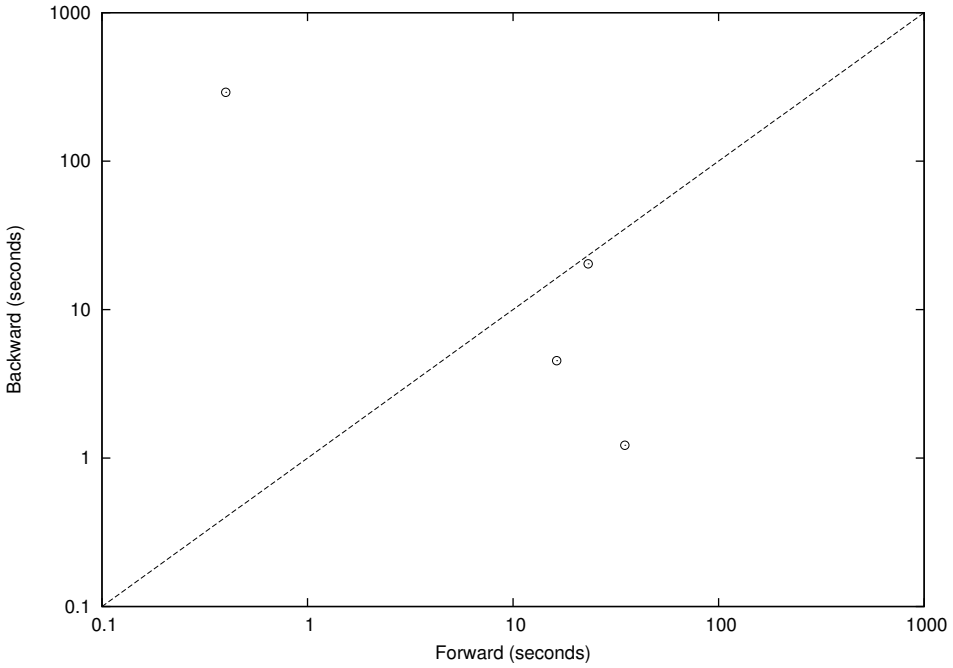

Figure 4.9: Graphical comparison of Forward and Backward Reach.

at *one* specific depth bound. In particular, the depth bound associated with each plotted point is the greatest depth bound at which both the Reach variants terminate within 40 minutes.

The benchmark properties are chosen because testing them using QuickCheck with a *standard generator* sometimes leaves some expressions unevaluated after a standard batch of (at most) a hundred tests. A standard generator is one of the form described in the section "Generating Recursive Data Types" of the QuickCheck manual [43].

This section takes the form of 23 explicitly-marked observations about the performance differences between the three Reach variants. It aims not just to highlight the differences, but also to account for why these differences arise. The style of presentation is intended to convey the exploratory nature of the comparison. This exploration paves the way for some more-general conclusions to be made in the next section. Unless otherwise stated, the observations refer to results in Tables 4.1 and 4.2.

4.7.1 Binary search trees

This section considers the application of Reach to part of the binary search tree library given in Section 4.1.1. Two benchmark problems – Tree_1 and Tree_2 – have `prop_ordDel` marked as the source. Tree_1 targets refutation of `prop_ordDel`, and Tree_2 targets evaluation of the right-hand-side of the second equation of `ext` – an expression sometimes left uncovered by QuickCheck. Recall that that `prop_ordDel` is defined as:

```
prop_ordDel :: (Nat, Tree Nat) -> Bool
prop_ordDel (e, t) = ord t ==> ord (del e t)
```

Observation 1 On the Tree_1 and Tree_2 problems, Forward Reach is an order of magnitude faster than Basic Reach in finding all solutions with a construction depth bound of five. □

The explanation is that Forward Reach can determine the result of a function when applied to partially-instantiated inputs, as demonstrated in Section 4.4.2. For example, Forward Reach can determine that the partial tree

$$\texttt{Node 0 (Node 1 } t_0\ t_1\texttt{)}\ t_2$$

falsifies `ord` for any values of t_0, t_1 and t_2. Consequently, it can determine that `prop_ordDel` applied to the input

$$\texttt{(}e\texttt{, Node 0 (Node 1 } t_0\ t_1\texttt{)}\ t_2\texttt{)}$$

does not lead to a target for any t_0, t_1, t_2 and e. By not demanding the variables to be further instantiated, Forward Reach avoids generation of large portions of the input space. In particular, many trees falsifying `ord` are never generated.

Observation 2 Of the 238,145 trees of natural numbers with construction depth four or less, only 8,544 satisfy `ord`. This is consistent with the order of magnitude performance benefit of Forward Reach over Basic Reach when construction depth is bounded to five. (Note that because the arguments to `prop_ordDel` are tupled, checking the property with a depth bound of five corresponds to generating trees of depth four or less.) □

Constraint	Time
`ord` t `&& member` x t	8.63
`member` x t `&& ord` t	4.57

Table 4.3: How the order of conjuncts affects the time taken (seconds) to find ordered trees (using the faulty definition of `ord`) with an unconstrained member using symbolic evaluation bounded with a call depth of eight.

Observation 3 On the Tree_2 problem, Backward Reach is approximately four times faster than Forward Reach in finding all solutions with a call-depth bound of eight. □

To reach the target in the Tree_2 problem, both Forward and Backward Reach must:

1. find an ordered tree t,
2. find an element x such that deletion of x from t causes evaluation of the target.

Step 2 amounts to finding a node in t containing x and a non-empty left subtree, as captured by the following predicate.

```
member x Empty = False
member x (Node y t0 t1)
  | x < y  = member x t0
  | x > y  = member x t1
  | x == y = case t0 of Node _ _ _ -> True ; _ -> False
```

The difference between Forward and Backward Reach on the Tree_2 problem can be characterised as follows: the former solves the constraint

`ord` t `&& member` x t

for unconstrained x and t by symbolic evaluation, whereas the latter solves

`member` x t `&& ord` t

for unconstrained x and t. That is, the difference is whether `ord` is solved before `member` (Forward Reach), or vice-versa, `member` before `ord` (Backward Reach).

Observation 4 Table 4.3 shows the times taken to solve each constraint ordering. The timings suggest that solving `member` first is better, which is consistent with the timings in Table 4.1 showing that Backward Reach is more effective than Forward Reach on the Tree_2 problem. □

One explanation for the difference in performance between the two constraint orderings is as follows. First recall that the `ord` predicate is faulty: although it holds for all ordered trees, it also holds for some unordered ones. In solving `ord` first, trees are first constructed which satisfy an overly-general ordering constraint. In solving `member` first, trees are first constructed which contain the element to be deleted, and in which the elements on the path from the root to that element are *correctly ordered*. Therefore, the latter introduces a correct and more restrictive ordering on part of a candidate tree earlier during evaluation. An example of an input that satisfies `ord` but which falsifies `member` is

```
(1, Node 2 (Node 0 (Node 1 Empty Empty)
                   (Node 0 Empty Empty)) Empty)
```

Observation 5 The timings in Tables 4.2 and 4.3 show that the constraint `member` x t `&& ord` t requires the same amount of time to solve as Backward Reach does to find all solutions to the Tree_2 problem. That is, the constraint appears to be an accurate characterisation of Backward Reach. However, the timings also show that the constraint `ord` t `&& member` x t is solved in half the time that Forward Reach takes to find all solutions to the Tree_2 problem. In this case, the constraint is not such an accurate characterisation of Forward Reach. □

There is an explanation why Forward Reach is slower than expected. Consider an x and a t such that `ord` t holds but `member` x t does not. Forward Reach applied to the Tree_2 problem must fully evaluate the right-hand-side of the implication in `prop_ordDel`, that is, determine the orderedness of the tree that results from deleting an element from a tree which does not contain that element. Such evaluation cannot lead to the target.

```
insert x s = makeBlack (ins x s)
  where makeBlack (T _ a y b) = T B a y b
        ins x E = T R E x E
        ins x (T col a y b)
          | x < y = bal col (ins x a) y b
          | x > y = bal col a y (ins x b)
          | otherwise = T col a y b

bal B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
bal B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
bal B a x (T R (T R c y b) z d) = T R (T B a x b) y (T B c z d)
bal B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
bal col a x b = T col a x b
```

Figure 4.10: A faulty insertion function for Red-Black trees.

4.7.2 Red-Black trees

This section considers a more sophisticated tree implementation – balanced red-black trees – by Okasaki [76]. Okasaki's tree representation is as follows.

```
data Colour = R | B
data Tree a = E | T Colour (T a) a (T a)
```

An ordered red-black tree is one satisfying the following three conditions: that trees are ordered (the `ordered` invariant), that no red node has a red parent (the `red` invariant), and that every path from the root to an empty node contains the same number of black nodes (the `black` invariant).

```
redBlack t = ordered t && black t && red t
```

The following property asserts that a faulty version of Okasaki's insertion function, defined in Figure 4.10, preserves the `redBlack` invariant.

```
prop_insertRB :: (Nat, Tree Nat) -> Bool
prop_insertRB (x, t) = redBlack t ==> redBlack (insert x t)
```

The fault is located in the left-hand-side of the third equation of the rebalancing function `bal`; the variables `b` and `c` are the wrong way around. Both the benchmark problems – RedBlack_1 and RedBlack_2 – mark the property `prop_insertRB` as the source. RedBlack_1 targets refutation of the property, and RedBlack_2 targets evaluation of the right-hand-side of the first equation of `bal` – an expression sometimes left uncovered by QuickCheck.

Observation 6 When using QuickCheck to try to refute `prop_insertRB`, no counter-example was found after 100,000 batches of 1000 random tests (amounting to over 30 minutes of testing). □

Observation 7 Using Basic Reach, a solution to RedBlack_1 is not found within the forty-minute cut-off period. However, using Forward Reach, a solution is found within four minutes. □

Again the advantage of Forward Reach is its ability to efficiently find trees satisfying the restrictive antecedent. In this example, trees must not only be ordered but must also be red-black. This requirement offers increased pruning opportunities compared to the plain ordering constraint. To illustrate, the partially-instantiated tree

`T B (T B (T B E 0 E) 1 E) 2 (T` x `E 3 E)`

satisfies `ordered` but falsifies `black` for all x, so the conjunction of `ordered` and `black` allows more input pruning than `ordered` alone. However, any tree satisfying `black` must have a fully instantiated structure with all nodes coloured, so there is no scope for further pruning from `red`. If `black` and `red` are swapped, then `red` has the opportunity to prune the input-space instead of `black`, and it might do so more effectively.

Observation 8 Of the 539,889,801 trees of natural numbers with construction depth four or less, only 3,675 satisfy `redBlack`. This large difference is consistent with the difficulty in finding a counter-example at random using QuickCheck and with the large performance difference between Basic and Forward Reach. □

Observation 9 If the `red` and `black` conjuncts are swapped in the definition of `redBlack`, the time taken to find all solutions to RedBlack_1 with a depth bound of five increases from 114.4 seconds to 724.8 seconds. In the worst case, if the order

```
redBlack t = red t && ord t && black t
```

is chosen, the time taken is longer than 40 minutes (2,400 seconds). □

	Constraint	Time
1.	`path1` x t `&& redBlack` t	1.42
2.	`path2` x t `&& redBlack` t	19.72
3.	`redBlack` t `&& path1` x t	23.38
4.	`redBlack` t `&& path2` x t	24.27
5.	`redBlack` t	23.04

Table 4.4: Times taken (seconds) to solve the constraints characterising Forward and Backward Reach on the RedBlack_2 problem with a call-depth bound of nine.

Observation 10 Forward and Backward Reach perform very similarly on the RedBlack_2 problem. □

Again, it is useful to characterise the behaviour of Forward and Backward Reach in terms of the constraints that must be solved to reach the target. With a call-depth bound of nine, there are only two paths from the source function `prop_insertRB` to the target: one via the first guarded alternative to the `ins` function, and the other via the second. In the former case, an x and a t must be found such that `redBlack` t and `path1` x t both hold.

```
path1 x E = False
path1 x (T col a y b) =
  case ins x a of
    T R (T R t0 x0 t1) x1 t2 -> x < y && isBlack col
    _ -> False
```

In the latter case, an x and a t must be found such that `redBlack` t and `path2` x t both hold.

```
path2 x E = False
path2 x (T col a y b) =
  case a of
    T R (T R t0 x0 t1) x1 t2 -> x > y && isBlack col
    _ -> False
```

Backward Reach is characterised by constraints 1 and 2 in Table 4.4, and Forward Reach by constraints 3 and 4.

Observation 11 The sum of the times taken to solve constraints 1 and 2 in Table 4.4 is consistent with time taken by Backward Reach to solve the RedBlack_2 problem. This is not surprising since Backward Reach analyses

the two paths to the target *separately*. In contrast, Forward Reach *shares* much work needed to analyse each path to the target. In particular, the solving of the `redBlack` predicate (constraint 5) is shared, and it accounts for almost all of the time taken to solve constraints 3 and 4. Indeed, the time taken to solve constraint 5 is very similar to the time taken by Forward Reach on the RedBlack_2 problem. □

4.7.3 A digital multiplexor

This section considers the circuit description of a multiplexor given in Section 3.4.1. To make this description suitable for Reach, it is modified to be first-order and to work over `Bool` rather than Lava's `Bit` type.

```
pick :: [(Bool, [Bool])] -> [Bool]
```

To illustrate, the expression

```
pick [(False, [False, True]), (True, [True, False])]
```

evaluates to `[True, False]`. The `pick` function returns the list paired with `True`; it is assumed that exactly one such list exists. To be correct, `pick` should return the same result as the following specification.

```
pickSpec ((b, xs):ps) = if b then xs else pickSpec ps
```

More specifically, the following property should hold.

```
prop_pick ps = oneHot ps && rect ps ==> pick ps == pickSpec ps
```

The `oneHot` precondition specifies that exactly one element in `ps` is a pair whose first component is `True`. The `rect` precondition ensures that all the second components of the pairs in `ps` have the same length.

Both the benchmark problems – Mux_1 and Mux_2 – mark `prop_pick` as the source function. Mux_1 targets refutation of `prop_pick`. In Mux_2, the target is an expression often missed by QuickCheck, marked as follows.

```
[]     == ys = null ys
(x:xs) == ys = case ys of
                 [] -> target False
                 y:ys -> (x <==> y) && (xs == ys)
```

Constraint	Time
1. `(oneHot` x `&& rect` x`) && prefix (pick` x`) (pickSpec` x`)`	0.39
2. `prefix (pick` x`) (pickSpec` x`) && (oneHot` x `&& rect` x`)`	6.49

Table 4.5: Times taken (seconds) to solve the constraints characterising Forward and Backward Reach on the Mux_2 problem with a call-depth bound of twelve.

The property `prop_pick` holds for all inputs, and consequently the call to `==` never returns `False`. So in both problems, the target is unreachable.

Observation 12 Forward Reach takes three orders of magnitude less time than Basic Reach to find all solutions to the Mux_1 and Mux_2 problems with a construction depth bound of six. □

Again, the performance benefit of Forward Reach can be explained by the restrictive antecedent. To illustrate, `prop_pick` is satisfied by

`(True,` x`):(True,` y`):`z

for all x, y and z; in this case, the `oneHot` precondition is not met. Similarly, `prop_pick` is also satisfied by

`(True,[]):(False,` x`:`y`):[]`

for all x and y; this time the `rect` precondition is not met.

Observation 13 Of the 1,466,766 inputs to `prop_pick` with a construction depth of six or less, only 3,120 satisfy the antecedent. This is consistent with the three orders of magnitude performance difference between Basic and Forward Reach. □

Observation 14 Forward Reach takes three orders of magnitude less time than Backward Reach to find all solutions to the Mux_2 problem with a call-depth bound of twelve. □

Whereas Forward Reach first solves the restrictive `oneHot` and `rect` constraints, Backward Reach first finds values returned by `pick` that are pre-

fixes of, but smaller in length than, values returned by `pickSpec`. The two approaches can be characterised by the constraints 1 and 2 respectively in Table 4.5, where the `prefix` function is defined as:

```
prefix [] _ = False
prefix (x:xs) [] = True
prefix (x:xs) (y:ys) = x == y && prefix xs ys
```

As illustrated above, the forward approach allows pruning. But so too does backward approach. For example, Backward Reach can determine that the input

`[(True, False:False:[]), (True, True:`x`:`y`)]`

does not reach the target for any x and y.

Observation 15 The timings in Table 4.5 suggest that the constraint ordering introduced by Forward Reach is better than that introduced by Backward Reach. This is consistent with the fact that Forward Reach is more efficient on the Mux_2 problem. However, the difference in the time taken to solve constraint 2 and the time taken by Backward Reach to solve the Mux_2 problem is significant. This suggests that constraint 2 is not an accurate characterisation of Backward Reach's behaviour. □

There is an explanation why Backward Reach is not as efficient as one might expect on the Mux_2 problem. With a call-depth bound of twelve, there are ten progressively deeper paths to the target. Backward Reach considers each of these paths as entirely *separate* problems. In doing so, it effectively solves constraint 2 in Table 4.5 ten times, each time demanding slightly more input in order to reach a deeper target.

4.7.4 Huffman compression

This section considers a program for Huffman compression implemented by Bird [14]. It contains functions for both compression and decompression of strings, along with a function for building Huffman trees. A Huffman tree is a binary tree with symbols at its leaves, and the path from the root to any symbol describes the unique, variable-length sequence of zeros and ones representing that symbol.

Two properties of Bird's program are considered. The first states that the decompresser (`decode`) is the inverse of the compressor (`encode`).

```
prop_decEnc cs =
    length ft > 1 ==> decode t (encode t cs) == cs
  where ft = collate cs
        t  = mkHuff ft
```

Here, `collate` builds a frequency table (`ft`) for an input string, and `mkHuff` builds a Huffman tree from a frequency table.

The second property asserts that `mkHuff` produces *optimal* Huffman trees, that is, for all binary trees t, if t is a Huffman tree then it has a *cost* no less than that produced by `mkHuff`. A binary tree is only a Huffman tree (as determined by `isHuff`) if it contains every symbol in the source text exactly once. The cost of a Huffman tree is defined as the sum of each symbol's frequency multiplied by its depth in the Huffman tree.

```
prop_optimal cs t =
    isHuff t cs ==> cost ft t >= cost ft (mkHuff ft)
  where ft = collate cs
```

The Huffman$_1$ problem targets refutation of `prop_decEnc`. Huffman$_2$ targets refutation of `prop_optimal`. Both properties are correct, and consequently, in each problem, the target is unreachable.

Observation 16 The times taken by Basic and Forward Reach to solve the Huffman$_1$ problem are very similar. At construction depths three to seven, Basic Reach is slightly more efficient than Forward Reach, and at depth eight, Forward Reach is slightly more efficient than Basic Reach. □

Observation 17 Of the 109,601 inputs to `prop_decEnc` at construction depth 8 or less, 109,564 satisfy the antecedent. □

There is little scope to satisfy `prop_decEnc` on a partially-instantiated input. It is a hyper-strict function, fully demanding all inputs that satisfy the antecedent. And indeed the vast majority of inputs do satisfy the antecedent.

Although the performance difference is very small, it is perhaps surprising that Basic Reach is occasionally slightly more efficient than Forward

Reach. Basic Reach freshly evaluates the property from scratch on each input, whereas Forward Reach shares some evaluation of the property on different inputs with common sub-structures. Therefore, one might expect Forward Reach always to be more efficient. However, Basic Reach is a deterministic evaluator and Forward Reach is a non-deterministic one, and there is some overhead in implementing non-determinism. Using list comprehensions for example, intermediate results are frequently wrapped in lists and subsequently unwrapped again.

Observation 18 Forward Reach is more efficient than Basic Reach on the Huffman$_2$ problem, allowing deeper searching within a 40-minute time bound. □

Observation 19 Of the 7,142,334 inputs to `prop_optimal` with construction depth six or less, only 1,383 satisfy the `isHuff` antecedent. This is consistent with the large performance difference between Basic and Forward Reach. □

4.7.5 Turner's abstraction algorithm

This section considers Peyton Jones implementation [78] of Turner's abstraction algorithm. The syntax for applicative expressions which may contain Turner combinators is as follows.

```
type Var   = Nat
type Const = Nat
data Exp   = Exp :@ Exp | V Var | F Comb
data Comb  = I | K | B | C | S | B' | C' | S' | Fun Const
```

Expressions contain applications, variables, and combinators. A combinator is a Turner combinator or a user-defined function name. Variables and function names are represented by natural numbers. The abstraction algorithm `abstr`, defined in Figure 4.11, abstracts a variable from an expression by introducing combinators. The property of interest here is Turner's law of abstraction [92], stating that if a variable is abstracted from an expression, and the resulting expression is applied to that variable, then one ends up with the original expression again.

```
abstr v (f :@ x)       = opt (F S :@ abstr v f :@ abstr v x)
abstr v (V w) | v == w = F I
abstr v e              = F K :@ e

opt (F S :@ (F K :@ p) :@ (F K :@ q))    = F K :@ (p :@ q)
opt (F S :@ (F K :@ p) :@ F I)           = p
opt (F S :@ (F K :@ p):@(F B :@ q :@ r)) = F B' :@ p :@ q :@ r
opt (F S :@ (F K :@ p) :@ q)             = F B :@ p :@ q
opt (F S :@ (F B :@ p :@ q):@(F K :@ r)) = F C' :@ p :@ q :@ r
opt (F S :@ p :@ (F K :@ q))             = F C :@ p :@ q
opt (F S :@ (F B :@ p :@ q) :@ r)        = F S' :@ p :@ q :@ r
opt e = e

abstr' (v, e) = abstr v e
```

Figure 4.11: Turner's abstraction algorithm.

```
prop_abstr (v, e) =
       noTurner e ==> reduce (abstr v e :@ V v) == e
```

Here, `noTurner` checks that an expression contains no Turner combinators and `reduce` applies the reduction rules for Turner combinators to an expression. Requiring that `e` does not contain any Turner combinators ensures that `reduce` only removes combinators introduced by `abstr` so that the equality holds.

The first benchmark problem – Turner_1 – targets refutation of `prop_abstr`. The second – Turner_2 – marks `abstr'` (see Figure 4.11) as the source and targets evaluation of the right-hand-side of the third equation of the `opt` function. The Turner_2 problem is not motivated by expressions left uncovered by QuickCheck, but simply by the aim of finding a lambda expression for which Turner's algorithm introduces a `C'` combinator.

Observation 20 Forward Reach is more efficient than Basic Reach on both the Turner_1 and Turner_2 problems. □

Observation 21 Of the 8,477 expressions with depth three or less, only 84 satisfy `noTurner`. This is consistent with the performance benefit of Forward Reach. □

Although there is no antecedent in the Turner_2 problem, there is scope for Forward Reach to determine that some partially-instantiated inputs cannot lead to evaluation of the target. For example, the input

`(0, V 0 :@ F` c`)`

does not lead to evaluation of the target for any combinator c; a richer applicative structure is needed for `abstr'` to introduce a `C'` combinator.

Observation 22 On the Turner_2 problem with a call-depth bound of eleven or twelve, Backward Reach is an order of magnitude more efficient than Forward Reach. □

Observation 23 On the Turner_2 problem with a call-depth bound of twelve, there are fifteen separate paths of increasing depth from the source to the target. All solutions found by Forward and Backward Reach result from following just one of these paths – in particular the *shallowest* one. Backward Reach requires 1.22 seconds to find all solutions that follow the shallowest path, and 0.04 seconds to find all solutions that follow the other fourteen paths. □

Compared to Backward Reach, Forward Reach performs much more work before hitting the call-depth bound. Working backwards, constraints at deeper call-depths are solved first, and as they are more likely to fail due to insufficient remaining call-depth, the search space is cut down, earlier in the analyser. This explanation is consistent with the observation that Backward Reach quickly determines that all but the shallowest path have no solutions.

4.8 Conclusions

Forward Reach Compared to Basic Reach, Forward Reach often requires significantly less time to find all solutions to a reach problem (Observations 1, 7, 12, 18 and 20). It is particularly effective when the result of the source function can be determined when applied to a partially-instantiated input; in such cases, a whole class of fully-instantiated inputs can be pruned from the search space (Section 4.4.2). The smaller the amount of input demanded

by the source function in order to produce a result, the lazier it is, and the bigger the benefit of using Forward Reach.

The benefit of Forward Reach is very apparent when the source function is a property with a restrictive antecedent, and the target can only be reached by satisfying the antecedent (Observations 2, 8, 13, 19 and 21). But there can be a benefit even if this is not the case (Observation 20) – it is the laziness of the source function that is important.

When tested with QuickCheck, some expressions in the benchmark properties are often left uncovered (Section 4.7). In one case, Forward Reach can reveal counter examples that are not found after a substantial period of QuickCheck testing (Observations 6 and 7).

Sometimes properties contain antecedents that are composed of several conjuncts, and the *order* in which such conjuncts are placed can have a significant impact on how long it takes to find a counter example (Observation 9). Ideally, conjuncts which demand the least amount of input to produce a result – the laziest conjuncts – should come first. However, determining the laziness of a function is not always easy, and may require some experimentation.

When the source function is *hyper-strict* – fully demanding all or most of its input before returning a result – the benefit of Forward Reach is much smaller (Observations 16 and 17).

Backward Reach On property-refutation problems, Forward and Backward Reach behave identically. But when the target is placed more liberally, the two often behave and perform quite differently.

One of the big differences is the order of evaluation. Forward Reach begins by evaluating expressions in the source function, whereas Backward Reach begins by evaluating expressions around the target expression. Sometimes more restrictive constraints are introduced earlier in the analysis using Forward Reach (Observation 14), and sometimes the same is true using Backward Reach (Observations 3, 4 and 22).

An advantage of Backward Reach is that expressions at *deeper* call-depths are evaluated first. Such constraints are more likely to break the call-depth

bound, offering increased scope for failure earlier in the analysis (Observation 23). In contrast, Forward Reach may perform much work only to determine very late that there is insufficient call-depth to reach the target.

Another disadvantage of Forward Reach is that it sometimes continues to reduce an expression graph to normal form even when there is no reference to the target or to a function on the call-path to the target (Observation 5). In Backward Reach, the target is always referenced and when it becomes unreferenced the current search path is immediately abandoned.

A disadvantage of Backward Reach is that when there are multiple paths to the target, each path is considered as an entirely separate problem (Observations 11 and 15). Some evaluation which is common to each path may be repeated.

4.9 Related work

4.9.1 Functional-logic programming in Curry

Curry [35] is a general-purpose programming language that integrates the main features of functional and logic programming. From functional programming, it takes lazy evaluation, higher-order functions, static typing and a Haskell-like syntax. From logic programming it takes logical variables, non-determinism and constraints.

Curry has two main strategies for dealing with logical variables: *narrowing* and *residuation*. When the value of a logical variable is demanded in order for evaluation to proceed, the narrowing strategy is to "guess" a value for that variable by non-deterministically instantiating it to each possible constructor of the appropriate type. (Implementations of Curry use *needed narrowing* [2], in which variables are only instantiated by the smallest amount necessary for evaluation to continue.) In contrast, the residuation strategy is to suspend evaluation of the current conjunct and evaluate another conjunct. If evaluation of the latter conjunct instantiates the variable suspended on by the former, then the former is resumed.

The narrowing strategy supports *polymodal* functions in Curry in the same sense that predicates are polymodal in Prolog. That is, further to comput-

ing an unknown output from known inputs, functions in Curry can also be used, for example, to compute unknown inputs from a known output. The residuation strategy supports both *constraint programming*, whereby constraints are suspended until sufficient data becomes available to check them, and *inter-process communication*, whereby one process can block (suspend) on a logical variable until another process instantiates it.

Curry and Reach are similar in that they both perform needed narrowing of Haskell-like programs – indeed the use of needed narrowing in Reach is inspired by its use in Curry. The main difference is that Curry provides a host of logic programming features and aims to facilitate general-purpose programming, whereas Reach is an analyser for plain functional programs and aims to facilitate property-based testing. In particular, Reach is concerned with the problem of causing evaluation of marked expressions in a program, and Curry is not.

4.9.2 Property-directed generation of test-data

Lindblad presents a method that takes predicates written in Haskell and generates data satisfying the predicate [59]. The primary motivation is to automatically generate relevant test data for program properties, avoiding the need to write custom QuickCheck generators. The method can be used in one of two ways.

1. To generate test data satisfying a particular constraint, such as the antecedent of a property, which can then be used to test programs written in any language.

2. To refute a property written in Haskell by finding inputs that satisfy the negated property.

Lindblad's method is based on a technique called *lazy instantiation*. Lazy instantiation evaluates a predicate step-by-step while instantiating the input as demanded by the predicate. If the predicate evaluates to false when the input is not fully instantiated, then a whole class of inputs is pruned from the generated test-set.

Lazy instantiation is very similar to needed narrowing, but there are also a number of differences. Most notably, Lindblad introduces a new language

construct, "select case", that permits implementation of a new *parallel conjunction* operator `>&<`. The idea is that an expression of the form $p\ x$ `>&<` $q\ x$ evaluates $p\ x$ and $q\ x$ in parallel. If at any stage a partially instantiated x falsifies p or q then the conjunction immediately evaluates to false. This is in contrast to standard conjunction, which may further instantiate x as demanded by p even if x already falsifies q. Lindblad shows that the use of parallel conjunction over normal conjunction can reduce the size of the search space, and can reduce the need to "tweak the order of conjuncts in the property" [59]. That is, Lindblad offers a very attractive solution to the problem noted in Observation 9, Section 4.7.2.

Lazy instantiation and Reach are similar in that they both use needed narrowing to aid property-based testing, although the two have been developed independently. In particular, Lindblad compares the performance of lazy instantiation and exhaustive testing on a number of benchmark properties, and observes that lazy instantiation is especially beneficial when checking properties with restrictive antecedents. Lindblad also observes that QuickCheck has difficulty in automatically testing such properties.

One of the main differences between lazy instantiation and Reach, other than the idea of parallel conjunction, is that Reach tackles a more general problem. Reach aims to cause evaluation of a marked target expression in a program and property-refutation is a special case of this problem.

4.9.3 Specification-based testing of Java programs

Khurshid proposes a framework for automated specification-based testing of Java programs, called *TestEra* [51]. Specifications take the form of pre and post-conditions on Java methods, and are expressed in *Alloy* [47], a declarative language based on first-order logic. An attractive feature of Alloy formulae is that they can be compiled by a tool called *the Alloy Analyzer* to propositional logic and solved using a SAT solver.

To test a Java method with a pre and post-condition, TestEra proceeds as follows. First, all values up to some bound satisfying the pre-condition are determined using the Alloy Analyzer. Second, each of these Alloy values is converted to a Java value and passed to the Java method. Finally, the result of the Java method is turned into an Alloy value, and the post-condition is

checked by the Alloy Analyzer. If the post-condition is not satisfied, TestEra reports a counter-example.

The idea behind TestEra is to use the pre-conditions on methods to automatically generate relevant test inputs. Khurshid observes that such inputs cannot be feasibly generated by random or even brute-force techniques. Khurshid points out that one of the major limitations of TestEra is that it "fails to explore program behaviours that are witnessed by large inputs only" [51]. However, Khurshid also notes that experiments have been performed which "show that it is feasible to achieve full statement and branch coverage for several data-structure benchmarks by testing on all inputs within a small input size" [51].

To use TestEra, the programmer must master two different languages: Java and Alloy. In Reach, programs and specifications can both be written in Haskell. Another difference is that TestEra solves constraints using the Alloy Analyzer (which in turn uses a SAT solver) as opposed to needed narrowing. It would be interesting to compare the two approaches, both in terms of performance and expressiveness of the constraint language.

4.9.4 Static checking and theorem proving

Mitchell proposes a program analyser called Catch [66] to automatically prove that a given Haskell program is free from pattern-match errors. In essence, this corresponds to proving that all calls to Haskell's `error` function are unreachable.

Like Reach, the Catch analyser operates on first-order functional programs. However, Mitchell proposes a transformation from higher-order programs to first-order ones so that Catch can be applied to full, higher-order, Haskell programs. Mitchell applies Catch to a range of such programs, and in each case observes one of three main outcomes.

1. Catch proves that the program is safe from pattern-match errors.
2. Catch warns of a potential crashing behaviour that in fact cannot occur.
3. Catch warns of genuine crashing behaviour.

Mitchell reports that in the second case, it is often possible to make a slight modification to the program so that Catch can prove it safe. In the course of his experiments, Mitchell discovers and fixes bugs in some existing, open-source Haskell programs.

By wrapping calls to `error` in a `target`, Reach can be used to answer a similar question to Catch. However, Catch and Reach give quite different answers; Mitchell sums it up as follows:

> *the tools are complementary: Reach can be used to find examples causing non-exhaustive patterns to fail, Catch can be used to prove there are no such examples.* [66]

A piece of work similar in spirit to Catch is Xu's *extended static checker for Haskell* [98] called ESC/Haskell (in homage to ESC/Modula-3 [84] and ESC/Java [28]). In contrast to Catch which is designed to work on programs with no additional annotations, ESC/Haskell allows programs to express *contracts* – similar to contracts in Meyer's Eiffel [64]. This allows richer properties about programs, further to pattern-match safety, to be expressed and proven.

In ESC/Haskell, contracts are expressed using normal Haskell code: no special property-language is required. Programs and contracts are together transformed into Haskell expressions containing `BAD` constructors which represent parts of the expression that should be unreachable if the program is valid. ESC/Haskell applies a range of simplification rules to the expression with the aim of removing all references to `BAD`. In particular, ESC/Haskell is able to use the contracts of all the functions in the program in the simplification process. So although writing more contracts increases the amount of verification to be done, it also increases the amount of knowledge available to the simplifier and hence the ability of the simplifier to prove theorems.

Xu reports that ESC/Haskell is able to verify a number of sorting algorithms. For example, insertion sort can be verified provided that the function to insert an element into an ordered list has a contract stating that it preserves the orderedness of the list.

The difference between Reach and ESC/Haskell is much the same as the difference between Reach and Catch: Catch and ESC/Haskell aim to verify

certain properties about programs whereas Reach aims to refute them.

Both Catch and ESC/Haskell are fully automatic approaches to verifying properties. These approaches are not always successful, and can leave valid properties unproven. Traditional theorem provers such as HOL Light [37] take a more interactive approach. The programmer can define functions and associated properties and prove that properties are equivalent to true by manual application of a series of semantics-preserving rewrites. HOL Light provides libraries of pre-proven theorems and useful proof tactics to assist the programmer in this task. Interactive proof assistants for functional languages, such as Starship [82] and Path [91], have been developed, but none is currently in wide use.

4.10 Limitations and future work

4.10.1 Higher-order functions

Adapting Forward Reach to support higher-order functions raises two separate problems.

1. The reduction of expressions containing applications of the more general form $e_0\ e_1$.
2. The synthesis of functional values as inputs to a higher-order source function.

The first problem can be solved using standard reduction rules [57] and will not be elaborated here. The second is more difficult, and requires a way to deal with applications of uninstantiated variables to expressions. Suppose that an expression of the form $v\ e$ is to be evaluated. One possible way to proceed is to bind v to the lambda expression

$$\lambda x.\ \texttt{case}\ x\ \texttt{of}\ \{\ c_1\ \vec{v_1}\ \mapsto\ w_1\ \vec{v_1}\ ;\ \cdots\ ;\ c_n\ \vec{v_n}\ \mapsto\ w_n\ \vec{v_n}\ \}$$

where c_1 to c_n are the constructors of the type of the expression e and the variables in $\vec{v_1}$ to $\vec{v_n}$ and $\vec{w}$ are fresh unbound variables with construction depths equal to one less than that of v, provided that the construction depth of v is larger than zero. This approach bounds the depth of synthesised func-

tions by a combination of the depth to which arguments may be evaluated and the depth of possible results. The types of the fresh variables need to be carefully inferred from the type of the original variable v. Existing work on *higher-order narrowing* [36] is also applicable.

Adapting Backward Reach to support higher-order functions raises an additional problem. The backward propagation rules must deal with the more general form of application $e_0\ e_1$. That is, one must consider the possibility of applying a target to an expression, and an expression to a target. One possible way to proceed is to extend the target context to allow lambda expressions as well as data constructors. A lambda in a target context can then be applied to an expression, and the partial application of a function to a target can introduce a lambda in the target context.

A different approach to supporting higher-order programs is to transform them to first-order ones. Indeed, Mitchell has recently developed such a transformation precisely for the purpose of applying a first-order analyser to higher-order programs [66]. Furthermore, Antoy and Tolmach have proposed an approach to higher-order narrowing by encoding higher-order terms in a first-order data type and using standard first-order narrowing strategies [3].

4.10.2 Garbage collecting the target

One of the problems with Forward Reach is that it continues to search for a target even when there is no reference to the target or to a target-reaching function in the expression graph being reduced. One possible way to detect such a situation is to augment Forward Reach with a reference-counting garbage collector [44]. If the target ever becomes unreferenced, the current search path can be safely abandoned.

4.10.3 Initial inlining

One of the first tasks performed by Backward Reach is to inline the program to produce an *expression* containing exactly one occurrence of the target. If there are multiple paths to the target, then multiple such expressions are produced, and each one is analysed separately. An alternative approach is to modify the analyser to operate directly on *programs*. This can be done

by applying the existing rules initially to the right-hand-side of the function containing the target. After applying the rules, the function will have the form $f\ \vec{v}$ = $\{e\}$ and the analyser can continue by non-deterministically replacing every call $f\ \vec{e}$ in the program with $\{e\}$ and unifying the variables in $\vec{v}$ with the expressions in $\vec{e}$. This alternative approach has the potential advantage of sharing some evaluation between different paths to the target.

4.10.4 Parallel conjunction

Following Lindblad [59], a parallel conjunction operator can be used to reduce the search space explored when properties contain antecedents that are composed of several conjuncts. In addition, there is much scope for using parallel conjunction to reduce the search-space explored by Backward Reach. Consider an expression of the following form.

$$\texttt{case}\ e_0\ \texttt{of}\ [\ \cdots,\ c_i\ \vec{v_i}\ \mapsto\ \texttt{case}\ e_1\ \texttt{of}\ [\ \cdots,\ c_j\ \vec{v_j}\ \mapsto\ f\ \{e\},\ \cdots\ ],\ \cdots\ ]$$

To reach the target, there are two constraints that must be solved:

1. $e_0 = c_i\ \vec{v_i}$, and
2. $e_1 = c_j\ \vec{v_j}$.

Currently Backward Reach will unfold f, solve any constraints that result, and then solve constraint 2 followed by constraint 1. An alternative approach is to solve all the constraints together using parallel conjunction.

4.10.5 Arithmetic constraint solvers

One of the major limitations of Reach is that, due to the bounds on construction-depth and call-depth, it can only find *small* inputs that reach the target. This is particularly concerning when programs use data types such as integers and characters. For example, it only takes an expression such as

```
if x > 100 then target True else False
```

to render Reach useless. One way to improve this situation would be to support primitive data types such as integers and characters, and use spe-

cialised decision procedures [55] to ensure the satisfiability of constraints involving values of such types.

4.10.6 Efficiency of implementation

Whereas programs tested by QuickCheck can be compiled using a state-of-the-art optimising compiler, Reach is only a simple interpreter implemented at a high level of abstraction. Therefore one can generate and run tests much more quickly using QuickCheck compared to using Reach. Reach would be more useful if it were more efficient. It is not unreasonable to expect an order of magnitude performance improvement by implementing Reach at a lower level of abstraction, and another order of magnitude by using compilation techniques instead of interpretation.

4.10.7 Generalisations

Reach has been applied mainly to property refutation and program coverage problems. It can also be used to aid program understanding by placing the target in a complex part of the program, or to crash programs by placing the target in an undefined case alternative or around a call to Haskell's `error` function.

Reach can also be generalised by allowing *multiple* targets with a *reach-any* or a *reach-all* semantics. In fact, Forward Reach as it stands already supports multiple targets with a reach-any semantics. More generally, a reach-any semantics can be achieved by taking several copies of the program, where each copy contains only one target from the original, and applying Reach separately to each. A reach-all semantics can be achieved in almost the same way, but instead of applying Reach separately to each copy, it is applied *sequentially* using the final state from one run as the initial state for the next.

Chapter 5

A Library for Demand-Driven Testing

Despite the advantages offered by Reach through the use of needed narrowing, it has many limitations. In particular, it only works on a simple subset of Haskell and the implementation is not very efficient. This chapter captures some of the strengths of Reach in a *small library* for a standard lazy functional language, simulating needed narrowing using a plain lazy evaluator. The library is a lot more efficient than the existing Reach implementation, and can be imported and used by any Haskell program.

5.1 Introduction

Claessen and Hughes propose an attractive approach to *property-based testing* of Haskell programs, as implemented in their library *QuickCheck* [20]. Properties relating the component functions of a program are specified in Haskell itself. The simplest properties are just boolean-valued functions, in which the body is interpreted as a predicate universally quantified over the argument variables, and a small library of operators provides for variations such as properties that are conditionally true. QuickCheck exploits Haskell's *type classes* to check properties using test-sets of *randomly generated* values for the universally-quantified arguments. If a failing case is discovered, testing stops with a report showing the counter-example.

Although QuickCheck is widely used by Haskell developers, and is often very effective, it has two main drawbacks:

Drawback 1 If failing cases are rare, none may be tested *even though some of them are very simple*; this seems to be an inevitable consequence of using randomly selected tests.

Drawback 2 When testing properties with *restrictive antecedents*, the programmer must typically resort to writing a *custom generator* in order to produce a good distribution of relevant test data.

5.1.1 SmallCheck

SmallCheck [83] is variation of QuickCheck which uses a different approach to the generation of test-data. Instead of random testing, properties are tested for *all the finitely many values up to some depth*, progressively increasing the depth used. For data values, depth means depth of construction. For functional values, it is a measure combining the depth to which arguments may be evaluated and the depth of possible results.

The principal motivation for SmallCheck is summarised by the following observations, akin to the *small scope hypothesis* behind model-checking tools such as Alloy [47].

1. If a program fails to meet its specification in some cases, it *almost always* fails in some *simple* case. Or in contrapositive form:

2. If a program does not fail in any simple case, it *hardly ever* fails in *any* case.

A successful test-run using SmallCheck can give exactly this assurance: specified properties do not fail in any simple case. In this way, SmallCheck addresses the first stated drawback of QuickCheck.

5.1.2 Lazy SmallCheck

This chapter presents *Lazy SmallCheck*. Lazy SmallCheck is like SmallCheck, but uses a slightly different approach to the generation of test data. Instead of fully-defined inputs, *partially-defined* inputs are generated that are progressively refined as demanded by the property under test. The generation of a *single* partially-defined input can eliminate the need to generate *many* fully-defined ones. This allows Lazy SmallCheck to test the same input-space as SmallCheck, but sometimes using significantly fewer tests. In particular, tests that falsify antecedents are often avoided, addressing the second stated drawback of QuickCheck.

5.1.3 Structure of this chapter

The rest of this paper is arranged as follows. Sections 5.2 and 5.3 review QuickCheck and SmallCheck respectively. Section 5.4 introduces Lazy SmallCheck, and Section 5.5 describes the Lazy SmallCheck implementation. Section 5.6 is a comparative evaluation. Section 5.7 discusses related work. Section 5.8 suggests avenues for future work and Section 5.9 concludes.

5.2 QuickCheck: a review

5.2.1 Arbitrary types and testable properties

QuickCheck defines a class of `Arbitrary` types for which there are random value generators. There are predefined instances of this class for most `Prelude` types. It also defines a class of `Testable` property types for which there is a method mapping properties to test computations. The `Testable` instances include:

```
instance Testable Bool
instance (Arbitrary a, Show a, Testable b)
  => Testable (a -> b)
```

Any `Testable` property can be tested automatically for some pre-assigned number of random values using

```
quickCheck :: Testable a => a -> IO ()
```

a class-polymorphic test-driver. It reports either success in all cases tested, or else a counterexample for which the property fails.

Example 1 Suppose the program being tested includes a function

```
isPrefix :: Eq a => [a] -> [a] -> Bool
```

that checks whether its first argument is a prefix of its second. One expected property of `isPrefix` can be specified as follows.

```
prop_isPrefix :: [Int] -> [Int] -> Bool
prop_isPrefix xs xs' = isPrefix xs (xs++xs')
```

The argument variables `xs` and `xs'` are understood to be *universally quantified*: the result of `prop_isPrefix` should be `True` *for all* finite, fully defined `xs` and `xs'`. As `prop_isPrefix` has a `Testable` type — its explicitly declared monomorphic type enables appropriate instances to be determined — it can now be tested.

```
*Main> quickCheck prop_isPrefix
OK, passed 100 tests.
```

Alternatively, if `isPrefix` actually interprets its arguments the other way round, the output from `quickCheck` might be

```
Falsifiable, after 1 tests:
[1]
[2]
```

as the property then fails for `xs=[1]`, `xs'=[2]`. □

5.2.2 Generators for user-defined types

For properties over user-defined types, appropriate `Arbitrary` instances must be written to generate random values of these types. QuickCheck provides various functions that are useful in this task.

Example 2 Consider the following data-type for logical propositions. To shorten the example, connectives are restricted to negation and disjunction.

```
data Prop = Var Name | Not Prop | Or Prop Prop
```

```
instance Arbitrary Prop where
  arbitrary = sized arbProp
    where arbProp 0 = liftM Var arbitrary
          arbProp n = frequency
            [ (1,liftM Var arbitrary)
            , (2,liftM Not (arbProp (n-1)))
            , (4,liftM2 Or (arbProp (n `div` 2))
                           (arbProp (n `div` 2))) ]
```

Figure 5.1: A QuickCheck `Arbitrary` instance for `Prop`.

Assuming that an `Arbitrary Name` instance is defined elsewhere, Figure 5.1 shows how a QuickCheck user might define an `Arbitrary Prop` instance. The `sized` function applies its argument to a random integer. The `frequency` function also abstracts over a random source, choosing one of several weighted alternatives: in the example, the probability of a `Var` construction is 1/7. □

As this example shows, defining generators for recursive types requires careful use of controlling numeric parameters.

5.2.3 Conditional properties

Often the body of a property takes the form of an implication, as it is only expected to hold under some condition. If implication were defined simply as a boolean operator, then cases where the condition evaluates to `False` would count as successful tests. Instead QuickCheck defines an implication operator `==>` with the signature

```
(==>) :: Testable a => Bool -> a -> Property
```

where `Property` is a new `Testable` type. Test cases where the condition fails are discarded and do not count as successful tests.

Example 3 Suppose that an abstract data type for sets is to be implemented. One possible representation is an ordered list. Of course, sets are unordered collections, but an ordered list permits the uniqueness of the elements to be preserved more efficiently by the various set operations.

```
type Set a = [a]
```

Each set operation may assume that the lists representing the input sets are ordered, and must ensure that the same is true of any output sets. For example, the operation to insert an element into a set, of type

```
insert :: Ord a => a -> Set a -> Set a
```

should preserve the familiar `ordered` predicate on lists.

```
prop_insertSet :: Char -> Set Char -> Property
prop_insertSet c s =
  ordered s ==> ordered (insert c s)
```

Checking this property succeeds with the usual message "`OK, passed 100 tests`", but great care must be taken when testing conditional properties using QuickCheck. For example, if the property is rewritten as

```
prop_insertSet c s =
  ordered s ==> collect (length s) (ordered (insert c s))
```

then the distribution of the lengths of tested lists can be observed. For example:

```
*Main> quickCheck prop_insertSet
OK, passed 100 tests.
43% 0.
34% 1.
17% 2.
5% 3.
1% 4.
```

The distribution is skewed by the property's condition: the longer the list, the more likely it is to be rejected by the condition, and the less likely it will count as a successful test. What is worrying is that the majority of test cases generated are trivial, containing empty or singleton lists. □

Regarding the problem of conditional properties skewing the test distribution, the QuickCheck authors write:

> *There is a risk of this kind of problem every time we use conditional laws, so it is always important to investigate the proportion of trivial cases among those actually tested. The best solution, though, is to replace the condition with a custom test data generator.* [20]

However, collecting coverage information and writing custom generators both mean more work for the programmer. In particular, writing a custom generator has three big drawbacks.

1. Writing good custom generators can be hard. Indeed, regarding the kinds of bugs found by QuickCheck, the authors write:

   > *We have observed that the errors we find are divided roughly evenly between errors in test data generators, errors in the specification and errors in the program.* [20]

2. The property that *all and only* required values can be produced by the custom generator may be hard to verify.

3. Properties showing that some invariant condition is preserved (a common pattern) must express the condition twice, in two different ways: once as a generator in the pre-condition, and once as a predicate in the post-condition.

5.2.4 Counter examples

A small counter-example is in general easier to analyse than a large one. QuickCheck, although beginning each series of tests with a small size parameter and gradually increasing it, is in many cases unlikely to find a simplest counter-example. To compensate for this, QuickCheck users may write type-specific *shrinking* functions. However, writing shrinking functions requires extra work and the mechanism still does not guarantee that a reported counter-example is minimal.

5.3 SmallCheck: a review

5.3.1 Small values

SmallCheck re-uses many of the property-based testing ideas in QuickCheck. It too tests whether properties hold for finite total values, using type-driven generators of test cases, and reports counter-examples. But instead of generating test cases at random, it enumerates all *small* test cases exhaustively.

Almost all other changes follow as a result of this one. The principle SmallCheck uses to define *small values* is to bound their *depth* by some small natural number.

Small data structures Depth is most easily defined for the values of algebraic data types. As usual for algebraic terms, the depth of a zero-arity construction is zero, and the depth of a positive-arity construction is one greater than the maximum depth of a component argument.

Example 2 (revisited) Recalling the data-type `Prop` of logical propositions, suppose the `Name` type is defined by:

```
data Name = P | Q | R
```

Then all `Name` values have depth 0, and `Or (Not (Var P)) (Var Q)` of type `Prop` has depth 3. □

Small tuples The rule for tuples is a little different. The depth of the zero-arity tuple is zero, but the depth of a positive-arity tuple is just the maximum component depth. Values are still bounded as tuples cannot have recursive components of the same type.

Small numeric values For primitive *numeric types* the definition of depth is with reference to an imaginary representation as a data structure. So the depth of an *integer* `i` is its absolute value, as if it was constructed algebraically as $\texttt{Succ}^{\texttt{i}}\ \texttt{Zero}$. The depth of a *floating point* number $\texttt{s} \times 2^{\texttt{e}}$ is the depth of the integer pair `(s,e)`.

Example 4 The small floating point numbers, of depth no more than 2, are `-4.0`, `-2.0`, `-1.0`, `-0.5`, `-0.25`, `0.0`, `0.25`, `0.5`, `1.0`, `2.0` and `4.0`. □

5.3.2 Serial types

Instead of a class `Arbitrary` of types with a random value generator, SmallCheck defines a class `Serial` of types that can be enumerated up to a given depth.

Serial data For all the `Prelude` data types, `Serial` instances are predefined. Writing a new `Serial` instance for an algebraic datatype is very straightforward. It can be concisely expressed using a family of combinators `cons<N>`, generic across any combination of `Serial` component types, where `<N>` is constructor arity.

Example 2 (revisited) The `Prop` datatype has constructors `Var` and `Not` of arity one, and `Or` of arity two. A `Serial` instance for it can be defined by

```
instance Serial Prop where
  series = cons1 Var \/ cons1 Not \/ cons2 Or
```

assuming a similar `Serial` instance for the `Name` type. □

A `series` is just a function from depth to finite lists

```
type Series a = Int -> [a]
```

The first few members of the `cons<N>` family are defined by:

```
cons0 c = \d -> [c]
cons1 c = \d -> [c a   | d > 0, a <- series (d-1)]
cons2 c = \d -> [c a b | d > 0, a <- series (d-1)
                               , b <- series (d-1)]
```

Serial functions Further to the `series` method of the `Serial` class, SmallCheck also provides a `coseries` method for generation of of functions, analogous to QuickCheck's `coarbitrary`.

5.3.3 Test coverage and counter examples

Just as QuickCheck has a top-level function `quickCheck` so SmallCheck has `depthCheck d` and `smallCheck d`.

```
depthCheck, smallCheck :: Testable a => Int -> a -> IO ()
```

The function `depthCheck d` tests a property for all inputs up to depth `d`, and `smallCheck d` applies `depthCheck` at increasing depths from `0` to `d`. If a property is refuted by `smallCheck d` then a counter-example of minimal depth is reported. If a property is not refuted, then a clearly-defined portion of the input-space on which it holds is reported – namely all inputs up to

depth `d`. In each case, the SmallCheck user learns something useful that the QuickCheck user would not.

5.3.4 Existential properties

SmallCheck supports *existential* quantifiers. Testing a random sample of values as in QuickCheck would rarely give useful information about an existential property: often there is a unique witness and it is most unlikely to be selected at random. But SmallCheck can exhaustively search for a *small* witness. There are several existential variants, but the basic one has the following signature.

```
exists :: (Show a, Serial a, Testable b) =>
  (a -> b) -> Property
```

The interpretation of `exists f` is that for some argument `x` testing the result `f x` succeeds.

Example 1 (revisited) Recall the property of `isPrefix` specified earlier.

```
prop_isPrefix xs xs' = isPrefix xs (xs++xs')
```

This property is *necessary* but not *sufficient* for a correct `isPrefix`. For example, it holds for the erroneous definition

```
isPrefix []     ys     = True
isPrefix (x:xs) []     = False
isPrefix (x:xs) (y:ys) = x==y || isPrefix xs ys
```

or even for an `isPrefix` that always returns `True`! In terms of the following full specification for `isPrefix`

$$\forall \texttt{xs} \forall \texttt{ys} (\texttt{isPrefix xs ys} \iff \exists \texttt{xs'} (\texttt{xs++xs'} = \texttt{ys}))$$

the partial specification `prop_isPrefix` captures only the $\Longleftarrow$ direction – re-expressing the existential implicitly by the introduction of `xs'` rather than `ys` as the second variable in the property. Viewing `isPrefix` as a decision procedure, `prop_isPrefix` assures its *completeness* but ignores its *soundness*.

Using SmallCheck, one can test for soundness too. The $\Longrightarrow$ direction of the specification can be expressed like this:

```
prop_isPrefixSound xs ys =
  isPrefix xs ys ==>
    exists $ \xs' -> xs++xs' == ys
```

□

5.3.5 Conditional properties

Like QuickCheck, SmallCheck treats conditional properties as a special case. Whereas QuickCheck discards tests which do not satisfy the condition, SmallCheck simply reports how many satisfy the condition and how many do not.

Example 3 (revisited) The property `prop_insertSet` can be tested for all inputs up to a given depth using the `depthCheck` function.

```
*Main> depthCheck 7 prop_insertSet
Depth 7:
  Completed 109600 test(s) without failure.
  But 108576 did not meet ==> condition.
```

Over 99% of the tests generated by SmallCheck do not satisfy the condition, yet such tests cannot possibly refute the property. □

To increase the proportion of relevant tests generated by SmallCheck, a custom generator can be written. But this means more work for the programmer and has other drawbacks, as discussed in Section 5.2.3.

5.4 Lazy SmallCheck

A consequence of lazy evaluation in Haskell is that functions can return results when applied to *partially-defined* inputs. To illustrate, consider the following Haskell function `ordered`.

```
ordered [] = True
ordered [x] = True
ordered (x:y:zs) = x <= y && ordered (y:zs)
```

When applied to `1:0:`$\perp$, where $\perp$ is is a call to Haskell's `error` function, `ordered` returns `False`. Indeed, `ordered (1:0:`xs`)` is `False` for *every* xs. Thus, by applying a function to a *single* partially-defined input, one can observe its result over *many* fully-defined ones.

This ability to see the result of a function on many inputs in one go is very attractive to property-based testing: if a property holds for a partially-defined input then it will also hold for all fully-defined refinements of that input. The aim of Lazy SmallCheck is to avoid generating such fruitless refinements.

Lazy SmallCheck is a compatible subset of SmallCheck, currently only capable of checking first-order properties with universal quantifiers. It requires no extensions to standard Haskell other than the ability to detect evaluation of `error`, and this facility is already supported by the main Haskell implementations through *imprecise exceptions* [50].

5.4.1 Implication

In SmallCheck and QuickCheck the `==>` operator returns a value of type `Property`, allowing tests falsifying the antecedent to be observed. Lazy SmallCheck does not treat conditional properties specially, so `==>` simply has the type `Bool -> Bool -> Bool`.

Example 3 (revisited) In Lazy SmallCheck, the property `prop_insertSet` has a different type due to the new type of `==>`.

```
prop_insertSet :: Char -> Set Char -> Bool
prop_insertSet c s =
  ordered s ==> ordered (insert c s)
```

Recall that SmallCheck requires 109,600 tests to check the `prop_insertSet` for all inputs up to depth 7. Passing the property to Lazy SmallCheck's `depthCheck` function yields

```
*Main> depthCheck 7 prop_insertSet
OK, required 1716 tests at depth 7.
```

Both testing libraries use the same definition of depth, so the input-space checked by each is identical. The difference is that by generating partially-

defined inputs, Lazy SmallCheck is able to perform the check with fewer tests. To see why, observe that `prop_insertSet` applied to the partially-defined inputs $\bot$ and `'b':'a':`$\bot$ is `True`. □

Example 3 will be used throughout the next three sections to illustrate further points about Lazy SmallCheck.

5.4.2 Laziness is delicate

The set invariant in the example can be strengthened. Not only should the list representing a set be ordered, but it should also contain no duplicates, as expressed by the function `allDiff`.

```
allDiff [] = True
allDiff (x:xs) = x `notElem` xs && allDiff xs
```

The stronger invariant is expressed as follows.

```
isSet s = ordered s && allDiff s

prop_insertSet c s = isSet s ==> isSet (insert c s)
```

The `isSet` invariant reduces the number of tests generated by Lazy SmallCheck.

```
*Main> depthCheck 7 prop_insertSet
OK, required 964 tests at depth 7.
```

This is because some lists satisfy `ordered` but not `allDiff`, so there is increased scope for falsifying the condition without demanding the value of the element being inserted.

However, now suppose that the conjuncts of `isSet` are reversed.

```
isSet s = allDiff s && ordered s
```

Checking `prop_insertSet` now requires some twenty times more tests than the version with the original conjunct ordering.

```
*Main> depthCheck 7 prop_insertSet
OK, required 20408 tests at depth 7.
```

The problem is that `&&` evaluates its left-hand argument first, and `allDiff` is less restrictive than `ordered` in this case.

5.4.3 Parallel conjunction

When a property is composed of several sub constraints, like `isSet`, putting the most restrictive one first helps Lazy SmallCheck reduce the number of tests. But it is not always clear what the order should be. In fact, the best order may differ depending on the depth at which the property is checked.

Lazy SmallCheck provides the user with an alternative to normal conjunction called *parallel conjunction* and represented by `*&*`. A parallel conjunction is falsified if *any* of its conjuncts is. This is in contrast to a standard conjunction which returns $\perp$ if its first argument is $\perp$, even if its second could be falsified. Replacing `&&` with `*&*` reduces the need to place conjuncts in a particular order, and often decreases the number of required tests.

The function `*&*` is defined in a datatype called `Property`, extending `Bool` to allow the distinction between sequential and parallel conjunction. Boolean values must be explicitly lifted to properties. After switching to `*&*` the example property becomes

```
isSet :: Ord a => Set a -> Property
isSet s = lift (ordered s) *&* lift (allDiff s)

prop_insertSet :: Char -> Set Char -> Property
prop_insertSet c s =
  isSet s *=>* isSet (insert c s)
```

(`Property` implication in Lazy SmallCheck is denoted `*=>*`.)

The parallel variant of `isSet` reduces the number of tests compared to either of the non-parallel ones.

```
*Main> depthCheck 7 prop_insertSet
OK, required 653 tests at depth 7.
```

This is because some lists falsify `ordered` but not `allDiff`, e.g. `1:0:`$\perp$, and vice-versa, some falsify `allDiff` but not `ordered`, e.g. `0:0:`$\perp$. Now suppose again that the conjuncts are reversed.

```
isSet s = lift (allDiff s) *&* lift (ordered s)
```

This time the number of tests does not change, highlighting that parallel conjunction is not as sensitive to the order of the conjuncts.

```
*Main> depthCheck 7 prop_insertSet
OK, required 653 tests at depth 7.
```

Despite the advantages of parallel conjunction, it must be introduced manually, with care. An automatic rewrite is not possible, since switching to `*&*` may expose intended partiality in the second conjunct. The first conjunct of `&&` can be used as a guard which assures that the input has a certain property before evaluating the second one. With `*&*` such guards disappear and the property may crash as a result.

Having to lift booleans to properties does introduce an unfortunate notational burden. Overloaded booleans [9] would be really helpful here.

5.4.4 Strict properties

Not all properties are as lazy as `prop_insertSet`. To illustrate, consider the following function that turns a list into a set, throwing away duplicates.

```
set :: Ord a => [a] -> Set a
set = foldr insert []
```

One might like to verify that `set` always returns valid sets.

```
prop_set :: [Char] -> Bool
prop_set cs = isSet (set cs)
```

To return `True`, `prop_set` demands the entire input, so there is no scope for the property to be satisfied by a partially-defined input. Checking with SmallCheck yields

```
*Main> depthCheck 6 prop_set
Depth 6:
  Completed 1957 test(s) without failure.
```

and with Lazy SmallCheck:

```
*Main> depthCheck 6 prop_set
OK, required 2378 tests at depth 6.
```

Not only is Lazy SmallCheck of no benefit in this case, but it is worse than SmallCheck because it fruitlessly generates some partially-defined inputs as well as all the totally-defined ones.

5.4.5 Serial types

Like SmallCheck, Lazy SmallCheck provides a `Serial` class with a `series` method. But now a series has the type

```
type Series a = Int -> Cons a
```

From the user's perspective, `Cons a` is an abstract type storing instructions on how to construct values of type `a`. It has the following operations[1].

```
cons  :: a -> Series a
empty :: Series a
(\/)  :: Series a -> Series a -> Series a
(><)  :: Series (a -> b) -> Series a -> Series b
```

To illustrate, SmallCheck's `cons<N>` family of operators is defined in Lazy SmallCheck in the following fashion.

```
cons0 f = cons f
cons1 f = cons f >< series
cons2 f = cons f >< series >< series
```

So SmallCheck `Serial` instances defined using the standard pattern are written identically in Lazy SmallCheck.

Depth customisation The left-associative `><` combinator implicitly takes a depth `d`, and passes `d` to its left argument and `d-1` to its right argument. The result is that each child of a constructor is given depth `d-1`, as in SmallCheck. If the depth argument to `><` is zero, then no values can be constructed.

Example 9 Suppose a generator for rose trees [14] is to be written.

```
data Rose a = Node a [Rose a]
```

The standard list generator might be deemed inappropriate to generate the children of a node because each child would be generated to a different depth. Instead, the programmer might write

```
instance Serial a => Serial (Rose a) where
  series = cons Node >< series >< children
```

[1] Based on the API of an *applicative functor* [63].

where `children` is a generator for lists in which each element is bounded by the same depth parameter.

```
children d = list d
  where
    list = cons []
        \/ cons (:) >< const (series (d-1)) >< list
```

□

Primitive series Like in SmallCheck, a series can be defined as a finite list of finite *fully-defined* candidate values. This is achieved using the `drawnFrom` combinator.

```
drawnFrom :: [a] -> Cons a
drawnFrom xs = foldr (\/) empty (map cons xs) 0
```

The depth parameter `0` is irrelevant in the above definition, as it is not inspected by any of the combinators used. (Only the `><` combinator inspects the depth.)

Example 10 Here is the `Serial` instance for `Int`.

```
instance Serial Int where
  series d = drawnFrom [-d..d]
```

Using `drawnFrom`, primitive values of type `Integer`, `Char`, `Float` and `Double` are generated just as they are in SmallCheck. □

5.5 Implementation

The full implementation of Lazy SmallCheck can be found in Appendix B. This section presents in full a working but cut-down version of Lazy SmallCheck. Only code for parallel conjunction, the `Testable` class, and for displaying counter-examples and counting tests is omitted.

5.5.1 Partially-defined inputs

The central idea of Lazy SmallCheck is to generate *partially-defined* inputs, that is, inputs containing some calls to `error`. An example of a partially-

defined input of type `Prop` is

```
Or (Or (Var Q) (Not (error "_|_"))) (error "_|_")
```

Using imprecise exceptions [50], one can apply a property to the above term and observe whether it evaluates to `True`, `False`, or `error "_|_"`. However, since the input contains several calls to `error "_|_"`, it cannot be determined which one was demanded by the program. This is the motivation for tagging each `error` with its *position* in the tree-shaped term. A position is a list of integers, uniquely describing the path from the root of the term to a particular sub-term.

```
type Pos = [Int]
```

For example, the position `[1,0]` refers to the 0^{th} child of the root constructor's 1^{st} child. Lazy SmallCheck encodes such positions in the string passed to `error`. Using the helper function

```
hole :: Pos -> a
hole p = error (sentinel : map toEnum p)
```

the above example term of type `Prop` is now represented as follows.

```
Or (Or (Var Q) (Not (hole [0,1,0]))) (hole [1])
```

Each argument to `error` is prefixed with a `sentinel` character, allowing holes to be distinguished from `error`-calls occurring in the property.

```
sentinel :: Char
sentinel = '\0'
```

5.5.2 Answers

The data type `Answer` is used internally to represent the result of a property applied to a partially-defined input.

```
data Answer = Known Bool | Unknown Pos
```

Using imprecise exceptions, the following function converts the boolean result of a property into an answer.

```
answer :: Bool -> IO Answer
answer a =
  do res <- try (evaluate a)
     case res of
       Right b -> return (Known b)
       Left (ErrorCall (c:cs)) | c==sentinel ->
         return (Unknown (map fromEnum cs))
       Left e -> throw e
```

The functions `try`, `evaluate`, and `throw` are all exported by Haskell's `Control.Exception` library: `evaluate` forces evaluation of the boolean value passed to it, before returning it in an `IO` action; `try` runs the given `IO` action, and returns a `Right` constructor containing the action's result if no exception was raised, otherwise it returns a `Left` constructor containing the exception. If the exception represents a hole, then the position of demand is extracted and returned. Otherwise the exception is re-thrown.

When a property applied to a term yields `Unknown pos`, Lazy SmallCheck *refines* the term by defining it at position `pos`.

5.5.3 Refinement

The `Cons` data type is a little more complicated than the simple list it replaces in SmallCheck. Lazy SmallCheck must not only generate inputs but also take an existing input and refine it at a particular position.

```
data Cons a = Type :*: [[Term] -> a]
```

This data type can be read as follows: to construct a value of type `a`, one must have a sum-of-products representation of the type,

```
data Type = SumOfProd [[Type]]
```

and a list of conversion functions (one for each constructor) from a list of *universal terms* (representing the arguments to the constructor) to an actual value of type `a`. A universal term is either a constructor with an identifier and a list of arguments, or a hole representing an undefined part of the input.

```
data Term = Ctr Int [Term] | Hole Pos Type
```

Working with universal terms, the refinement operation can be defined generically, once and for all: it walks down a term following the route specified by the position of demand,

```
refine :: Term -> Pos -> [Term]
refine (Ctr c xs) (i:is) =
    map (Ctr c) [ls ++ y:rs | y <- refine x is]
  where (ls, x:rs) = splitAt i xs
refine (Hole p (SumOfProd sop)) [] = new p sop
```

and when it reaches the desired position, a constructor of the right type applied to the correct number of holes is inserted.

```
new :: Pos -> [[Type]] -> [Term]
new p sop =
  [ Ctr c (zipWith (ı -> Hole (p++[i])) [0..] ts)
  | (c, ts) <- zip [0..] sop ]
```

5.5.4 Series combinators

The `Series` combinators `cons`, `empty`, `\/` and `><` are defined in Figure 5.2, along with two auxiliary functions. The `conv` auxiliary allows a conversion function of type `Term -> a` to be obtained from the second component of a `Cons a` value. The `nonEmpty` auxiliary is used to ensure that a partially-defined value is not generated when there is no fully-defined refinement of that value within the depth limit.

5.5.5 Refutation algorithm

The algorithm to refute a property takes two parameters, the property to refute and an input term, and behaves as follows.

```
refute :: (Term -> Bool) -> Term -> IO ()
refute p x = do
  ans <- answer (p x)
  case ans of
    Known True  -> return ()
    Known False -> putStrLn "Counter example found"
                >> exitWith ExitSuccess
    Unknown pos -> mapM_ (refute p) (refine x pos)
```

```
cons :: a -> Series a
cons a d = SumOfProd [[]] :*: [const a]

empty :: Series a
empty d = SumOfProd [] :*: []

(\/) :: Series a -> Series a -> Series a
(a \/ b) d = SumOfProd (psa ++ psb) :*: (ca ++ cb)
  where SumOfProd psa :*: ca = a d
        SumOfProd psb :*: cb = b d

(><) :: Series (a -> b) -> Series a -> Series b
(f >< a) d =
    SumOfProd [ta:p | notTooDeep, p <- ps] :*: cs
  where SumOfProd ps :*: cfs = f d
        ta :*: cas = a (d-1)
        cs = [ \(x:xs) -> cf xs (conv cas x)
             | notTooDeep, cf <- cfs ]
        notTooDeep = d > 0 && nonEmpty ta

nonEmpty :: Type -> Bool
nonEmpty (SumOfProd ps) = not (null ps)

conv :: [[Term] -> a] -> Term -> a
conv cs (Hole p _) = hole p
conv cs (Ctr i xs) = (cs !! i) xs
```

Figure 5.2: Lazy SmallCheck's `Series` combinators.

A variant of Lazy SmallCheck's `depthCheck` function can be now be defined.

```
check :: Serial a => Int -> (a -> Bool) -> IO ()
check d p = refute (p . conv cs) (Var [] t)
  where t :*: cs = series d
```

For simplicity of presentation, these two definitions do not attempt to print counter examples, count the number of tests performed, or support checking of multi-argument properties.

5.5.6 Parallel conjunction

Parallel conjunction is a straightforward extension to the refutation algorithm. The main difference is that answers contain values of type `Property` rather than `Bool`. Internally, a `Property` is just a representation of a logical formula. To evaluate a `Property` of the form `p *&* q`, `p` is evaluated. If it is unknown, then `q` is also evaluated, *without* refining the input as demanded by `p`. If either `p` or `q` evaluates to `False` then the value of the whole conjunction is taken to be `False`. If both `p` and `q` are unknown, then the input is refined at the position demanded by `p`. This means that the number of tests generated can decrease when switching from `&&` to `*&*`, but never increase. There is however an evaluation overhead when `p` is unknown, because `*&*` will evaluate `q` in this case and `&&` will not.

5.5.7 Variations

Two alternative implementations of Lazy SmallCheck have been explored, both avoiding repeated conversion of universal terms to Haskell values of a particular type. One uses `Data.Generics` and only works in GHC, while the other requires an extra method in the `Serial` class so that refinement can be defined on a per-type basis. These variants are more efficient, but typically by no more than a factor of three. The implementation presented here has the advantage of giving depth and generation control to the programmer in a simple manner that is largely compatible with the core SmallCheck subset.

5.6 Comparative evaluation

Previous sections have included some in-principle comparisons between the three libraries. This section presents some quantitative results. Table 5.1 shows the runtimes of several example properties tested to varying depths with SmallCheck and Lazy SmallCheck. QuickCheck is not represented in this table because it does not have the same notion of a depth bound. However, the time taken by QuickCheck to refute an invalid property can be meaningfully compared with that taken by SmallCheck and Lazy SmallCheck; such timings are noted in the discussion.

All the example properties are first-order and universally-quantified. All SmallCheck generators are written using the simple standard pattern. All QuickCheck generators are written in the simple manner described in the section "Generating Recursive Data Types" of the QuickCheck manual [43]. The following paragraphs discuss the results, focusing on some of the more interesting examples.

5.6.1 RedBlack

The RedBlack program is an implementation of sets using ordered Red-Black trees by Okasaki [76], but with a fault injected. This is the same program as that described in Section 4.7.2. Recall the predicate `redBlack` defining ordered Red-Black trees

```
redBlack t = ord t && black t && red t
```

and the `insert` function which should preserve the `redBlack` invariant.

```
prop_insertRB :: Int -> Tree Int -> Bool
prop_insertRB x t =
  redBlack t ==> redBlack (insert x t)
```

No counter example was found within 20 minutes of testing at depth 4 using SmallCheck. QuickCheck, with simple random generation of trees, did not find a counter example after 100,000 batches of 1000 tests (amounting to 32 minutes of testing). Testing with Lazy SmallCheck revealed the fault in a fraction of a second at depth 4, and with the fault removed, verified the property at depth 4 within 7 seconds.

Property		**Depth**				
		3	**4**	**5**	**6**	**7**
RedBlack	L	0.03	*0.15			
	S	0.20	×			
Turner	L	0.01	0.47	×		
	S	0.01	0.07	×		
SumPuz	L	0.05	3.68	421.80	×	
	S	0.05	4.48	682.86	×	
Huffman_2	L	0	0	0.63	22.9	×
	S	0	0.01	7.65	×	
Countdown_1	L	0.01	0.14	2.27	39.3	800.4
	S	0.05	17.43	×		
Countdown_2	L	0.01	1.23	666.95	×	
	S	0.01	1.44	737.10	×	
Circuits_2	L	0	0.01	0.01	0.03	0.06
	S	0	0.01	0.52	63.80	×
Circuits_3	L	0.06	13.28	×		
	S	0.02	5.08	×		
Catch	L	0.07	6.22	830.02	×	
	S	0.02	88.23	×		
Mate	L	0	0.37	*29.87		
	S	0.06	×			
Property		**Depth**				
		7	**8**	**9**	**10**	**11**
ListSet	L	0.01	0.02	0.03	0.06	0.13
	S	0.05	0.39	4.06	694.10	×
Huffman_1	L	0.27	2.76	27.57	315.81	×
	S	0.08	0.73	7.69	90.38	×
Circuits_1	L	0.06	0.29	1.62	10.06	70.44
	S	0.04	0.20	1.21	8.38	65.88
Multiplier	L	0.16	0.24	0.28	0.38	0.44
	S	0.16	0.21	0.30	0.34	0.42
Instantiator	L	3.71	30.03	243.82	×	
	S	3.67	29.55	241.63	×	

Key: ⋆ Counter example found L Lazy SmallCheck
× Longer than 20 minutes S SmallCheck

Table 5.1: Times to check benchmark properties using SmallCheck and Lazy SmallCheck at various depths.

The number of tests is a few times lower when using parallel conjunction inside the `redBlack` invariant. However, in this case the evaluation overhead of using `*&*` is substantial and cancels the benefit of fewer tests.

5.6.2 Huffman

The Huffman program is an implementation of Huffman compression by Bird [14]. This is the same program as that described in Section 4.7.4. Recall that the two properties of interest are (1) that the decompresser (`decode`) is the inverse of the compressor (`encode`)

```
prop_decEnc cs =
    length ft > 1 ==> decode t (encode t cs) == cs
  where ft = collate cs
        t  = mkHuff ft
```

and (2) that `mkHuff` produces *optimal* Huffman trees. That is, for all binary trees t, if t is a Huffman tree then it has a *cost* no less than that produced by `mkHuff`. Also recall that a binary tree is only a Huffman tree (as determined by `isHuff`) if it contains every symbol in the source text exactly once.

```
prop_optimal cs t =
   isHuff t cs ==> cost ft t >= cost ft (mkHuff ft)
  where ft = collate cs
```

In checking `prop_decEnc`, SmallCheck was more efficient than Lazy SmallCheck by a constant factor of 3. This property is hyper-strict for most inputs. In checking `prop_optimal`, due to the condition that input trees must be Huffman trees, Lazy SmallCheck allowed testing to one level deeper within the 20 minute cut-off.

5.6.3 Mate

The Mate program solves mate-in-N chess problems. It represents a chess board as two lists, the first containing white's piece-position pairs and the second containing black's.

```
data Board  = Board [(Kind,Square)] [(Kind,Square)]
data Kind   = King   | Queen  | Rook
            | Bishop | Knight | Pawn
type Square = (Int,Int)
data Colour = Black | White
```

It includes a function `checkmate` returning whether or not a given colour is checkmated on a given board. Now consider the conjecture that for all chess boards b, if b is a *valid board* and white has only a king and a pawn, then black cannot be in checkmate.

```
prop_checkmate b@(Board ws bs) =
   ( length ws == 2
  && Pawn `elem` map fst ws
  && validBoard b
   ) ==> not (checkmate Black b)
```

A valid board is one satisfying a number of healthiness criteria, such as each side has exactly one king, kings cannot be placed on touching squares, and no two pieces can occur on the same square.

Neither SmallCheck at depth 4 after 20 minutes, nor QuickCheck with a 100,000 batches of 1000 random tests after 18 minutes, revealed a counter example. Lazy SmallCheck within 30 seconds at depth 5 produces

```
Counter example found:
Board [(King,(3,2)),(Pawn,(2,1))]
      [(Queen,(1,3)),(King,(1,2)),(Bishop,(1,1))]
```

The order of conjuncts in the property has a significant impact on performance, and a lot of experimentation was required to find the best order. The time taken to find a counterexample was more than 20 minutes if the order was unfortunately chosen. However, using parallel conjunction, no ordering required more than 22 seconds to find a counter example.

5.6.4 Reduceron instantiator

The instantiator is taken from the source code of the Reduceron, and is shown in Figure 5.3. The functions `isArg`, `isAp`, `isFun`, `isInt`, and `isEnd` all inspect the tag of a Reduceron bytecode node encoded as an 18-element bit-vector. The functions `getArg` and `getAp` strip off the tag of argument

```
instantiate :: [[Bit]] -> [Bit] -> [Bit] -> [Bit]
instantiate args base node =
  pick [ isArg node --> markEnd (isEnd node) arg
       , isAp node  --> markEnd (isEnd node) (mkApNode app)
       , (isFun node <|> isInt node) --> node
       ]
  where
    argHot = decode (getArg node)
    arg    = pick (zip argHot args)
    app    = decrement (base /+/ getAp node)
```

Figure 5.3: Template instantiation function, from the Reduceron.

and application nodes respectively, and `mkApNode` prefixes an `Ap` tag to a node. The function `markEnd` sets or resets the end-bit of a node, as specified by the first argument. The function `decode` returns a one-hot representation of a binary-encoded number. The operator `-->` simply constructs a pair from its two arguments.

The property of interest is that `instantiate` is a circuit-level equivalent of the `inst` function defined in the Reduceron semantics in Figure 2.7, but for the fact that `instantiate` works on bit-level representations of bytecode nodes. The property's pre-condition asserts that

- all arguments are non-`End` nodes,
- the node to instantiate is wrapped in at most one `End` marker (the syntax defined in Figure 2.3 permits nodes to be wrapped in multiple `End` markers),
- the base address is larger than or equal to zero,
- all nodes (the argument nodes and the node to instantiate) contain integer fields that are larger than or equal to zero and are small enough to be encoded in the 18-element bit-vector representation of a bytecode node,
- and all application nodes have an integer field strictly larger than zero.

Despite the pre-condition, Lazy SmallCheck and SmallCheck perform similarly on this property. Although the pre-condition does permit pruning, it is not restrictive enough to give Lazy SmallCheck a significant advantage.

5.6.5 Other examples

The remaining examples follow a similar pattern. SmallCheck is more effective on strict properties and Lazy SmallCheck more so on lazy ones. Of these examples, ListSet is the set implementation using ordered lists (along with the insertion property) given earlier, Countdown is a solver for a popular numbers game (along with a lemma and a refinement theorem) taken from [45], SumPuz is a cryptarithmetic solver (with a soundness property) from [18], Turner is Peyton Jones's implementation of Turner's abstraction algorithm [78] (with Turner's law of abstraction [92]), Circuits is part of a library from the Reduceron (Circuits_2 is the multiplexor example described in Section 4.7.3), Catch is a specification (with a soundness property) for part of the Catch tool [67], and Multiplier is the property defined in Section 3.5.3 capturing the correctness of a circuit-level sequential multiplier written using the Recipe library. Unfortunately, the other property defined in Chapter 3, capturing the correctness of the stack processor Poly, could not be tested as that would require simulation support for block RAMs in Lava, which is currently unavailable.

5.6.6 Summary of results

In two of the fifteen example properties, Lazy SmallCheck found a counter example in good time, and SmallCheck and QuickCheck did not. In five others, Lazy SmallCheck permitted deeper checking than SmallCheck, and in another seven, SmallCheck had a constant factor advantage over Lazy SmallCheck, ranging from a negligible factor of just over 1, to just over to a more significant factor of 7.

Five of the example properties have an implication where the condition is composed of several conjuncts, and could potentially be improved by using parallel conjunction. In two of these, parallel conjunction had no impact on the number of tests, but neither did it introduce a significant evaluation overhead. In another, the number of tests was reduced, but this was cancelled out by the evaluation overhead. And in another, parallel conjunction reduced the runtime by up to a factor of three for some conjunct orderings, but had no effect on others. In the remaining example, the use of

parallel conjunction eliminated the need to put a long series of conjuncts in a particular order for a counter example to be found.

5.7 Related work

5.7.1 Needed narrowing

Lazy SmallCheck's refutation algorithm is closely related to *needed narrowing* [2], an evaluation strategy used by some functional-logic languages, including Curry [35], and some Haskell analysers (see Chapter 4). Needed narrowing allows functions to be applied to partially-defined inputs, but this is achieved using logical variables rather than calls to `error` as in Lazy SmallCheck. As needed narrowing is designed for functional-logic programs, it also deals with non-deterministic functions.

A typical implementation of needed narrowing stores the partially evaluated result after each test, and resumes the evaluation after refining the input. Lazy SmallCheck instead evaluates the property from scratch every time an undefined part of the input is demanded. This means that needed narrowing is more efficient. For *small inputs*, it would be interesting to explore just how big (or small) this benefit is.

5.7.2 Residuation

Parallel conjunction is related to *residuation*, another evaluation strategy used by some functional-logic languages including Curry [35] and Escher [60]. Under residuation, if the value of a logical variable is demanded by some logical conjunct in the system, then that conjunct *suspends* on the variable, and another conjunct is evaluated. If evaluation of this second conjunct happens to instantiate the variable suspended on by the first, then the first conjunct is resumed.

In parallel conjunction, when evaluation of the first conjunct calls error, the second conjunct is immediately evaluated on the *same* input. If the second conjunct also calls error then the input is refined. Therefore, a parallel conjunction of the form `p *&* q` is similar to evaluating `p` by residuation

and `q` by narrowing. The end result in both cases is that if either conjunct is falsified, then so is the whole conjunction. There is no need for resumption and suspension mechanisms in Lazy SmallCheck because it evaluates the conjunction from scratch every time a refinement is made.

5.7.3 Gast

Gast [54] is a library for property-based testing in Clean. It exploits Clean's generic programming features to offer a default test-generator for all user-defined types. Like SmallCheck, it generates fully-defined and finite values. Unlike SmallCheck, it employs a blend of random and systematic generation. Constructors of an algebraic data type are selected at random, and duplicate tests are avoided by keeping a record of which inputs have been tried already.

5.7.4 EasyCheck

EasyCheck [17] is another testing library, written in the functional-logic language Curry. It can also exploit narrowing to achieve demand-driven generation of test data, making use of the data refinement and narrowing mechanisms built into Curry, although this possibility is not discussed much. EasyCheck provides a number of combinators for expressing properties about non-deterministic functions. Apart from this, the main difference is that EasyCheck uses *level diagonalisation*, which has the advantage that it allows systematic generation of deep and shallow inputs in a fair order. There are also some disadvantages of level diagonalisation: any counter examples produced are not necessarily minimal, and it is not clear to the programmer which inputs have been tested and which have not.

5.8 Limitations and future work

Currently Lazy SmallCheck is limited to checking universally-quantified, first-order properties. In future it would be interesting to try adding support for higher-order functions and existential quantifiers. A possible approach to generating functions as test data, on demand, has been discussed in Section 4.10.1. Just as Lazy SmallCheck allows pruning when a universally-

quantified expression evaluates to true on a partially-defined input, so it should also allow pruning when an existentially-quantified expression evaluates to false.

It would also be interesting to compare Lazy SmallCheck with a full-strength narrowing implementation, such as the Münster Curry Compiler [61]. This comparison would help establish whether it is worth adding narrowing to an existing Haskell compiler to aid property-based testing, or whether lazy evaluation and imprecise exceptions already provide most of the benefit.

Another avenue for investigation would be the ability to import QuickCheck, SmallCheck, and Lazy SmallCheck in a program and test the same properties using any tool.

5.9 Conclusions

If a property is refuted by SmallCheck then a *simplest* counter example is reported, and such a counter example is usually the easiest to investigate. Alternatively, if a property is not refuted then a *clearly-defined* portion of the input space on which it holds is reported, and this knowledge is valuable in judging the effectiveness of testing. In each case the SmallCheck user learns something useful that the QuickCheck user would not. Furthermore, the SmallCheck user can write data generators easily using a simple standard pattern, and can enjoy a richer specification language supporting existential quantification.

Lazy SmallCheck is a variation of SmallCheck which generates *partially-defined* inputs that are progressively refined as demanded by the property under test. Even though the programmer specifies conditional properties as simple logical implications, typically a plentiful supply of condition-satisfying inputs are generated automatically. This outcome is thanks not just to Haskell's lazy evaluation strategy, which can compute well-defined outputs for partially-defined inputs, but also to parallel conjunction. Parallel conjunction reduces the need for programmers to tweak conjunct orderings in properties in order to obtain the maximum benefit of Lazy SmallCheck. Of course, it is very difficult to say how often conditional properties occur in general, but they arose quite readily in the fifteen benchmark properties

considered here, the majority of which were taken from existing programs described in the literature. In seven of the fifteen properties, Lazy SmallCheck allowed deeper testing than SmallCheck, and in two of these, counter examples were revealed that were simply infeasible to find using QuickCheck and SmallCheck, at least without writing a custom generator.

Although SmallCheck and Lazy SmallCheck are sometimes more effective than QuickCheck, the reverse is also true. For example, as part of his ICFP'07 invited talk, Hughes tested an SMS message-packing program using QuickCheck. QuickCheck uncovered a bug when packing messages of multiple-of-eight length. Such large, strictly-demanded messages would be outside the reach of SmallCheck and Lazy SmallCheck.

Put simply: QuickCheck, SmallCheck, and Lazy SmallCheck are complementary approaches to property-based testing in Haskell.

Chapter 6

Conclusions

The first conclusion of this thesis is that lazy functional language implementations based can usefully exploit wide, parallel memories. This has been demonstrated by a theoretical analysis of the number of clock-cycles consumed by the Wide and Narrow Reducerons (Sections 2.6 and 2.7), and by an experimental comparison between FPGA prototypes of the two (Section 2.9). The FPGA prototype of the Wide Reduceron is on average five-and-a-half times faster than the narrow one across a range of benchmark programs.

The FPGA prototype of the Wide Reduceron has also been compared with existing Haskell implementations running on a PC (Section 2.9). Despite being clocked some thirty times slower than the PC, the FPGA prototype performs better than mature bytecode interpreters and within a factor of five (on non arithmetic-intensive programs) of an advanced optimising compiler. Advances in FPGA technology and further development of the Wide Reduceron should reduce this gap (Section 2.10). One attractive goal for future research is to utilise the wide, parallel memories to process many nodes of a pure functional data structure simultaneously (Section 2.10.7).

Following standard practice in circuit design, the FPGA prototypes of the Wide and Narrow Reducerons have been described using a combination of structural and behavioural description. However, this has been done using a hardware description language without built-in support for behavioural description. This thesis has shown that behavioural description can actually be done conveniently in a *pure structural language* with the help of a small

library of higher-order, pure structural components (Chapter 3). Although the library is simple and useful (Section 3.10), it needs to be improved with respect to producing helpful error messages and obtaining understandable feedback from low-level synthesis tools (Section 3.8).

The second conclusion of this thesis is that *needed narrowing* facilitates automatic property-based testing. This is supported by the observed ability of Forward Reach, which is based on needed narrowing, to often generate target-reaching inputs much more rapidly than Basic Reach, which is based on exhaustive testing (Section 4.8). It is also supported by the observation that Forward Reach can sometimes find counter-examples to program properties which cannot be found feasibly using automatic random testing (Section 4.8). The benefit of needed narrowing is most apparent when the source function is lazy, and this is often the case for program properties with restrictive antecedents – a common pattern in property-based testing.

Although Forward and Backward Reach are both based on needed narrowing, they often operate quite differently. One of the main differences is the order in which they evaluate expressions, and this can have a big impact on the sizes of the search spaces explored. Despite observing some advantages and disadvantages of Forward and Backward Reach (Section 4.8), this thesis is unable to make a general recommendation of one over the other; sometimes the evaluation order introduced by Forward Reach is better than that introduced by Backward Reach, and sometimes the reverse is true. A promising avenue for future work is to adapt Backward Reach to solve multiple constraints simultaneously using Lindblad's parallel conjunction technique (Section 4.10.4). This has the potential to yield a version of Reach which is always better than any of the existing versions.

The final conclusion of this thesis is that needed narrowing can be usefully simulated using a standard lazy evaluator. As a result, the ability of needed narrowing to effectively test program properties can be captured in a *library* for a standard lazy functional language. This has been demonstrated in the development of Lazy SmallCheck (Section 5.4). Compared to Reach, Lazy SmallCheck is much simpler to implement. Compared to the existing testing libraries QuickCheck and SmallCheck, Lazy SmallCheck can avoid the need to write custom generators in order to produce a good distribution of relevant test data in a short amount of time (Section 5.9).

Appendix A

Implementation of Recipe

```
module Recipe
  ( Bit       -- type Bit = Signal Bool
  , Recipe    -- instance Monad Recipe
  , Var       -- type Var
  , val       -- :: Var -> [Bit]
  , newSig    -- :: Int -> Recipe Var
  , newReg    -- :: Int -> Recipe Var
  , skip      -- :: Recipe ()
  , tick      -- :: Recipe ()
  , tickN     -- :: Int -> Recipe ()
  , (<==)     -- :: Var -> [Bit] -> Recipe ()
  , (<=|)     -- :: Var -> [Bit] -> Recipe ()
  , cond      -- :: Bit -> Recipe () -> Recipe () -> Recipe ()
  , (|>)      -- :: Bit -> Recipe () -> Recipe ()
  , while     -- :: Bit -> Recipe () -> Recipe ()
  , doUntil   -- :: Bit -> Recipe () -> Recipe ()
  , forever   -- :: Recipe () -> Recipe ()
  , waitWhile -- :: Bit -> Recipe ()
  , waitUntil -- :: Bit -> Recipe ()
  , par       -- :: [Recipe ()] -> Recipe ()
  , follow    -- :: Bit -> Recipe a -> (Bit, a)
  ) where

import Lava
import List

-- A shorter name for Lava's boolean signals
type Bit = Signal Bool

-- Auxiliary functions
tree :: (a -> a -> a) -> [a] -> a
```

```
tree f [x] = x
tree f (x:y:ys) = tree f (ys ++ [f x y])

pick :: [(Bit, [Bit])] -> [Bit]
pick xs = map (tree (<|>)) (transpose ys)
  where ys = [map (b <&>) x | (b, x) <- xs]

mux2 :: Bit -> Bit -> Bit -> Bit
mux2 sel a b = (inv sel <&> a) <|> (sel <&> b)

row :: ((a, b) -> (c, a)) -> (a, [b]) -> ([c], a)
row circ (carryIn, []) = ([], carryIn)
row circ (carryIn, x:xs) = (y:ys, carryOut)
  where (y, carry) = circ (carryIn, x)
        (ys, carryOut) = row circ (carry, xs)

delayEn :: Bit -> Bit -> Bit -> Bit
delayEn init en inp = out
  where out = delay init (mux2 (inv en) inp out)

regEn :: [Bit] -> Bit -> [Bit] -> [Bit]
regEn [] en inps = []
regEn (x:xs) en inps = delayEn x en y : regEn xs en ys
  where y:ys = inps

setReset :: Bit -> Bit -> Bit -> Bit
setReset init s r = out
  where q = delay init (out <&> inv r)
        out = s <|> q

-- State monad
data State s a = State (s -> (s, a))

run :: State s a -> s -> (s, a)
run (State f) = f

instance Monad (State s) where
  return a = State (\s -> (s, a))
  c >>= f  = State (\s -> case run c s of
                            (s', a) -> run (f a) s')

-- Recipe monad
type Recipe a = State (Bit, Env) a

skip :: Recipe ()
skip = return ()

tick :: Recipe ()
tick = State (\(start, env) -> ((delay low start, env), ()))
```

```
tickN :: Int -> Recipe ()
tickN 0 = skip
tickN n = tick >> tickN (n-1)

cond :: Bit -> Recipe () -> Recipe () -> Recipe ()
cond cond p q = State (\(start, env) ->
  let ((fin0, env0), _) = run p (start <&> cond, env)
      ((fin1, env1), _) = run q (start <&> inv cond, env0)
  in  ((fin0 <|> fin1, env1), ()))

infixr 4 |>
(|>) :: Bit -> Recipe () -> Recipe ()
guard |> p = cond guard p skip

while :: Bit -> Recipe () -> Recipe ()
while cond p = State (\(start, env) ->
  let ((fin, env'), a) = run p (cond <&> ready, env)
      ready = start <|> fin
  in  ((inv cond <&> ready, env'), a))

doUntil :: Bit -> Recipe () -> Recipe ()
doUntil cond p = State (\(start, env) ->
  let ((fin, env'), b) = run p (ready, env)
      ready = start <|> (fin <&> inv cond)
  in  ((fin <&> cond, env'), b))

forever :: Recipe () -> Recipe ()
forever p = while high p

waitWhile :: Bit -> Recipe ()
waitWhile cond = while cond tick

waitUntil :: Bit -> Recipe ()
waitUntil cond = while (inv cond) tick

par :: [Recipe ()] -> Recipe ()
par ps = State (\(start, env) ->
  let (fins, env') = row circ (env, ps)
      circ (env, p) = fst (run p (start, env))
      fin = tree (<&>) (map (\s -> setReset low s fin) fins)
  in  ((fin, env'), ()))

-- Environment
type VarId = Int

type Input = (Bit, [Bit])

data Env = Env { freshId  :: VarId
```

```
              , readEnv  :: [(VarId, Input)]
              , writeEnv :: [(VarId, Input)] }

-- Mutable variables
data Var = Var { varId :: VarId, val :: [Bit] }

newVar :: ([Input] -> [Bit]) -> Recipe Var
newVar f = State (\(start, env) ->
  let v    = freshId env
      inps = [i | (w, i) <- readEnv env, v == w]
  in  ((start, env { freshId = v+1 }), Var v (f inps)))

newReg :: Int -> Recipe Var
newReg width = newVar reg
  where reg inps = regEn (replicate width low) en (pick inps)
          where en = tree (<|>) (map fst inps)

newSig :: Int -> Recipe Var
newSig width = newVar pick

-- Assignment
infix 5 <==
(<==) :: Var -> [Bit] -> Recipe ()
r <== x = State (\(start, env) ->
  let wenv = (varId r, (start, x)) : writeEnv env
  in  ((start, env { writeEnv = wenv }), ()))

infix 5 <=|
(<=|) :: Var -> [Bit] -> Recipe ()
r <=| x = (r <== x) >> tick

-- Top-level
follow :: Bit -> Recipe a -> (Bit, a)
follow start r = (fin, a)
  where ((fin, env), a) = run r (start, initialEnv)
        initialEnv      = Env 0 (writeEnv env) []
```

Appendix B

Implementation of Lazy SmallCheck

```
module LazySmallCheck
  ( Serial(..) -- :: class
  , Series     -- :: type Series a = Int -> Cons a
  , Cons       -- :: *
  , cons       -- :: a -> Series a
  , (><)       -- :: Series (a -> b) -> Series a -> Series b
  , empty      -- :: Series a
  , (\/)       -- :: Series a -> Series a -> Series a
  , drawnFrom  -- :: [a] -> Cons a
  , cons0      -- :: a -> Series a
  , cons1      -- :: Serial a => (a -> b) -> Series b
  , cons2      -- :: (Serial a, Serial b) =>
               --      (a -> b -> c) -> Series c
  , cons3      -- :: ...
  , cons4      -- :: ...
  , Testable   -- :: class
  , depthCheck -- :: Testable a => Int -> a -> IO ()
  , smallCheck -- :: Testable a => Int -> a -> IO ()
  , (==>)      -- :: Bool -> Bool -> Bool
  , Property   -- :: *
  , lift       -- :: Bool -> Property
  , neg        -- :: Property -> Property
  , (*&*)      -- :: Property -> Property -> Property
  , (*|*)      -- :: Property -> Property -> Property
  , (*=>*)     -- :: Property -> Property -> Property
  , (*=*)      -- :: Property -> Property -> Property
  ) where
```

```
import Control.Monad
import Control.Exception
import System.Exit

infixr 0 ==>, *=>*
infixr 3 \/, *|*
infixl 4 ><, *&*

type Pos      = [Int]
data Term     = Var Pos Type | Ctr Int [Term]
data Type     = SumOfProd [[Type]]
type Series a = Int -> Cons a
data Cons a   = Type :*: ([[Term] -> a])

class Serial a where
  series :: Series a

-- Series combinators
cons :: a -> Series a
cons a d = SumOfProd [[]] :*: [const a]

empty :: Series a
empty d = SumOfProd [] :*: []

(><) :: Series (a -> b) -> Series a -> Series b
(f >< a) d = SumOfProd [ta:p | shallow, p <- ps] :*: cs
  where
    SumOfProd ps :*: cfs = f d
    ta :*: cas = a (d-1)
    cs = [\(x:xs) -> cf xs (conv cas x) | shallow, cf <- cfs]
    shallow = d > 0 && nonEmpty ta

nonEmpty :: Type -> Bool
nonEmpty (SumOfProd ps) = not (null ps)

(\/) :: Series a -> Series a -> Series a
(a \/ b) d = SumOfProd (ssa ++ ssb) :*: (ca ++ cb)
  where
    SumOfProd ssa :*: ca = a d
    SumOfProd ssb :*: cb = b d

conv :: [[Term] -> a] -> Term -> a
conv cs (Var p _) = error ('\0':map toEnum p)
conv cs (Ctr i xs) = (cs !! i) xs

drawnFrom :: [a] -> Cons a
drawnFrom xs = SumOfProd (map (const []) xs) :*: map const xs
```

```
-- Helpers, a la SmallCheck
cons0 f = cons f
cons1 f = cons f >< series
cons2 f = cons f >< series >< series
cons3 f = cons f >< series >< series >< series
cons4 f = cons f >< series >< series >< series >< series


-- Standard instances
instance Serial () where
  series = cons0 ()

instance Serial Bool where
  series = cons0 False \/ cons0 True

instance Serial a => Serial (Maybe a) where
  series = cons0 Nothing \/ cons1 Just

instance (Serial a, Serial b) => Serial (Either a b) where
  series = cons1 Left \/ cons1 Right

instance Serial a => Serial [a] where
  series = cons0 [] \/ cons2 (:)

instance (Serial a, Serial b) => Serial (a, b) where
  series = cons2 (,) . (+1)

instance (Serial a, Serial b, Serial c) =>
    Serial (a, b, c) where
  series = cons3 (,,) . (+1)

instance (Serial a, Serial b, Serial c, Serial d) =>
    Serial (a, b, c, d) where
  series = cons4 (,,,) . (+1)

instance Serial Int where
  series d = drawnFrom [-d..d]

instance Serial Integer where
  series d = drawnFrom (map toInteger [-d..d])

instance Serial Char where
  series d = drawnFrom (take (d+1) ['a'..])

instance Serial Float where
  series d = drawnFrom (floats d)

instance Serial Double where
  series d = drawnFrom (floats d)
```

```
floats :: RealFloat a => Int -> [a]
floats d = [ encodeFloat sig exp
           | sig <- map toInteger [-d..d]
           , exp <- [-d..d]
           , odd sig || sig == 0 && exp == 0
           ]

-- Term refinement
refine :: Term -> Pos -> [Term]
refine (Var p (SumOfProd ss)) [] = new p ss
refine (Ctr c xs) p = map (Ctr c) (refineList xs p)

refineList :: [Term] -> Pos -> [[Term]]
refineList xs (i:is) = [ls ++ y:rs | y <- refine x is]
  where (ls, x:rs) = splitAt i xs

new :: Pos -> [[Type]] -> [Term]
new p ps = [ Ctr c (zipWith (\i t -> Var (p++[i]) t) [0..] ts)
           | (c, ts) <- zip [0..] ps ]

-- Find total instantiations of a partial value
total :: Term -> [Term]
total val = tot val where
  tot (Ctr c xs) = [Ctr c ys | ys <- mapM tot xs]
  tot (Var p (SumOfProd ss)) = [y | x <- new p ss, y <- tot x]

-- Answers
answer :: a -> (a -> IO b) -> (Pos -> IO b) -> IO b
answer a known unknown =
  do res <- try (evaluate a)
     case res of
       Right b -> known b
       Left (ErrorCall ('\0':p)) -> unknown (map fromEnum p)
       Left e -> throw e

-- Refute
refute :: Result -> IO Int
refute r = ref (args r)
  where
    ref xs = eval (apply r xs) known unknown
      where
        known True = return 1
        known False = report
        unknown p = sumMapM ref 1 (refineList xs p)

        report =
          do putStrLn "Counter example found:"
             mapM_ putStrLn $ zipWith ($) (showArgs r)
```

```
                            $ head [ys | ys <- mapM total xs]
            exitWith ExitSuccess

sumMapM :: (a -> IO Int) -> Int -> [a] -> IO Int
sumMapM f n [] = return n
sumMapM f n (a:as) = seq n (do m <- f a ; sumMapM f (n+m) as)

-- Properties with parallel conjunction
data Property =
    Bool   Bool
  | Neg    Property
  | And    Property Property
  | ParAnd Property Property
  | Eq     Property Property

eval :: Property -> (Bool -> IO a) -> (Pos -> IO a) -> IO a
eval p k u = answer p (\p -> eval' p k u) u

eval' (Bool b)     k u = answer b k u
eval' (Neg p)      k u = eval p (k . not) u
eval' (And p q)    k u =
  eval p (\b-> if b then eval q k u else k b) u
eval' (Eq p q)     k u =
  eval p (\b-> if b then eval q k u else eval (Neg q) k u) u
eval' (ParAnd p q) k u =
    eval p (\b-> if b then eval q k u else k b) unknown
  where
    unknown pos = eval q (\b-> if b then u pos else k b)
                         (\_-> u pos)

lift b   = Bool b
neg p    = Neg p
p *&* q  = ParAnd p q
p *|* q  = neg (neg p *&* neg q)
p *=>* q = neg (p *&* neg q)
p *=* q  = Eq p q

-- Boolean implication
(==>) :: Bool -> Bool -> Bool
False ==> _ = True
True ==> x = x

-- Testable properties
data Result =
  Result { args     :: [Term]
         , showArgs :: [Term -> String]
         , apply    :: [Term] -> Property
         }
```

```
data P = P (Int -> Int -> Result)

run :: Testable a => ([Term] -> a) -> Int -> Int -> Result
run a = f where P f = property a

class Testable a where
  property :: ([Term] -> a) -> P

instance Testable Bool where
  property apply = P $ \n d ->
    Result [] [] (Bool . apply . reverse)

instance Testable Property where
  property apply = P $ \n d -> Result [] [] (apply . reverse)

instance (Show a, Serial a, Testable b) =>
    Testable (a -> b) where
  property f = P $ \n d ->
    let t :*: c = series d
        c' = conv c
        r = run (\(x:xs) -> f xs (c' x)) (n+1) d
    in  r { args = Var [n] t : args r
          , showArgs = (show . c') : showArgs r }

-- Top-level interface
depthCheck :: Testable a => Int -> a -> IO ()
depthCheck d p =
  do n <- refute $ run (const p) 0 d
     putStrLn $ "OK, required "    ++ show n ++
                " tests at depth " ++ show d ++ "."

smallCheck :: Testable a => Int -> a -> IO ()
smallCheck d p = mapM_ (`depthCheck` p) [0..d]
```

Bibliography

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. Fortress Language Specification. `http://projectfortress.sun.com/Projects/Community/`.

[2] Sergio Antoy, Rachid Echahed, and Michael Hanus. A Needed Narrowing Strategy. In *POPL '94: Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 268–279. ACM, 1994.

[3] Sergio Antoy and Andrew P. Tolmach. Typed Higher-Order Narrowing without Higher-Order Strategies. In *FLOPS '99: 4th Fuji International Symposium on Functional and Logic Programming*, pages 335–353. Springer, 1999.

[4] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 5th ACM SIGPLAN Erlang Workshop*. ACM, 2006.

[5] Peter Ashenden. Recursive and Repetitive Hardware Models in VHDL. Technical Report 93-19, Department of Computer Science, University of Adelaide, 1993.

[6] Peter Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., 1999.

[7] Lennart Augustsson. Compiling lazy functional languages, part II. PhD Thesis, Chalmers University of Technology, 1987.

[8] Lennart Augustsson. BWM: A Concrete Machine for Graph Reduction. In *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, pages 36–50. Springer, 1992.

[9] Lennart Augustsson. Overloaded booleans. `http://augustss.blogspot.com/`, 2007.

[10] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[11] Darius Bacon. Clickcheck and Peckcheck: libraries for automatic specification-based testing. `http://www.accesscom.com/~darius/software/clickcheck.html`.

[12] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[13] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[14] Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

[15] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-machine. In *LFP '88: Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 244–258. ACM, 1988.

[16] Celoxica. Handel-C Language Reference Manual. `http://www.celoxica.com/`.

[17] Jan Christiansen and Sebastian Fischer. EasyCheck – Test Data for Free. In *FLOPS '08: Proceedings of the 9th International Symposium on Functional and Logic Programming*. Springer, LNCS 4989, 2008.

[18] K. Claessen, C. Runciman, O. Chitil, R. J. M. Hughes, and M. Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *Advanced Functional Programming (AFP'02)*, pages 59–99. Springer, LNCS 2638 , 2002.

[19] Koen Claessen. Embedded Languages for Describing and Verifying Hardware. PhD Thesis, Chalmers University of Technology, 2001.

[20] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs . In *Proceedings of Inter-*

national Conference on Functional Programming (ICFP). ACM SIGPLAN, 2000.

[21] Koen Claessen and Gordon Pace. An Embedded Language Framework for Hardware Compilation. In *Participants Proceedings of the 2002 Workshop on Designing Correct Circuits*, 2002.

[22] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, 1994.

[23] Brendan Eich. Javascript at ten years. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, page 129. ACM, 2005.

[24] Don Syme et al. The F# Draft Language Specification. `http://research.microsoft.com/fsharp/`.

[25] Kathleen Fisher et. al. CUFP: Commercial Users of Functional Programming. `http://cufp.galois.com/`.

[26] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, 1969.

[27] Robby Findler, Michael Hanus, and Simon Thompson, editors. *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education.*

[28] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM, 2002.

[29] Simon Frankau. Hardware Synthesis from a Stream-Processing Functional Language. PhD Thesis, University of Cambridge, 2004.

[30] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[31] Andy Gill and Colin Runciman. Haskell Program Coverage. In *Haskell '07: Proceedings of the ACM SIGPLAN 2007 Haskell Workshop*, pages 1–12. ACM, 2007.

[32] Dimitry Golubovsky, Neil Mitchell, and Matthew Naylor. Yhc.Core - from Haskell to Core. *The Monad.Reader*, (7):45–61, April 2007.

[33] Brian Grattan, Greg Stitt, and Frank Vahid. Codesign-extended applications. In *CODES '02: Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 1–6. ACM, 2002.

[34] Sumit Gupta, Rajesh Kumar Gupta, Nikil D. Dutt, and Alexandru Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):441–470, 2004.

[35] Michael Hanus. Curry: An Integrated Functional Logic Language. Language report, available online at `http://curry-language.org/`, March 2006.

[36] Michael Hanus and Christian Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[37] John Harrison. The HOL Light theorem prover. `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[38] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language (3rd Edition)*. Addison-Wesley, 2008.

[39] John L. Hennessy and David A. Patterson. *Computer Architecture; A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1992.

[40] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *FLOPS '06: Proceedings of the 8th International Symposium on Functional and Logic Programming*, pages 208–225, 2006.

[41] Ralf Hinze and Norman Ramsey, editors. *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany*. ACM, 2007.

[42] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 12–1–12–55. ACM, 2007.

[43] John Hughes. QuickCheck: An Automatic Testing Tool for Haskell. `http://www.cs.chalmers.se/~rjmh/QuickCheck/manual.html`.

[44] John Hughes. The Design and Implementation of Programming Languages. PhD Thesis, University of Oxford, 1983.

[45] Graham Hutton. The Countdown Problem. *Journal of Functional Programming*, 12(6):609–616, November 2002.

[46] Daisuke Ikegami. RushCheck: a lightweight random testing tool for Ruby. `http://rushcheck.rubyforge.org/`.

[47] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, 2006.

[48] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient Interpretation by Transforming Data Types and Patterns to Functions. In *Trends in Functional Programming*, volume 7. Intellect, 2007.

[49] Thomas Johnsson. Compiling lazy functional languages. PhD Thesis, Chalmers University of Technology, 1987.

[50] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 25–36. ACM, 1999.

[51] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-Based Testing of Java Programs Using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.

[52] Oleg Kiselyov. Number-parameterized types. *The Monad.Reader*, Issue 5, October 2005.

[53] Philip Koopman. *An Architecture for Combinator Graph Reduction*. Academic Press, 1990.

[54] Pieter W. M. Koopman, Artem Alimarine, Jan Tretmans, and Marinus J. Plasmeijer. Gast: Generic automated software testing. In *IFL'02*, pages 84–100. Springer, LNCS 2670, 2002.

[55] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[56] James Larus and Christos Kozyrakis. Transactional memory. *Communications of the ACM*, 51(7):80–88, 2008.

[57] John Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 144–154. ACM, 1993.

[58] Christopher League. QCheck/SML: a library for automatic unit testing of Standard ML modules. `http://contrapunctus.net/league/haques/qcheck/`.

[59] Fredrik Lindblad. Property directed generation of first-order test data. In *TFP '07: Revised Selected Papers from the 8th Symposium on Trends in Functional Programming, TFP 2007*, volume 8, pages 105–123. Intellect, 2008.

[60] John W. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, 1999(3), 1999.

[61] Wolfgang Lux. The Münster Curry Compiler. `http://danae.uni-muenster.de/~lux/curry/`, 2003.

[62] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional programming*, pages 277–288. ACM, 2007.

[63] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[64] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall Internaional, 1992.

[65] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc., 1994.

[66] Neil Mitchell. Transformation and Analysis of Functional Programs. PhD Thesis, University of York, 2008.

[67] Neil Mitchell and Colin Runciman. A Static Checker for Safe Pattern Matching in Haskell. In *TFP'05*, volume 6, pages 15–30. Intellect, 2007.

[68] Tom Moertel. LectroTest. `http://search.cpan.org/~tmoertel/Test-LectroTest-0.3600/`, 2007.

[69] Tony Morris. Reductio: automated specification-based testing for java. `http://reductiotest.org/`.

[70] Alan Mycroft and Richard Sharp. A Statically Allocated Parallel Functional Language. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 37–48. Springer, 2000.

[71] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., 2006.

[72] H. Riis Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[73] Rickard Nilsson. ScalaCheck: a powerful tool for automatic unit testing. `http://code.google.com/p/scalacheck/`.

[74] Martin Odersky. The Scala Language Specification. `http://www.scala-lang.org/`.

[75] University of York. PRESENCE-3 Development. `http://www.cs.york.ac.uk/arch/neural/hardware/presence-3/`.

[76] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

[77] Ian Page and Wayne Luk. Compiling Occam into field-programmable gate arrays. In *Oxford Workshop on Field Programmable Logic and Applications*, pages 271–283. Abingdon EE&CS Books, 1991.

[78] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[79] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[80] Simon Peyton Jones and David Lester. *Implementing Functional Langauges: A Tutorial*. Prentice-Hall, 1992.

[81] John Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[82] Colin Runciman and Mike Firth. Formalised development of software by machine assisted transformation. *SIGSOFT Software Engineering Notes*, 15(4):115–117, 1990.

[83] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: exhaustive testing for small values. In *Haskell'08: Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM, 2008.

[84] K. Rustan, M. Leino, and Greg Nelson. An extended static checker for Modula-3. In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 302–305. Springer, 1998.

[85] Mark Scheevel. NORMA: a graph reduction processor. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 212–219. ACM, 1986.

[86] Richard Sharp. Higher-Level Hardware Synthesis. PhD Thesis, University of Cambridge, 2002.

[87] Mary Sheeran. Hardware Design and Functional Programming: a Perfect Match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

[88] William Stoye. The Implementation of Functional Languages using Custom Hardware. PhD Thesis, University of Cambridge, 1985.

[89] Donald E. Thomas and Philip R. Moorby. *The Verilog hardware description language*. Kluwer Academic Publishers, 1998.

[90] Tom Shackell et al. The York Haskell Compiler. Available online at `http://www.haskell.org/haskellwiki/Yhc`, February 2007.

[91] Mark Anders Tullsen. Path, a program transformation system for Haskell. Technical report, Yale University, 2002.

[92] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software – Practice and Experience*, 9(1):31–49, 1979.

[93] Philip Wadler. How to replace failure by a list of successes. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

[94] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.

[95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52. Springer, 1995.

[96] John Wakerly. *Digital Design: Principles and Practices*. Prentice-Hall, fourth edition, 2006.

[97] Michael Ward. Supercombinator Soft Machines. 4^{th} year project, University of York, Department of Computer Science, June 2000.

[98] Dana N. Xu. Extended static checking for Haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59. ACM, 2006.