Improving Implicit Parallelism

José Manuel Calderón Trilla

Computer Science
University of York

Doctor of Philosophy

September 2015

ABSTRACT

We propose a new technique for exploiting the inherent parallelism in lazy functional programs. Known as *implicit parallelism*, the goal of writing a sequential program and having the compiler improve its performance by determining what can be executed in parallel has been studied for many years. Our technique abandons the idea that a compiler should accomplish this feat in 'one shot' with static analysis and instead allows the compiler to *improve* upon the static analysis using iterative feedback.

We demonstrate that iterative feedback can be relatively simple when the source language is a lazy purely functional programming language. We present three main contributions to the field: the automatic derivation of parallel strategies from a demand on a structure, and two new methods of feedback-directed auto-parallelisation. The first method treats the runtime of the program as a *black box* and uses the 'wall-clock' time as a fitness function to guide a heuristic search on bitstrings representing the parallel setting of the program. The second feedback approach is *profile directed*. This allows the compiler to use profile data that is gathered by the runtime system as the program executes. This allows the compiler to determine which threads are not worth the overhead of creating them.

Our results show that the use of feedback-directed compilation can be a good source of refinement for the static analysis techniques that struggle to account for the cost of a computation. This lifts the burden of 'is this parallelism worthwhile?' away from the static phase of compilation and to the runtime, which is better equipped to answer the question.

TABLE OF CONTENTS

Abstract ii

Table of Contents iii

List of Figures vi

List of Tables viii

Acknowledgements ix

Declaration x

1 INTRODUCTION 1
- 1.1 Lazy Languages and Parallelism 2
- 1.2 Goals and Contributions of this Thesis 4
- 1.3 Thesis Roadmap 4

I The Idea 5

2 PARALLELISM IN A FUNCTIONAL LANGUAGE 7
- 2.1 A Short History of Graph Reduction 7
- 2.2 Functional Programming and Parallelism 9
- 2.3 Sequential Reduction Machines 10
- 2.4 Parallel Reduction Machines 15
- 2.5 Approaches to Parallelism 16
- 2.6 Summary of the Chapter 29

3 BIRD'S EYE VIEW OF OUR TECHNIQUE 31
- 3.1 F-lite: a Lazy Purely Functional Core Language 31
- 3.2 Overview of the Compiler 38
- 3.3 Summary 45

II The Discovery and Placement of Safe Parallelism 46

4 FINDING SAFE PARALLELISM 47
- 4.1 Original Motivation vs. Our Motivation 49
- 4.2 Overview 49

4.3 Two-Point Forward Analysis 53
4.4 Four-Point Forward Analysis 61
4.5 Projection-Based Analysis 70
4.6 Summary 81

5 DERIVATION AND USE OF PARALLEL STRATEGIES 82
5.1 Expressing Need, Strategically 83
5.2 Deriving Strategies from Projections 85
5.3 Using Derived Strategies 89

III Experimental Platform, Benchmark Programs, and Results 93

6 EXPERIMENTAL PLATFORM 94
6.1 Defunctionalisation (Higher-Order Specialisation) . . . 95
6.2 Keeping Track of all the Threads 98
6.3 Trying for par: Switching off Parallelism 100
6.4 Benchmark Programs 101

7 RUN-TIME DIRECTED SEARCH 103
7.1 Heuristic Algorithms 103
7.2 Research Questions 107
7.3 Experimental Setup and Results 108
7.4 Summary of Bitstring Searching 114

8 PROFILE DIRECTED SEARCH 115
8.1 par-Site Health 116
8.2 Search Algorithm 117
8.3 Experimental Results and Discussion 118
8.4 Transfer to GHC 124
8.5 Limitations 125
8.6 Summary of Profile-Directed Search 127

IV Conclusions and Future Directions 130

9 CONCLUSIONS 131

10 FUTURE DIRECTIONS 134
10.1 Specialising on Depth 135
10.2 Hybrid Approaches 137
10.3 Automating Pipeline Parallelism 137

Appendices 139

A BENCHMARK PROGRAMS 139

A.1 SumEuler 139

A.2 MatMul 140

A.3 Queens 141

A.4 Queens2 142

A.5 SodaCount 144

A.6 Tak 147

A.7 Taut 148

B LAZIFY EXPRESSIONS 151

LIST OF FIGURES

Figure 1 Eager and Lazy evaluation order for squaring a value. 10

Figure 2 G-Code execution example 11

Figure 3 Simple parallel graph reduction model 15

Figure 4 Call-by-value reduction 33

Figure 5 Call-by-name reduction 33

Figure 6 Call-by-need reduction 34

Figure 7 Normal order reduction 34

Figure 8 Eager (left) and Lazy (right) evaluation order for const 35

Figure 9 Abstract Syntax for F-lite 37

Figure 10 Source listing for Tak 39

Figure 11 Source listing for Tak after analysis, transformation, and par placement 39

Figure 12 Semantics of seq and par. 51

Figure 13 Flat Domain 55

Figure 14 Two-point Domain 55

Figure 15 The *meet* ($\sqcap$) and *join* ($\sqcup$) for our lattice 56

Figure 16 An Abstract Semantics for Strictness Analysis on a Two-Point Domain 57

Figure 17 Domain for pairs of flat-domain values 62

Figure 18 Wadler's Four-point Domain 64

Figure 19 Definition of $cons^{\#}$ for a Four-Point Domain . 64

Figure 20 Modification to $\mathcal{A}$ for List Constructors 65

Figure 21 Abstraction of Case Expressions on Lists . . . 67

Figure 22 Abstract Syntax for Contexts of Demand . . . 73

Figure 23 Projections for the 4-point Domain 75

Figure 24 Conjunction and Disjunction for Projections 1 76

Figure 25 Conjunction and Disjunction for Projections 2 77

Figure 26 Conjunction and Disjunction for Projections 3 78

Figure 27 Projection-based Strictness Analysis 80

Figure 28 Using Evaluation Transformers 85

Figure 29 Rules to generate strategies from demand contexts 86

Figure 30 Defunctionalisation Rules 99

Figure 31 Speedups against number of fitness evaluations for Queens2 and SodaCount 113
Figure 32 Statistics on the number of reductions carried out by the threads a par site sparks off 116
Figure 33 SumEuler speedup 122
Figure 34 Queens speedup 122
Figure 35 Queens2 speedup 123
Figure 36 Taut speedup 123
Figure 37 A tree representation of a recursive computation135
Figure 38 Lazify an F-lite Expression 151

LIST OF TABLES

Table 1 Performance of Tak with different par settings 44

Table 2 Analysis of $length^{\#}$ and $sum^{\#}$ Using 4-point Domains 68

Table 3 Analysis of $append^{\#}$ $xs^{\#}$ $ys^{\#}$ Using 4-point Domains 69

Table 4 The Evaluation Transformer for $append$ 84

Table 5 Results from using heuristic search 111

Table 6 Profile-Directed Search with an overhead of 10 reductions 120

Table 7 Profile-Directed Search with overheads of 100 reductions 120

Table 8 Profile-Directed Search with an overhead of 1000 reductions 120

Table 9 Naive transfer to GHC 124

ACKNOWLEDGEMENTS

Less than a year ago I was going to quit my PhD. I had struggled throughout the process and had decided that I was not cut out for it. It was not the first time I almost quit, and it was not the first time that Prof. Runciman convinced me to press on and get my work out into the world. I cannot imagine a better supervisor. I feel grateful every day that he thought I was worth keeping around.

During my six years in the UK my partner, Danielle, was deported. It is a testament to her level of support for me that she went through the process of *coming back* to the country that deported her so that we would not have to live apart. It was not easy, but it has been worth it.

My parents have never stopped me from following my curiosity. You would think that after asking to go to music school and then switching to CS for grad school they'd make me pick something and stick with it. I am so thankful to have the supportive and understanding parents that I have.

Layla and Mike, for having the cutest baby. I can't wait to spend more time with you three.

The members of York's PLASMA group have been a constant source of joy and knowledge. I owe them many thanks. Matt Naylor, who patiently answered all my silly questions during my first year. Jason Reich, who was always enthusiastic about everyone's work. Chris Bak for empathising with my PhD progress and playing board games with me. Glyn Faulkner for getting me excited about different languages and technologies. And Rudy and Michael for humoring me while I pitch them my half-baked language ideas.

Simon Poulding provided tons of insight into heuristic search and stats. Mike Dodds for keeping his door open to me.

Ian, Gary, Jamie, and Gareth always made me feel welcome when I would intrude on RTS breaks. Russell for understanding that sometimes I just want to watch TV and vent. Jimmy for always hearing me out and his steadfast confidence in me. Alan for his joke. Sam and Dani for always being such gracious hosts.

Rich, Chris, and Tara for helping me feel less homesick.

Danielle again, for putting up with who I am but always helping me be better.

DECLARATION

I, José Manuel Calderón Trilla, declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as references.

Earlier versions of parts of this thesis were published in the following papers:

1. José Manuel Calderón Trilla and Colin Runciman: Improving Implicit Parallelism in *Haskell '15: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pg 153-164, ACM, 2015

2. José Manuel Calderón Trilla, Simon Poulding, and Colin Runciman: Weaving Parallel Threads in *SSBSE '15: Search-Based Software Engineering*, pg 62-76, Springer, 2015

The first paper was conceived, implemented, and written by myself with significant guidance and input from Prof. Runciman.

The second paper's central idea was my own. Prof. Runciman, Simon Poulding, and myself developed the 'plan of attack'. I then implemented the experimental platform and ran the experiments. Simon Poulding performed the statistical analysis of the results. The paper itself was written jointly by Simon Poulding and myself.

1

INTRODUCTION

> *The always dubious "feed in an arbitrary program and watch it run faster" story is comprehensively dead.*
>
> – Peyton Jones [1999]

There is a very common refrain that programmers use. It goes something like this: "If you want a [X] program, you must [Y]". We can choose appropriate X's and Y's to prove a point about the difficulty of programming. Here are some common examples:

- "If you want a fast program, you must write it in C"
- "If you want an efficient program, you must write it in assembler"
- "If you want a performant program, you must use cache-conscious structures"
- "If you want a parallel program, you must write a parallel program"

This thesis is concerned with the last of these examples. What does it mean? For many, the idea that a compiler can *automatically* parallelise a program that was written in a sequential manner is a pipe-dream. This version of the refrain attempts to emphasise the point that utilising parallelism requires *active thought and action* by the programmer, we can not get parallelism 'for free'.

We seek to show that this is not always the case. We do this by attempting the inverse of the refrain: writing a compiler that is able to take a *sequential* program and transform it into a better performing[1] *parallel* program. A system that can achieve this goal is said to take advantage of a program's *implicit*, or *inherent*, parallelism.

Implicit Parallelism

The potential parallelism that is present in a program without

[1] This is key!

> the need for any annotations, calling of parallel functions, or use of parallel libraries.

It is worth noting that this is no longer the standard meaning [Belikov et al., 2013]. Increasingly, researchers take implicit parallelism to mean *semi*-implicit parallelism. Under their meaning implicit parallelism often refers to techniques that provide parallelism for a programmer through an abstract interface. We feel that implicit parallelism should refer to techniques requiring zero input from the programmer, and we will use semi-implicit parallelism to refer to techniques that require the programmer to *opt in*.

1.1 LAZY LANGUAGES AND PARALLELISM

Advocates of purely functional programming languages often cite easy parallelism as a major benefit of abandoning mutable state [Hughes, 1983; Peyton Jones, 1989]. This idea drove research into the theory and implementation of compilers that take advantage of *implicit parallelism* in a functional program. Additionally, when research into implicit parallelism was more common, the work was often based on novel architectures or distributed systems, not commodity hardware [Peyton Jones et al., 1987; Hammond and Michelson, 2000].

Despite this research effort, the ultimate goal of writing a program in a functional style and having the compiler find the implicit parallelism still requires work. We believe there are several reasons why previous work into implicit parallelism has not achieved the results that researchers hoped for. Chief amongst these is that the static placement of parallel annotations is not sufficient for creating well-performing parallel programs [Hammond and Michelson, 2000; Hogen et al., 1992; Tremblay and Gao, 1995; Harris and Singh, 2007]. This work explores one route to improvement: the compiler can use runtime profile data to improve initial decisions about parallelism in much the same way a programmer would manually tune a parallel program.

In the case of custom hardware, research was unable to keep up with huge improvements in sequential hardware. Today most common desktop workstations are parallel machines; this steers our motivation away from the full utilisation of hardware. Many programmers today write sequential programs and run them on parallel machines. We argue that even modest speedups are worthwhile if they occur 'for free'.

Historically Moore's law has often provided a 'free lunch' for those looking to run faster programs without the programmer expending any engineering effort. Throughout the 1990s in particular, an effective way of having a faster x86 program was to wait for Intel™ to release its new line of processors and run the program on your new CPU. Unfortunately, clock speeds have reached a plateau and we no longer get speedups for free [Sutter, 2005]. Increased performance now comes from including additional processor cores on modern CPUs. This means that programmers have been forced to write parallel and concurrent programs when looking for improved wall-clock performance. Unfortunately, writing parallel and concurrent programs involves managing complexity that is not present in single-threaded programs. The goal of this work is to convince the reader that not all hope is lost. By looking for the *implicit parallelism* in programs that are written as single-threaded programs we can achieve performance gains without programmer effort.

Our work focuses on F-Lite [Naylor and Runciman, 2010]: a pure, non-strict functional language that is suitable as a core language of a compiler for a higher-level language like Haskell. We have chosen to use a non-strict language because of the lack of arbitrary side-effects [Hughes, 1989], and many years of work in the area of implicit parallelism [Hogen et al., 1992; Hammond, 1994; Jones and Hudak, 1993] however we feel that many of our techniques would transfer well to other language paradigms.

With the choice of a lazy functional language we introduce a tension, the evaluation order for lazy languages can be seen to be at odds with the goal of only evaluating expressions when they are needed (which is an inherently sequential evaluation order). For this reason we must use *strictness analysis* in order to statically determine what expressions in a program are definitely needed, allowing us to evaluate them in parallel. We note that even eager languages would require some form of analysis because eager languages tend to allow arbitrary side-effects, necessitating the careful introduction of parallelism in order to avoid altering the order-dependent semantics of eager programs.

In short, this work argues that static analysis is necessary but not sufficient for the automatic exploitation of implicit parallelism. We argue that *some* form of runtime feedback is necessary to better utilise the parallelism that is discovered via static analysis.

1.2 GOALS AND CONTRIBUTIONS OF THIS THESIS

The primary contribution of this thesis is to demonstrate that using search techniques based on dynamic execution of an automatically parallelised program is a robust way to help diminish the *granularity* problem that is difficult for static analysis to overcome.

Our contributions can be seen as follows:

- A method to automatically derive parallel strategies from a *demand context*
- A novel use of heuristic search techniques in representing the possible parallelism in a program as a multi-dimensional search space
- The use of runtime profiles to *disable* automatically introduced parallelism in a program

We show that for some programs, the combination of search and static analysis can achieve speed-ups without the need for programmer intervention.

1.3 THESIS ROADMAP

The thesis is divided into three broad parts:

Part I explores the central concepts and the overall idea. In Chapter 2 we review the work on parallel functional programming, discussing the benefits and drawbacks to different approaches of writing (or not writing!) parallel programs. Chapter 3 provides an overview of our technique and a standard vocabulary for the rest of the work.

Part II is devoted to the *static* aspects of our approach. This includes a review of strictness analysis and the motivation for utilising a projection-based analysis in Chapter 4. We present our technique for exploiting the results of strictness analysis in Chapter 5.

In Part III we first describe our experimental platform in Chapter 6, then discuss two experiments: Chapter 7 experiments with using heuristic search techniques based on the overall runtime of a program and in Chapter 8 we provide the compiler with access to more detailed runtime profiles.

Lastly, Part IV discusses our conclusions (Chapter 9) and possible future work (Chapter 10).

Part I

The Idea

2

PARALLELISM IN A FUNCTIONAL LANGUAGE

Fully automatic parallelization is still a pipe dream.
– Marlow [2013]

This chapter explores previous approaches to parallelism in a functional language, both implicit and explicit. We do this so that we may better understand the trade-offs we accept when utilising implicit parallelism and the difficulties involved.

Plan of the Chapter

The chapter is organised as follows. We begin in Section 2.1 by giving a brief overview of the history of graph reduction as a method for the execution of functional programs. Section 2.2 discusses the benefits that functional languages provide for parallelisation; we will focus on call-by-need (or lazy) semantics. We then explore the various methods for the *implementation* of lazy languages for sequential machines in Section 2.3. Understanding how lazy languages can be executed we will then show in Section 2.4 how the techniques can be extended to allow for parallel execution.[1] We then turn our attention to the programmer's view of parallelism in a lazy language. Section 2.5 explores the current state of the art for parallel programming (both explicit and implicit parallelism) in a lazy functional language. We also explore the explicitly parallel techniques because they inform the way we approach implicit parallelism.

2.1 A SHORT HISTORY OF GRAPH REDUCTION

1978 Turing Award winner, John Backus, used his acceptance speech to ask the question: "Can Programming be liberated from the von Neumann Style?" [Hudak et al., 2007]. The crux of his argument

[1] One of the nice things about side-effect-free languages is that this step is not too difficult.

was that the traditional and ubiquitous architecture of the day was not suitable for the eventual shift to parallelism and the performance gains that could be achieved through parallelism's use. This fed the interest in novel computer architectures that would more readily support parallelism.

More declarative languages, and particularly functional languages, were seen as being better suited to Backus' challenge than more traditional imperative languages. This is because many imperative languages were designed for the simplicity of compilation. They free the programmer from repetitive bookkeeping, such as saving registers for function calls and saving the intermediate computations in an expression, but do not conceal all aspects of the underlying machine. Most crucially, they allow arbitrary mutation and side-effects. This limits the amount of parallelism possible because the result of a function call can depend on more than the values of the passed arguments.

Work had already been carried out on non-von Neumann architectures before that time. However, much of it was in the form of abstract machines that functioned 'on top' of the von Neumann architecture [Turner, 2012].

In the early 1970s the idea of graph reduction is introduced [Wadsworth, 1971] and with it, the concept of lazy evaluation was ready to be formalised. Lazy evaluation has its roots in papers by Henderson and Morris, and Friedman and Wise [Turner, 2012]. People began to think of ways to better implement graph reduction machines. A big breakthrough for software reduction was Turner's SKI-Combinator reduction scheme in 1979 [Turner, 2012; Clack, 1999].

In the 1980s we saw a great interest in parallel architectures for functional languages. The two main conferences in the field were 'LISP and Functional Programming' (which was held on even years) and 'Functional Programming and Computer Architecture' (which was held on odd years).

Several novel architectures were developed with the hopes that they could surpass stock hardware in their performance. This line of research continued through the 1980s and into the early 1990s [Harrison and Reeve, 1987; Peyton Jones et al., 1987; Clack, 1999; Hammond, 1994].

The G-Machine in 1984 showed that lazy functional languages (which had always been considered inefficient and bound to being interpreted) could be implemented in an efficient manner on stock hardware via the compilation of supercombinators to machine code. However, the abstract machine could also be used in the implementation of a novel architecture itself [Augustsson and Johnsson, 1989a].

The 1990s saw a decline in the amount of research aimed at parallel functional programming. This was mainly due to the results from earlier research being less successful than had been hoped (some novel architectures did see good results, but they could not keep up with the improvements seen in the sequential hardware sphere) [Hammond, 1994; Clack, 1999].

The late 1990s and the 2000s saw a resurgence in the interest in parallel functional programming because of the ubiquity of multi-core computers today. Generally, many of the techniques discussed earlier are used, (GHC using the STG-Machine, for example) [Goldberg, 1988; Harris et al., 2005].

While still suffering from the von Neumann bottleneck, using multiple cores in lazy functional languages is widely considered to be a success. Many high-level abstractions have been introduced that provide the programmer with powerful tools to exploit their multi-core systems. Strategies [Trinder et al., 1998], parallel skeletons [Hammond and Rebón Portillo, 2000], the Par-Monad [Marlow et al., 2011], and Repa [Keller et al., 2010] are among the most prominent successes. We will explore these in more depth in Section 2.5.

2.2 FUNCTIONAL PROGRAMMING AND PARALLELISM

Research into parallelism in lazy purely functional languages has a long history that dates back to the early work on lazy functional languages [Hughes, 1983; Augustsson and Johnsson, 1989b; Plasmeijer and Eekelen, 1993; Peyton Jones, 1989].[2]

We are able to illustrate the issue with a simple example. The two reductions of sqr in Figure 1 illustrate the key differences between lazy evaluation and eager, or strict, evaluation.

In the case of eager evaluation the argument to sqr is evaluated *before* entering the function body. For lazy evaluation the argument is passed as a suspended computation that is only *forced* when the value is needed (in this case when x is needed in order to multiply $x * x$). Notice that under lazy evaluation $5 * 5$ is only evaluated once, even though it is used twice in the function. This is due to the *sharing* of the result. This is why laziness is often described as call-by-need *with sharing* [Hammond and Michelson, 2000].

[2] For a comprehensive review we suggest [Hammond and Michelson, 2000]

Eager Evaluation	Lazy Evaluation
$\quad \mathrm{sqr}\ (5 * 5)$	$\quad \mathrm{sqr}\ (5 * 5)$
$= \mathrm{sqr}\ 25$	$= \mathrm{let}\ x\ =\ 5 * 5\ \mathrm{in}\ x * x$
$= \mathrm{let}\ x\ =\ 25\ \mathrm{in}\ x * x$	$= \mathrm{let}\ x\ =\ 25\ \mathrm{in}\ x * x$
$= 25 * 25$	$= 25 * 25$
$= 625$	$= 625$

Figure 1: Eager and Lazy evaluation order for squaring a value.

In the case of sqr in Figure 1, both eager and lazy evaluation required the same number of *reductions* to compute the final result. This is not always the case; take the following function definitions

$$\begin{aligned}
&\mathrm{bot}\ ::\ \mathrm{Int}\ \rightarrow\ \mathrm{Int}\\
&\mathrm{bot}\ x\ =\ x + \mathrm{bot} x\\
\\
&\mathrm{const}\ ::\ a\ \rightarrow\ b\ \rightarrow\ a\\
&\mathrm{const}\ x\ y\ =\ x
\end{aligned}$$

In an eager language the expression $\mathrm{const}\ 5\ \mathrm{bot}$ will never terminate, while it would return 5 in a lazy language as only the first argument to const is actually *needed* in its body.

2.3 SEQUENTIAL REDUCTION MACHINES

2.3.1 G-*Machine*

Unlike conventional register machines, the G-Machine is a stack-based machine designed to perform *normal order* graph reduction. The key point for the G-Machine [Augustsson and Johnsson, 1989a] is that it extended the idea that Turner introduced with the compilation of functional programs to SKI combinators. But instead of relying on pre-determined combinators, why not use the high-level declarations defined by the programmer? By compiling code for each of these, we are able to produce efficient code for each top-level function, whereas before we only had efficient code for each pre-determined combinator. These top-level function definitions were coined *supercombinators*

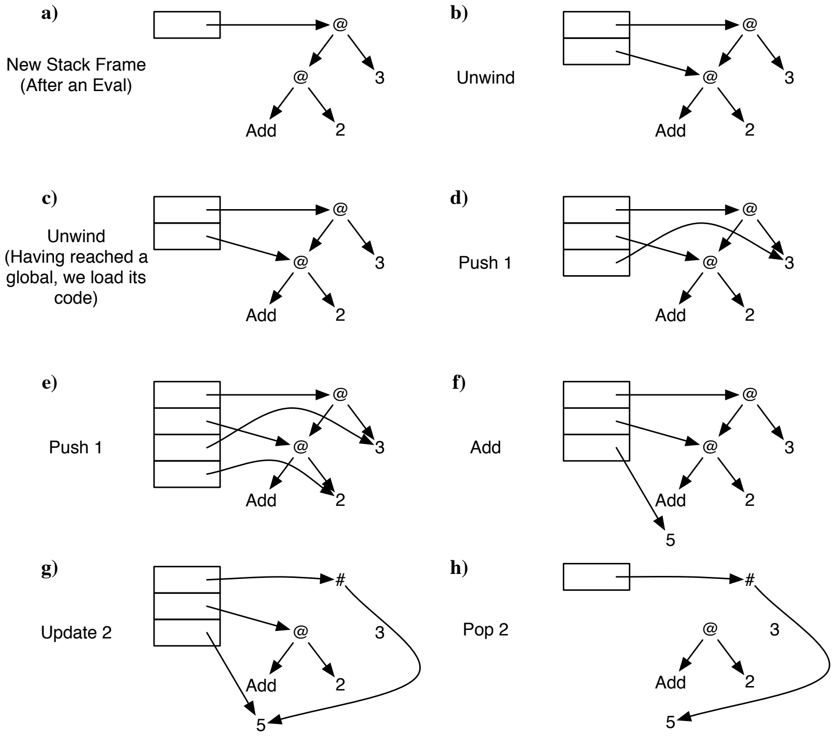


Figure 2: Walk-through of the G-Code for Add 2 3

by Hughes [Hughes, 1983]. Each supercombinator is required not to contain any lambdas on the right-hand side. In order to accomplish this for the G-Machine, Augustsson and Johnsson expanded on the lambda-lifting technique first outlined by Hughes [Augustsson and Johnsson, 1989a; Hughes, 1983].

Each supercombinator is compiled to what is essentially a reverse postfix notation for the right-hand side of the function definition. The resulting code constructs the graph representation of the function body and then reduces that graph to weak head normal form (WHNF).

In Figure 2 we walk through a simple G-Machine reduction. At (a) we have a reduction about to take place, the top of the stack is pointing to the root of the expression. In (b) the GCode instruction Unwind is executed, placing a pointer to the application node's function value on the stack. Unwinding continues until a global function is encountered. When unwind reaches a global function, as in (c), the G-Machine loads the code for that function and executes it. The code for Add is Push 1, Push 1, Add, Update 2, Pop 2, Unwind[3]. The first three instructions are what actually add the two arguments, with the last three instructions being used to update the graph and begin unwinding again. Push 1 pushes a pointer to the argument of the application node pointed to by the address located at stack address 1 (the stack addressing starts at 0). This is done twice at (d) and (e) in order to have pointers to both arguments at the top of the stack. Then we have the Add instruction which dereferences the top two pointers and adds the resulting values, the resulting value is then pushed onto the stack; this is seen at (f). With the result now on the stack, updating takes place, (g). The Update instruction takes a value (which is the arity of the function being evaluated) and updates the stack pointer that originally pointed to the root of the expression. The pointer is replaced by an indirection node, which, in turn, is set to point to the same value as the top stack pointer. With the updating finished the expression's original graph is no longer needed. Therefore, we can pop all of the intermediate pointers on the stack, which will always be equal to the arity of the expression. At (h) we are left with the updated pointer that can now be used by the calling function. We execute the Unwind instruction again, entering the G-Machine into its unwind state, which will ensure that proper stack dumping occurs when there is nothing left to evaluate (such as in this case).

2.3.2 *Spineless* G-*Machine*

The standard G-Machine updates the shared graph after every reduction step. While this is conceptually simple and easy to implement, such frequent rewrites in the heap can be seen as wasteful. The Spineless G-Machine improves on this by only updating the graph when loss of sharing is a possibility [Burn et al., 1988]. It is known as 'spineless' because the chain of application nodes that would make up the spine in the standard G-Machine is not built up in the heap. Instead it is exclusively accumulated on the stack until an update is required. The key point in this variant of the G-Machine is that updating should be associated with the potential loss of sharing and not with a reduction [Burn et al., 1988].

[3] This version of Add will only work with primitives as its arguments. In order to accept expressions there would need to be an Eval instruction added after every Push.

The way this was accomplished was by adding a new type of application node called a "SAP" node (for shared application). A SAP node indicates that the node is 'updatable' and that the result of any reduction at that node should be updated in the graph.

The mechanism for the updates is slightly different for functions than it is for values. First let us look at how functions are handled. When a SAP node is pushed onto the stack during an unwinding a new stack frame is created. When a pointer outside of the bounds of the stack frame is required we can take this as a signal that the reduction occurring at the updatable node has resulted in a function that 'fails' the arity test. We then rewrite the subgraph at the SAP node since we know that sharing could be lost if we continued otherwise. For values the update is triggered when the value has been reduced to its base-value form.

The remaining task for the Spineless G-Machine is to determine which application nodes should be a SAP node, and which should be a standard AP node. The authors give three different strategies for identifying which nodes should be marked as SAP. The simplest strategy is that all nodes should be marked as sharing nodes; this ensures that no sharing will be lost but will find the same wasteful re-writing that the standard G-Machine exhibited. The next strategy involves simple static sharing analysis to identify the nodes where the arguments will definitely *not* be shared, so all other nodes are marked as sharing nodes. Lastly, a dynamic method is suggested that attempts to identify as few sharing nodes as possible, subject to safety (therefore minimising the re-writing) while adding an overhead cost of having to check when a shared subgraph is created. The authors call this mechanism "dashing" and argue that the savings of having as few re-writes as possible make the overhead cost of dashing worthwhile [Burn et al., 1988].

2.3.3 *STG-Machine*

A few years after the design of the Spineless G-Machine, another enhancement was introduced, the Spineless Tagless G-Machine. This descendant of the G-Machine is a fully realised Spineless G-Machine that eliminates the need for tagged nodes by using a uniform representation for all values in its graph representation. All values are represented on the heap as *closures*.

Each closure consists of a code pointer and zero or more pointer fields used to point to the values of free variables. The code the closure points to is what determines the behaviour of that node in the graph. When the code for a closure is entered, the free variables can be accessed via an environment pointer that points to the original node in the graph (which has the pointer fields mentioned above) [Peyton Jones, 1992]. Because there is no tag on a node, the only way of determining the purpose of the node is to enter the closure. As expressed by Peyton Jones "each closure (including data values) is represented uniformly, and scrutinized only by entering it." [Peyton Jones, 1992].

The STG-Machine uses the self-updating model when having to update a subgraph. Instead of updating after every reduction, as in the G-Machine, each closure is responsible for updating itself when an update is necessary. In the case of a thunk, entering the closure will result in the thunk being overwritten and the resulting closure's code will simply return the value that has already been evaluated.

The STG-Machine became the basis for GHC and has had many improvements over the years [Hudak et al., 2007]. Interestingly, one of the improvements to the STG-Machine was the introduction of tags. The elegance of being able to treat every node the same has a cost on the performance on modern architectures [Marlow et al., 2007]. Because so much of the performance in modern CPUs comes from the speed of its cache (therefore avoiding the memory bottleneck) the indirections that arise from the code pointers in each closure have a significant cost on performance. As stated in the introduction to their 2007 paper [Marlow et al., 2007]:

> The tagless scheme is attractive because the code to evaluate a closure is simple and uniform: any closure can be evaluated simply by entering it. But this uniformity comes at the expense of performing indirect jumps, one to enter the closure and another to return to the evaluation site. These indirect jumps are particularly expensive on a modern processor architecture, because they fox the branch-prediction hardware, leading to a stall of 10 or more cycles depending on the length of the pipeline.

As much as we would like to focus on elegant abstractions and distance ourselves from any low-level concerns, there will always be

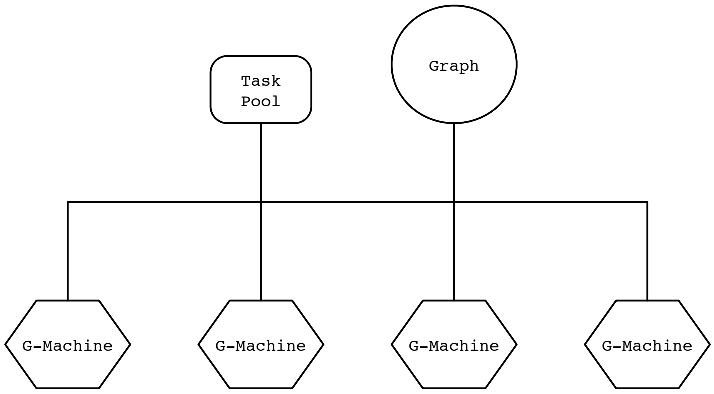


Figure 3: A parallel G-Machine

some focus on performance, and that will sometimes require compromises to accommodate hardware realities.

2.4 PARALLEL REDUCTION MACHINES

Having looked at the G-Machine as a sequential abstract machine we can now look at how graph reduction can occur on parallel architectures. The simplest parallel variant would be a parallel G-Machine. It turns out that once a sequential graph reduction has been accounted for there is not much to add in order to provide facilities for parallel evaluation. If one were to use a shared graph then almost all communication could take place via that graph.

The spark pool[4] is where idle processors can look for subgraphs that have been sparked off for evaluation. This simple-yet-functioning model is actually the basis for the implementation described in chapter 6 [Peyton Jones and Lester, 1992]. This model is not specific to the G-Machine, but can be used with any of the graph reduction machines, indeed GHC uses a similar model that is extended with thread-local heaps [Marlow et al., 2009]. A worker thread allocates from its local heap, when a thread's heap has overflowed the garbage collector stops *all* running threads and moves local-heap data to the shared heap.

[4] A *spark* is a pointer to an expression that has the *potential* to be executed in parallel. This allows the runtime system to 'throttle' the amount of parallelism in a program, when utilisation of the parallel machines is low, sparks are more likely to 'catch' than when utilisation is high [Clack and Peyton Jones, 1986].

2.4.1 $\langle \nu, G \rangle$-*Machine*

A departure from the stack-based approach of the G-Machine, here we have a packet-based abstract machine [Augustsson and Johnsson, 1989b; Harrison and Reeve, 1987]. The packets (or frames as they are called in the original paper) are similar to the closures we have seen for the STG-Machine (although the $\langle \nu, G \rangle$-Machine was described and implemented first [Augustsson and Johnsson, 1989b]). One difference is that on top of the code pointer and pointer to its arguments, each node contains a dynamic pointer that points to the caller of the expression the packet represents. Each node also contains a block of free 'working space' in each packet (we will see why in a moment). The key difference is the way that the $\langle \nu, G \rangle$-Machine deals with its stack. As opposed to having a central stack, each packet has its own, using the free space allocated with the creation of the packet as its local stack. Therefore the stack for a task is always in the heap with the task itself [Augustsson and Johnsson, 1989b]. This is in contrast to the standard G-Machine, where the stack resides 'in' the machine itself and the tasks use that stack until they are blocked, at which point the task's stack is transferred to the heap until the task is awoken. The $\langle \nu, G \rangle$-Machine avoids any complications of dealing with the stack by distributing the stack amongst the graph. The stack frame at any point in the graph is accessed through the linked list of packets pointing to their caller.

2.5 APPROACHES TO PARALLELISM

When looking at parallel programming, it is important to make the distinction between concurrency and parallelism. Concurrency embodies the idea of multiple workers (threads, computers, agents, etc.) making progress on independent tasks. A standard example for concurrency is a modern web-server. Each connection to the web-server can be thought of as an independent sub-task of the program. The web-server does not have to be run on a multi-core machine for the concurrency to take place. A single-core machine is capable of running concurrent threads through scheduling and context switching.

Parallelism describes the simultaneous execution of tasks with the purpose of achieving a gain in performance. Tasks that can be easily divided into independent sub-tasks can easily be made into parallel algorithms. For example, if one wanted to compute the monthly average temperature for a given year, each month could be computed independently. If enough of these independent computations happen simultaneously there can be a substantial improvement in the program's wall-clock speed. Ideally, the increase in performance would scale at the same rate as the available parallel machinery (2x performance with two processors, 5x performance with 5 processors). Unfortunately, there are some unavoidable factors that prevent this ideal from being realised [Hughes, 1983; Hudak et al., 2007; Hammond, 1994]. The most basic fact preventing this ideal is that the executing machinery (whether virtual or physical) will necessarily introduce some overhead in the generation and management of parallel threads of execution [Peyton Jones and Lester, 1992]. Beyond that, it is unusual for a program to be perfectly parallel except in trivial cases. Most parallel programs exhibit non-uniform parallelism and complex data dependencies. This results in programs where parallel threads vary greatly in their processing time and contain threads of execution that will depend on the results of other threads. This results in threads having to wait for the result of another thread before commencing (known as blocking).

2.5.1 *Haskell*

Haskell is a lazy functional language benefiting from constant development since its inception in 1987[5] [Hudak et al., 2007; Jones, 2003]. Haskell as defined by the Haskell Report [Jones, 2003] does not feature parallelism as part of the language. However, functional programming has a long history of parallel implementations and Haskell is no different in this regard. Even the early implementations of Haskell had facilities for parallel computation.

HASKELL PARALLELISM The Glasgow Haskell Compiler (GHC) has extensive parallelism and concurrency support; this is part of what makes the compiler a popular implementation [Hudak et al., 2007]. While our eventual goal is to have access to compilers that take advantage of the implicit parallelism in our programs, it is useful to understand the tools and libraries that enable programmers to use *explicit* parallelism in their Haskell programs. Some of these tech-

[5] 1987 was when the academic community decided that an 'standard' language was needed to unify the study of lazy functional programming[Hudak et al., 2007], however, the first Haskell report was not published until 1990 [Jones, 2003]

niques are well-established and have decades of use and experience to draw from.

2.5.2 *Explicit Parallelism with `par` and `seq`*

The most popular Haskell compiler, GHC [Hudak et al., 2007], is able to compile and run parallel Haskell programs 'out of the box'. This ability is limited to the *shared memory processors*, also known as symmetric multiprocessors (SMP), that are nearly ubiquitous with the rise of multi-core architectures in modern CPUs. The GHC project provides several ways to utilise parallel-capable machines.

The first method is through the *par* and *seq*[6] combinators. The *seq* combinator has the following type.

$$seq :: \alpha \rightarrow \beta \rightarrow \beta$$

An expression of the form *seq a b* first forces the evaluation of its first argument to WHNF[7] and then returns its second argument. This allows a programmer to express sequential evaluation.

It is important to realise that GHC's implementation of *seq* is *not* guaranteed to force the evaluation of its first argument *before* its second argument, though it does guarantees that $seq\ \bot\ b$ results in $\bot$. The compiler may find an optimisation that circumvents the sequencing created by the combinator. In order to provide a combinator that *does* provide this guarantee GHC provides the *pseq*[8] combinator.

In the literature, the *par* combinator appears in one of two forms [Hudak et al., 2007; Hughes, 1983]. In order to differentiate between the two forms we will refer to the earlier version as the *applicative* par, or *parAp* and the more recent (and more common) version as *Haskell par*.[9]

Haskell *par* takes the same form as *seq* and has the same type signature. The combinator takes two arguments, sparks off the first for evaluation in parallel, and returns the second.

[6] While `seq` was introduced into Haskell for Haskell '98 [Jones, 2003] it was used for many years before that [Hudak et al., 2007]. One of the earliest descriptions was by Hughes who introduced a `synch` combinator that performed the same function in 1983 [Hughes, 1983].

[7] For an expression to be in Weak Head Normal Form it must be evaluated such that the outermost constructor is known, but not necessarily any further. To say that a function is in WHNF means that the function is partially applied.

[8] The 'p' in `pseq` stands for parallel. The idea being that parallel programs are more likely than sequential programs to require that `seq` guarantees its behaviour.

[9] The reason we will be referring to this as Haskell par is because most users will know this version of the combinator from their use of Haskell

$$
\begin{aligned}
&\mathit{par} &&:: \alpha \rightarrow \beta \rightarrow \beta \\
&\mathit{par}\ a\ b &&= b
\end{aligned}
$$

The applicative *parAp* expresses a function application whose parameter has been sparked off to be evaluated in parallel. Semantically, this means that the applicative par has the following form

$$
\begin{aligned}
&\mathit{parAp} &&:: (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \\
&\mathit{parAp}\ f\ x &&= f\ x
\end{aligned}
$$

Interestingly, the version of *par* that an implementation chooses does not change the expressiveness. Each version of *par* can actually be defined in terms of the other. Defining the applicative par in terms of Haskell par gives us

$$\mathit{parAp}\ f\ x = \mathit{par}\ x\ (f\ x)$$

In order to define Haskell par in terms of applicative par we must use the K combinator

$$K\ x\ y = x$$

$$\mathit{par}\ x\ y = \mathit{parAp}\ (K\ y)\ x$$

The benefit of *parAp* is that you get the sharing of the parallel computation without needing to give a name to a subexpression. For example, the following use of *parAp* is very typical.

$$\mathit{parAp}\ f\ (g\ 10)$$

This does as you would expect: evaluate g 10 and share that with f. In order to achieve the same effect using Haskell *par* we must give g 10 a name.

$$
\begin{aligned}
&\mathbf{let} \\
&\quad x = g\ 10 \\
&\mathbf{in} \\
&\mathit{par}\ x\ (f\ x)
\end{aligned}
$$

While language implementors may have strong preferences for one over the other, there are a few good arguments for the use of Haskell *par*. Haskell *par* is the simpler of the two versions, using it as an infix combinator makes it easy to spark an arbitrary number of expressions easily, *a 'par' b 'par' c*, and the use-case for applicative *par* can be easily expressed using Haskell *par*, as shown above.

writing a program with par and seq Now that we have our essential combinators we are able to define a parallel algorithm. One of the big sellers of functional programming is the wonderfully concise *quicksort* definition

```
quicksort          :: (Ord α) ⇒ [α] → [α]
quicksort (x : xs) = lesser ++ x : greater
  where
    lesser  = quicksort [y | y ← xs, y ⩽ x]
    greater = quicksort [y | y ← xs, y > x]
quicksort _        = []
```

The obvious way to parallelise this algorithm is to ensure that each of the two recursive calls can be executed in parallel. This is a common form of parallelism known as 'divide and conquer' [Hammond and Rebón Portillo, 2000]. This can be done by changing the second line to

```
quicksort (x : xs)  = greater ‘par‘ (lesser ++ x : greater)
```

The issue with the above is that while the left-hand side is sparked off in parallel, it will only be evaluated to WHNF if the spark catches. This will result in only the first *cons* of the list. The rest of the list will only be evaluated if/when *greater* is needed.[10] This fails to exploit all of the parallelism we desire and highlights the sometimes conflicting nature of parallelism and laziness.

In order to help us attain the parallelism we are aiming for, we can introduce a function *force* that ensures that its parameters are evaluated fully. As found in the textbook "Real World Haskell" [O'Sullivan et al., 2009]

```
force      :: [α] → ()
force list = force′ list ‘pseq‘ ()
  where
    force′ (_ : xs) = force′ xs
    force′ []       = ()
```

This function takes a list and enforces spine-strictness. As long as the list is not an infinite structure, the use of this function is safe. With this function in hand we could adapt our basic parallel *quicksort* into a better performing one.

An interesting point is that this definition of *force* can be much simpler:

[10] Because of laziness it is possible that `greater` will never be needed. An example would be if only the head of the sorted list is requested.

```
force          :: [α] → ()
force (_ : xs) = force xs
force []       = ()
```

Because the function is fully saturated, the recursive call will continue evaluating through the entirety of the list. This only goes to show that even experts sometimes misunderstand when combinators like *seq* and *pseq* are needed.

```
parQuicksort          :: (Ord α) ⇒ [α] → [α]
parQuicksort (x : xs) = force greater ‘par‘ (force lesser ‘pseq‘ (lesser ++ x : greater))
  where
    lesser  = parQuicksort [y | y ← xs, y ⩽ x]
    greater = parQuicksort [y | y ← xs, y > x]
parQuicksort _ = []
```

Notice that there were a few more changes than just calling *force* with the parallel spark. By also forcing the evaluation of *greater* and *lesser* before appending the two lists we ensure that both *greater* and *lesser* are constructed completely. While this version of a parallel *quicksort* does execute its recursive calls in parallel, it has come at a cost. First, the resulting list is no longer lazy. Using this function for determining the head of the resulting list would result in the entire list being computed. While the loss of laziness can sometimes be a worthwhile trade-off (particularly if the entire list will be required anyway) for an increase in speed, the second major cost is that this parallel *quicksort* does not result in a faster program!

Despite the added functions to ensure that laziness did not get in the way of our desired parallelism, the parallel sorting was actually slower than the sequential sort. Running a parallel *quicksort* on a two core machine, O’Sullivan found that this algorithm actually resulted in a 23% decrease in performance [O’Sullivan et al., 2009]. The reason for the slowdown comes up often in the design of parallel algorithms: there is a minimum amount of work a thread must perform in order to make the expense of sparking off the thread worthwhile.[11] This issue highlights what is known as granularity [Plasmeijer and Eekelen, 1993]. While the sparking of threads is relatively cheap in a system such as GHC, there is still a cost, and if the amount of work a thread performs is very little, the cost may not be worth it.

[11] On modern multi-core machines there are additional factors that can affect whether a thread can be worthwhile, cache effects, cost of context switching, etc.

One way of tackling this issue is by introducing the notion of *depth* to the function. An additional parameter can be be added to the function that acts as a count for the depth.

```
parQuicksort (x : xs) depth = if depth ⩽ 0
                              then
                                 quicksort (x : xs)
                              else
                                 . . . ⟨parQuicksort with (depth - 1)⟩
```

In this version we check to see if the desired maximum depth has been reached and, if so, then the recursive call is to the standard sequential sorting algorithm. If the max depth is not reached then we use the body of the parQuicksort defined above, with the depth argument to each recursive call being $(depth - 1)$. The desired maximum depth is determined by the value given as the depth argument at the top-level call of the function. This method of controlling the granularity is a useful one and allows for easy experimentation on the maximum depth for the problem at hand. The downside is that it contributes yet another concern for the programmer to worry about.

Managing the granularity of a parallel program is one of the bigger challenges facing parallel functional programming [Peyton Jones, 1989]. Whether decided by the programmer (with annotations) or by the compiler implicitly, the question of "is this task worth it?" will always come up when deciding what tasks should be sparked.

2.5.3 *Explicit Parallelism with the* Par *Monad*

One of the main benefits of using *par* and *seq* is that they provide a simple interface for expressing parallel computation without the need to worry about concurrency issues such as task/thread creation and synchronisation. Task communication is via the same mechanism that allows for lazy evaluation [Peyton Jones, 1989]. However, laziness can be difficult to reason about and gaining the maximum benefit from parallelism can often require the *forcing* of values beyond WHNF, much like what was done with the *force* function in the previous section. This motivated another approach to deterministic parallelism: the *Par* Monad [Marlow et al., 2011].

The *Par* Monad makes the following trade-off: for the cost of making the dependencies between computations explicit and prohibiting the use of shared lazy structures, users gain predictable performance and the ability to reason about when evaluation takes place. The library provides an API for dataflow parallelism in Haskell.

Internally the *Par* Monad is implemented using I-Structures [Arvind et al., 1989] in the form of *IVar*s, which provide the mechanism for communication between tasks. *IVar* are mutable cells with write-once semantics. The disciplined use of I-Structures in the *Par* Monad is what ensures deterministic parallelism.

To illustrate the data-flow nature of the *Par* Monad we can use its API to write a function that computes two calls to *fib* in parallel (this code is adapted from Marlow [2013, pg. 59]).

```
twoFib m n = runPar $ do
                        i ← new
                        j ← new
                        fork (put i (fib m))
                        fork (put j (fib n))
                        a ← get i
                        b ← get j
                        return (a, b)
```

The *Par* Monad provides us with an API that is quite explicit about the data-flow of the program. This allows the library to schedule the threads as appropriate. The function *put* takes an *IVar* and a value that can be *forced* and writes the value to the *IVar*. By being so explicit about the structure of the program the library allows for intuitive reasoning about the parallel performance.

2.5.4 *Semi-Implicit Parallelism with Strategies*

This section will look at Parallel Strategies to illustrate how a programmer can write the algorithm they want (for the most part) and then use strategies to specify how that algorithm should be parallelised. Strategies become central to our technique and are mentioned throughout this thesis, as they are used as the method of introducing parallelism into a program.

Strategies came in two waves; the first wave was introduced by Trinder et al. [1998] in ”Algorithm + Strategy = Parallelism”. This incarnation of strategies is simple to understand and easy to use effectively. The second wave was an adaptation of the idea that addressed an issue caused by the garbage collector, but the main *idea* remains the same [Marlow et al., 2010].

The type declaration for a Strategy is

```
type Strategy α = α → ()
```

The key point to take away from this is that strategies do not, and can not, affect the *value* of the computation.

It is possible to define strategies that do not introduce parallelism, but instead ensure that evaluation is carried out to some degree. For example, the strategy for doing nothing

```
r0 :: Strategy a
r0 _ = ()
```

And for evaluating the argument to WHNF

```
rwhnf   :: Strategy α
rwhnf x = x ‘seq‘ ()
```

These strategies are not often used on their own, but are used in conjunction with other strategies to achieve a goal. For example, applying a strategy to each element of a list can be expressed as

```
seqList                :: Strategy α → Strategy [α]
seqList strat []       = ()
seqLIst strat (x : xs) = strat x ‘seq‘ (seqList strat xs)

parList                :: Strategy α → Strategy [α]
parList strat []       = ()
parList strat (x : xs) = strat x ‘par‘ (parList strat xs)
```

Both of these functions are common strategies for working on lists. In the first case, *seqList* elements in the list are *sequentially* evaluated using *strat* In *parList* each element is evaluated in parallel using *strat* this gives no guarantee of the evaluation order, only that the sparks will point to each element of the list.

In order to use a strategy, the function *using* was defined; taking an expression and a strategy, it bridges the gap between the specified algorithm and the desired strategy.

```
using     :: α → Strategy α → α
using x s = s x ‘seq‘ x
```

This allows us to define *parMap* as

```
parMap            :: Strategy β → (α → β) → [α] → [β]
parMap strat f xs = map f xs ‘using‘ parList strat
```

With this we can evaluate all the elements of a list to whatever level the first argument specifies.

2.5.5 *Semi-Implicit Parallelism with Skeletons*

There are many cases where the *structure* of two programs are the same even though they are computing different results. As mentioned above, *quicksort* shares a structure with many algorithms known as 'divide and conquer'. Algorithmic skeletons allow for the re-use of *structural* parallelism between programs [Hammond and Rebón Portillo, 2000].

To a functional programmer, skeletons are higher-order functions that are passed the computation specific actions of a program and ensure that the actions are parallelised according to the pre-defined structural parallelism. Hammond and Rebón Portillo [2000] introduce skeletons for the *divide and conquer*, *farm* (like a generic *parMap*), *pipe*, and *reduce* (like a generic parallel *fold*) patterns. The *divide and conquer* skeleton is defined as follows:

```
dc triv solve divide conq x
  | triv x      = solve x
  | otherwise = conq (solveAll triv solve divide conq (divide x))
  where
    solveAll xs =
      Collection.map (dc triv solve divide conq) xs ‘using‘ parColl all
```

The process is rather simple, *dc* takes a predicate *triv* that determines whether the input value, *x*, is trivial. If so, we use the function *solve* for the trivial case. When *x* is not trivial we must split the problem into smaller chunks using *divide*. With the problem divided, we use *solveAll* which recursively calls *dc* in parallel over all of the divided parts of the problem. The parallelism is introduced using a Strategy in *solveAll*. Lastly, *conq* is used to combine the results of the sub-problems.

There are two restrictions on the use of *dc*. The first is that the result be a member of the class *Evaluable* which has three member functions, *none*, *outer*, and *all*. These functions correspond to strategies with various degrees of evaluation: *none* performs no evaluation of the value, *outer* evaluates to WHNF, and *all* evaluates to normal form. The second is that the container type must be a member of *Coll* which provides a uniform interface for *map*ing over its elements.

Skeletons provide a powerful tool when working with problems that are known to be parallelisable. If a programmer can recognise which skeleton is appropriate for their problem domain they can focus on the aspects that are specific to their problem and let the skeleton manage the parallelism [Hammond and Rebón Portillo, 2000].

2.5.6 *Semi-Implicit Parallelism via Rewriting*

Some recent work has shown promising results with the use of rewriting. The programmer uses pre-selected functions (which is what makes these techniques semi-implicit) and the compiler uses predefined rewrite rules to convert the high-level function to low level parallel code [Steuwer et al., 2015]. This technique frees the programmer from concerning themselves with the low-level optimisation techniques, which in the case of GPU programming are often vendor specific. The compiler writers provide different sets of rewrite rules for each of the provided high-level functions. The compiler then searches the space of rewrite rules, allowing it to tune the performance of each function to a specific GPU. Because this search is done offline, the programmer can freely use any of the provided parallel functions knowing that whatever GPU the program runs on, an efficient sequence of rewrite rules is known to go from the high-level source to the target GPU.

This technique, like ours, can be seen as applying search-based techniques to the problem of knowing when parallelism can be accomplished. The benefit of their method is that it is restricted to a finite set of pre-determined parallel functions.

Another rewriting approach uses a *proof* of the program in separation logic, to automatically parallelise the program where it is safe, having the safety guaranteed by the provided proof [Hurlin, 2009]. The proof can be transformed as the program is transformed providing assurance that the resulting program has the same semantic behaviour. This technique does save the programmer from having to manually parallelise the code, at the cost of requiring that the algorithm has a separation proof. In any instance where such a proof is required for any other reason, this technique can be seen as 'free'. Otherwise the cost of proving separation could be limit the viability of this method.

2.5.7 *Implicit Parallelism*

While explicit parallelism has many advantages and can show great performance increases, many desire the ability to express a functional program *without* having to specify where parallelism can take place. This idea, that parallelism can be achieved without programmer intervention, is known as implicit parallelism. For non-strict languages it is not possible to parallelise all possible expressions because doing so may introduce non-termination that would not have occurred otherwise. *Strictness Analysis* is usually used to determine which expressions are *needed* and therefore safe to parallelise. Strictness analysis will be covered in far more detail in Chapter 4, but we will provide a high-level overview here to provide some context.

Strictness Analysis for Implicit Parallelism

Laziness can work against our desire to exploit possible parallelism in functional programs. Because of this, researchers have discovered that using static analysis at compile time in order to discover strict portions of a program can yield promising results. This analysis has been named *strictness analysis* [Mycroft, 1980; Loogen, 1999; Peyton Jones, 1989]. A trivial example is that of the addition of two expressions, such as the fibonacci sequence.

```
nfib n
  | n ⩽ 0     = 0
  | n ≡ 1     = 1
  | otherwise = nfib (n − 1) + nfib (n − 2)
```

Because the (+) function is strict in both its arguments we know that both recursive calls to *nfib* will be required. This fact (that we *know* that the arguments to a function will be required by the function) is what enables us to exploit the parallelism that is inherent in the program.

Parallelism obtained by strictness analysis has been deemed *conservative* parallelism [Peyton Jones, 1989]. This is due to the idea that sparking an expression that will definitely be required by the program is not seen as risky. However, keep in mind that while it is known as *conservative* parallelism it does not mean that there is any shortage of parallel tasks, it only refers to the fact that no *speculative* parallelism will be undertaken. Throughout this thesis we will sometimes refer to conservative parallelism as *safe* parallelism. This is to reinforce that the key is avoiding the introduction of non-termination where there was none in the original program.

One of the issues with standard strictness analysis when used for implicit parallelism is that it does not take into account the *context* in which an expression takes place. Take the standard recursive definition of *append*:

$$\begin{aligned} &\mathit{append}\ [\,]\ \mathit{ys} &&= \mathit{ys} \\ &\mathit{append}\ (x : \mathit{xs})\ \mathit{ys} &&= x\ :\ \mathit{append}\ \mathit{xs}\ \mathit{ys} \end{aligned}$$

In general the first argument is needed, but only to the outermost constructor. However, when *append* is used in certain contexts, like returning its result to a function that requires a finite list:

$$\mathit{length}\ (\mathit{append}\ \mathit{xs}\ \mathit{ys})$$

here *append* actually requires both of its arguments to be finite. This notion of context has been dealt with in two main ways in the literature: *Evaluation Transformers* and *Projection-Based Strictness Analysis*. We will examine Evaluation Transformers in Chapter 5, and Projection-Based Strictness Analysis in Chapter 4. In short both of these techniques allow the compiler or runtime system to take advantage of the information that an expression's context provides regarding how defined a value must be.

Many of the attempts at automatic parallelisation used the technique pioneered by Hogen et al. [1992], which uses strictness analysis to identify safe parallelism and *evaluation transformers* [Burn, 1987] to good effect. The main setback of this popular approach is that there is no method for *refining* the initial placement of parallelism. Strictness analysis is not equipped to determine the *benefit* of evaluating an expression in parallel, only the *safety*. We refer to this problem as the *granularity problem* throughout the thesis.

Feedback-Directed Implicit Parallelism

Harris and Singh [2007] proposed using *iterative feedback* in order to assist the compiler in determining which expressions are expensive enough to warrant evaluation in parallel. Their approach forgoes the use of strictness analysis to place parallelism into a program. Instead they utilise runtime profiling to measure the lifetime of thunks in the heap and the relationship between the thunks. They can then study dependency graph between thunks and determine which thunks are worthwhile for parallelism.

Because each thunk has a unique allocation site, they are able to provide an estimate for the 'total work' performed by an allocation site. They define this measure as t/a where t is the amount of time that *at least* one thunk from that allocation site is being evaluated, and α is the number of thunks allocated during that time. This prevents an allocation site from being deemed worthwhile because of a long lifetime but in reality it creates numerous thunks each requiring a small amount of work, flooding the runtime system with small-grained tasks [Harris and Singh, 2007, Section 3].

2.6 SUMMARY OF THE CHAPTER

In this chapter we have provided an broad introduction to the field of parallel functional programming. The literature includes work on runtime systems, static analyses, and library design. Over time programming with *par* and *seq* directly has proven to be difficult, the level of abstraction is too high for reasoning about performance but too low to be able to generalise one program's parallel implementation to another.

Parallel Strategies solve this problem to some degree, allowing the algorithm to be separated from the parallelism permits programmers to reuse their intuitions about which Strategies may be worthwhile for different problem domains. Strategies still suffer from the fact that lazy programs can be difficult to reason about however, and often do not provide the speedups a programmer might expect.

The *Par* monad attacks the problem from the other direction. By giving programmers a clearer view of when evaluation is occurring, performance can be more easily predicted. This comes at the slight cost of requiring the program to be more explicit in some cases[12] and that all structures be evaluated to normal form.

While much of the recent work in parallel functional programming has been in explicit or semi-implicit techniques there has been a recent resurgence in interest for fully implicit parallelism. The technique in Harris and Singh [2007] produced promising results but suffered from difficulties scaling to larger programs. Our work seeks to benefit from the insight presented in loc. cit. that using runtime profiles can help solve the granularity problem, but instead of using iterative feedback to *find* parallelism, we use the feedback to *prune* the parallelism that has been introduced by strictness analysis.

[12] The library does provide some high-level functions that allow the programmer to use the *Par* monad without needing to write their code in the monad itself, but on complex problems it is often necessary to write the *Par* monad code directly.

Our work can be seen as a combination of the most successful methods to date: static analysis for the placement of parallelism [Hogen et al., 1992] and feedback iteration to improve the static placement [Harris and Singh, 2007].

3

BIRD'S EYE VIEW OF OUR TECHNIQUE

I thought the "lazy functional languages are great for implicit parallelism" thing died out some time ago

– Lippmeier [2005]

This chapter is meant to accomplish two important goals: to provide a common vocabulary for the rest of the thesis and to familiarise the reader with the 'gist' of our proposed technique. In regards to the first goal we will introduce the syntax of F-lite and define terms that will be used throughout the rest of the thesis. By offering an overview of our technique, the reader will also possess context for each of the later chapters.

That being said, this chapter can be skipped if the reader is comfortable with functional languages and compilers.

Plan of the Chapter

The chapter begins by defining the syntax and semantics of F-lite and Folle in Section 3.1 which are higher-order and first-order languages respectively. Section 3.2 presents a high-level view of our compiler and its organisation.

3.1 F-LITE: A LAZY PURELY FUNCTIONAL CORE LANGUAGE

The use of a small functional language as the internal representation of a compiler is a common technique in functional compilers [Plasmeijer and Eekelen, 1993; Peyton Jones and Lester, 1992; Augustsson and Johnsson, 1989a; Dijkstra et al., 2009]. By using a small core language as an internal representation, source language features are simply syntactic sugar that is translated to a simpler but no less expressive language. This provides compiler writers with a smaller surface area for analysis and transformation. This has been used to great effect in the Glasgow Haskell Compiler (GHC) which uses a small core language similar to ours [Peyton Jones and Marlow, 2002; Peyton Jones and Santos, 1998].

3.1.1 *Why be Lazy?*

Functional languages vary widely in their syntax, features, and type systems, but almost all functional languages are either strict (eager) or non-strict (and usually lazy) in their evaluation model. It is important to understand the distinction between these two systems. Because functional languages can be seen as enriched lambda calculi, we can study different evaluation orders[1] by demonstrating them on a simple lambda calculus. There are a few evaluation strategies that can be used with the lambda calculus:

1. Call-by-value
2. Normal-order
   a) Call-by-name
   b) Call-by-need

Call-by-name and call-by-need are both *subsets* of normal-order reduction [Abramsky, 1990]. The differences between these strategies can be easily illustrated using the following function definitions:

$$\begin{aligned} \mathsf{sqr}\ x &= x * x \\ \mathsf{bot}\ \_ &= \bot \end{aligned}$$

Now assume we want to evaluate the expressions $sqr\ (2 * 3)$ and $bot\ (2 * 3)$. We can manually reduce each of these expressions using each of the evaluation orders.

[1] Many texts describe them as *evaluation strategies*. We use the term 'order' to avoid confusion with parallel strategies, which are a different concept that play a central role in this thesis.

$$\begin{aligned} & \mathrm{sqr}\ (2*3) \\ = {} & \mathrm{sqr}\ 6 \\ = {} & \mathrm{let}\ x\ =\ 6\ \mathrm{in}\ x*x \\ = {} & 6*6 \\ = {} & 36 \end{aligned}$$

$$\begin{aligned} & \mathrm{bot}\ (2*3) \\ = {} & \mathrm{bot}\ 6 \\ = {} & \mathrm{let}\ x\ =\ 6\ \mathrm{in}\ \bot \\ = {} & \bot \end{aligned}$$

Figure 4: Call-by-value reduction

CALL-BY-VALUE Note that the argument to *sqr* and *bot* is evaluated *before* we enter the function's body.

$$\begin{aligned} & \mathrm{sqr}\ (2*3) \\ = {} & \mathrm{let}\ x\ =\ (2*3)\ \mathrm{in}\ x*x \\ = {} & \mathrm{let}\ x\ =\ (2*3)\ \mathrm{in}\ (2*3)*x \\ = {} & \mathrm{let}\ x\ =\ (2*3)\ \mathrm{in}\ 6*x \\ = {} & 6*(2*3) \\ = {} & 6*6 \\ = {} & 36 \end{aligned}$$

$$\begin{aligned} & \mathrm{bot}\ (2*3) \\ = {} & \mathrm{let}\ x\ =\ 2*3\ \mathrm{in}\ \bot \\ = {} & \bot \end{aligned}$$

Figure 5: Call-by-name reduction

CALL-BY-NAME Here reduction delays the evaluation of a function's argument until its use. However, the result of evaluating a value is not shared with other references to that value. This results in computing $2*3$ twice.

$$
\begin{array}{ll}
\begin{aligned}
& \mathsf{sqr}\ (2 * 3) \\
={} & \mathsf{let}\ x \ = \ 2 * 3\ \mathsf{in}\ x * x \\
={} & \mathsf{let}\ x \ = \ 6\ \mathsf{in}\ 6 * x \\
={} & 6 * 6 \\
={} & 36
\end{aligned}
&
\qquad
\begin{aligned}
& \mathsf{bot}\ (2 * 3) \\
={} & \mathsf{let}\ x \ = \ 2 * 3\ \mathsf{in}\ \bot \\
={} & \bot
\end{aligned}
\end{array}
$$

Figure 6: Call-by-need reduction

CALL-BY-NEED This is designed to avoid the duplication of work that is often a result of call-by-name evaluation. Notice that in this evaluation $(2 * 3)$ is bound to x as before but the result of computing the value of x the first time *updates* the binding. This is why call-by-need is often referred to as call-by-name *with sharing*, or *lazy*.

An important point is that for languages without arbitrary side-effects call-by-name and call-by-need are semantically equivalent. Call-by-need is an optimisation for the *implementation* of reduction.

$$
\begin{array}{ll}
\begin{aligned}
& \mathsf{sqr}\ (2 * 3) \\
={} & \mathsf{let}\ x \ = \ 2 * 3\ \mathsf{in}\ x * x \\
={} & \mathsf{let}\ x \ = \ 2 * 3\ \mathsf{in}\ 6 * x \\
={} & \mathsf{let}\ x \ = \ 2 * 3\ \mathsf{in}\ 6 * 6 \\
={} & 36
\end{aligned}
&
\qquad
\begin{aligned}
& \mathsf{bot}\ (2 * 3) \\
={} & \bot
\end{aligned}
\end{array}
$$

Figure 7: Normal order reduction

NORMAL ORDER This method of evaluation is the only method that obeys the semantic property that $\lambda \_ \to \bot \equiv \bot$. This is because normal order reduction will evaluate under a lambda [Abramsky, 1990].

Of the four, only the first three are commonly used as the basis for programming languages. Most languages are call-by-value, this includes functional languages such as Scheme, OCaml, SML, and Idris. Fewer languages are call-by-name, Algol 60 being the most notable case. Scala, while being call-by-value by default, does allow programmers to specify that some functions use call-by-name. Lastly, call-by-need is used by Haskell, Clean, Miranda, and our own F-lite.

The reader may have noticed that in our examples above the result of evaluation was always the same *when they terminated*, regardless of evaluation order. This is an observation of a more general property about rewrite systems known as *confluence*. The lambda calculus was proven to be a confluent system by Church and Rosser in 1936 [Church and Rosser, 1936]. When discussing the lambda calculus specifically, it is referred to as the Church-Rosser property.

Church-Rosser Property

> The fact that the pure lambda calculus is *confluent* means that if there is more than one possible reduction step, the choice of reduction does not alter the final result *as long as the chosen reduction steps eventually terminate*.

We can illustrate the ramifications of the property with another simple example. The function *const* is a function that takes two arguments and returns the first. The value *inf* is an infinite list of 1s.

```
const x y = x

inf = 1 : inf
```

The expression *const "NF" inf* has multiple reducible expressions (redexes), but only one normal form (NF): *"NF"*.

```
  const "NF" inf
= const "NF" (1 : inf)
= const "NF" (1 : 1 : inf)
= ...   -- reduce forever
```

```
  const "NF" inf
= "NF"
```

Figure 8: Eager (left) and Lazy (right) evaluation order for const

The Church-Rosser property gives us a profound guarantee for our functional programs: given a valid expression, there is only one normal form for the expression. This is true regardless of the order of reductions carried out (given that the series of reductions actually terminates). So given a program, there can be many possible reduction orders that all lead to the same result. Additionally, if *any* reduction order terminates, then call-by-need evaluation terminates [Bird, 2014]. What does this mean for sequential computation? For lazy languages, such as Haskell, this means that there can only be non-termination if there would have been non-termination under any other evaluation model.

With the Church-Rosser theorem in hand and knowing that call-by-need programs are more likely to terminate than call-by-value equivalents, does the evaluation order we choose affect our aims with regard to automatic parallelisation? The main motivator for choosing call-by-need is that the evaluation order makes purity an essential part of the language. Because evaluation of an expression can be delayed for any amount of time, allowing arbitrary side-effects would make programming extremely difficult. By keeping the language pure we gain the full benefits of the Church-Rosser property.

Systems designed to take advantage of implicit parallelism have been written for languages that use each of the three main evaluation orders. We have decided on call-by-need semantics because it emphasises purity and features the sharing of computation built into the execution model. The focus on purity allows the compiler to take certain liberties with program transformation that may not otherwise be valid [Peyton Jones and Santos, 1998]. In the case of auto-parallelisation, we are able to know that we could only alter the semantics of a program by introducing non-termination. As we will see in Chapter 4 there are methods to ensure we avoid this.

AN ASIDE Many languages, including functional languages, that use call-by-value semantics also provide the ability to perform arbitrary side-effects and mutation. This greatly hampers the feasibility of implicit parallelisation because the *sequence* of side-effects can alter the semantics of the program. While programmers *could* write pure programs that do not rely on shared state, it is not enforced by the compiler as it is for languages like Haskell. That being said, there are techniques that can be used to find safe parallelism in strict languages [Might and Prabhu, 2009].

```
type Prog    = [Decl]

data Decl    = Func Id [Pat] Exp
             | Data Id [Id] [(Id, [TypeExp])]

type Id      = String

data Exp     = App Exp [Exp]
             | Case Exp [Alt]
             | Let [Binding] Exp
             | Var Id
             | Con Id
             | Fun Id
             | Int Int
             | Lam [Id] Exp
             | Freeze Exp
             | Unfreeze Exp

data Alt     = (Pat, Exp)

type Binding = (Id, Exp)

type Pat     = Exp    -- Applications of Cons to Vars
```

Figure 9: Abstract Syntax for F-lite

3.1.2 *The Abstract Syntax of F-lite*

Having motivated our choice of a lazy language we can now present F-lite [Naylor and Runciman, 2010] completely. We start with Figure 9 where the abstract syntax of F-lite is defined.

The language is an enriched lambda calculus with many of the usual conveniences: case expressions, recursive lets, and user-defined types. A key point is the presence of expressions of the form *Freeze e* and *Unfreeze e*. These expression are not found in the concrete syntax but are instead added by the compiler to make the suspension and forcing of lazy values explicit. This is a common technique when analysing non-strict programs with projections [Paterson, 1996; Hinze, 1995]. We will see the utility of these expressions in Chapter 4. We provide a function to translate standard F-lite expressions into F-lite expressions with explicit *Freeze* and *Unfreeze* in Appendix B.

3.2 OVERVIEW OF THE COMPILER

As the majority of our work is in the form of a compiler it is important to understand its organisation. Most of the phases present in our compiler can be found in standard compilers for lazy functional languages with the exception being that our compiler 'iterates' by running the program and altering the compilation based on that feedback. The compiler is organised into 8 main phases, as follows:

1. Parsing
2. Defunctionalisation
3. Projection-based Strictness Analysis
4. Generation of strategies
5. Placement of `par` annotations
6. G-Code Generation
7. Execution
8. Feedback and iteration

Parsing is completely standard and we will therefore omit discussing that stage of compilation. An interested reader is pointed towards "Implementing Functional Languages: A Tutorial" [Peyton Jones and Lester, 1992].

The rest of this dissertation focuses on the static analysis phases of the compiler (defunctionalisation, strictness analysis, and the generation of strategies) and the feedback and iteration phase. To see how the pieces all fit together we will demonstrate the important stages of the compiler with two small programs.

3.2.1 *Automatic Parallelisation: 1990s Style*

When research into implicit parallelism was at its height in the late 1980s and 1990s much of the focus was on using static analysis to introduce parallelism to programs. We can see how this was done with a simple example.

The program listed in Figure 10 is the Tak program benchmark, often used for testing the performance of recursion in interpreters and code generated by compilers [Knuth, 1991].

```
tak :: Int -> Int -> Int -> Int
tak x y z = case x <= y of
                True  -> z
                False -> tak (tak (x - 1) y z)
                             (tak (y - 1) z x)
                             (tak (z - 1) x y)

main = tak 24 16 8
```

Figure 10: Source listing for Tak

Luckily, this program is already first-order, so we do not need to worry about defunctionalisation. We therefore proceed directly to our strictness analysis (in this compiler we use a *projection-based* strictness analysis, which is discussed in Chapter 4). This phase of the compiler is able to determine which function arguments are *definitely* needed by each function. In the case of tak the strictness analysis determines that all three arguments are needed.

After we perform our projection-based strictness analysis we can introduce the safe par annotations, transforming the program into a parallelised version. The result of this transformation on Tak is listed in Figure 11.

Each needed argument is given a name via a let binding. This is so that any parallel, or seqed, evaluation can be shared between threads. When there are multiple strict arguments (as is the case for tak) we choose to spark the arguments in left-to-right order except for the last strict argument, which we seq. This is a common technique that is used to avoid potential collisions [Trinder et al., 1998]. Collisions occur when a thread requires the result of another thread

```
tak x y z = case x <= y of
                True  -> z
                False -> let x' = tak ((x - 1)) y z
                             y' = tak ((y - 1)) z x
                             z' = tak ((z - 1)) x y
                         in (par x'          -- par-site 0
                                (par y'      -- par-site 1
                                    (seq z'
                                        (tak x' y' z'))))

main  = tak 24 16 8
```

Figure 11: Source listing for Tak after analysis, transformation, and par placement

before that result has been evaluated. By ensuring that one of the arguments is evaluated in the current thread (by using `seq`) we give the parallel threads more time to evaluate their arguments, lessening the frequency of collisions.

Now that the parallelism has been introduced we can run our program on a multi-processor machine and hope to get speedups.

This is the hallmark of the conventional approach to automatic parallelism: determine a method of identifying potential parallelism, then transform the program to exploit that parallelism. However, this approach only addresses one half of the criteria needed for taking advantage of implicit parallelism: where is parallelism *safe*?

While static analysis has determined that `x'` and `y'` can be evaluated in parallel *safely*, it does not determine whether parallel evaluation of those expressions is *worthwhile*. This is the crux of the *granularity problem*. In the case of Tak we can run the program and see that we do indeed achieve performance increases from the parallelism that was introduced.[2] However, there is no process by which the compiler may alter its decision of what `par` annotations to introduce.

In the literature it was common to use further static analysis to estimate the cost of evaluating an expression. This estimate would then be used to avoid introducing parallelism that is too *fine-grained*. This was commonly done with both heuristic oracles and/or using a static cost analysis [Hogen et al., 1992] .

3.2.2 *Static Analysis is Not Enough*

The propensity for researchers to prefer static analysis over dynamic technique has many benefits, particularly the fact that they incur no runtime costs and have many decades of work to build upon. However, in some cases the (over)reliance on static analyses can be a hindrance instead of a benefit. We argue that implicit parallelism is an area that requires compiler writers to abandon the 'static analysis only' methodology.

An intuitive argument for our view is that parallelising programs is something that even *experts* have difficulty with. The work of determining what parallelism is worthwhile is often an *iterative* process between the programmer and the program. In other words; programmers do not 'write once', but instead annotate a program, profile the

[2] But it is not the fastest that we can achieve!

program, and alter the program according to the results of profiling. Why do we forbid compilers from utilising the same process?

Program analysis does not have to pick sides; compilers can benefit (if the implementors wish) from *both* static and dynamic analysis. In the case of implicit parallelism this means having the compiler perform a form of static analysis that provides an initial decision about parallelism in the program, then having those decisions either confirmed or rejected by dynamic, profile-driven analysis. There are several ways to achieve this goal. The method we suggest is to utilise *feedback-directed* compilation.[3]

Feedback-Directed Compilation

> The use of runtime profiles from previous executions of the program in order to inform an aspect of compilation or optimisation.

This frees the compiler from having to answer all of the difficult questions statically (and in one shot). Might and Prabhu [Might and Prabhu, 2009] suggest that the compiler has two questions it must ask when parallelising a sequential program :

1. Where is parallelisation *safe*?
2. Where is parallelisation beneficial?

Compiler writers have many tools available when wanting to answer the first: strictness analysis, dependency analysis, control-flow analysis, etc. [Hogen et al., 1992; Might and Prabhu, 2009]. In fact, these analyses are very good at answering the first question. The issue that has plagued work in automatic parallelisation is that there is often *too much* safe parallelism.[4] Determining the answer to the second question then becomes essential. Unfortunately determining the cost (or benefit) of evaluating an expression in parallel is notoriously difficult to do statically.

The problem is that while the safety of parallelising an expression is a static property of the program, the benefit is also affected by extrinsic factors such as the architecture the program is executed on. To be more precise; if there is a minimum cost to create parallelism, static analysis can determine that some expressions are definitely *not*

[3] Sometimes referred to as feedback-directed optimisation or iterative compilation

[4] This is true for pure languages, in languages that allow arbitrary side-effects finding safe parallelism can be more difficult.

worthwhile (by over-approximating the cost of evaluating an expression), however, static analysis cannot determine that an expression is *definitely* worthwhile. This is because the benefit of parallelising an expression is a function of *more* than the static semantics of a program.

Let us consider a simple thought experiment:

First, consider an ideal parallel machine that has an infinite number of processing units, zero-cost communication, and no context switches (each task gets its own processor).[5] The only cost to creating parallelism is the exact cost of the machine instructions used to initialise the parallel task. If a task requires less time than the initialisation cost, it is a net cost to the program and not worthwhile for parallelism. While unrealistic, this scenario demonstrates that there are some expressions that will not be worthwhile no matter the final program substrate.

Now consider the same machine, except with a finite number of processing units. Because each task is no longer able to retain a processor to itself, the machine must schedule tasks which imposes a context-switch cost on the tasks that are interrupted. Now whether a task is worthwhile depends not only on its intrinsic cost, but on the effect it has on the rest of the computation by interrupting other tasks. It is easy to see that more context-switches occur as the number of processing units available is reduced.

The difficulty of predicting the benefit of a parallel task becomes even more difficult as the machine becomes more realistic. Because of this we feel that it is sensible to ask: "Why have the compiler try at all?". Instead of having the compiler approximate this complex program property, we can run the program itself and have the compiler 'see' the effect its parallelisation has on the final program.

One question remains: in what way do we combine the static analysis with the feedback-directed compilation? There are several apparent approaches:

1. Have the compiler introduce parallelism sparsely, and use the iteration to introduce more parallelism

2. Have the compiler introduce what seems to be *too much* parallelism and have the iteration remove some of the parallelism

[5] We did say *ideal* machine.

3. Have the compiler provide an initial 'reasonable' parallelisation and have the iteration introduce/remove parallelism as necessary

In choosing our approach we chose to play to the strengths of the language we are conducting our experiments with. In pure functional languages there is an abundance of safe parallelism. Historically, parallelism in pure lazy languages often suffers more from the granularity problem than from a lack of available parallelism.For this reason we have opted to use the second approach: use static analysis to introduce as much parallelism as possible and use runtime feedback to disable some of the introduced parallelism.

We note that for a language that allows arbitrary side-effects it may be more beneficial to use the first approach as the runtime feedback would aid in determining which tasks were actually independent.

Later in Chapter 10 we will explore a variant of the third approach.

3.2.3 *Automatic Parallelism in the 21st Century*

In order to address this issue we take advantage of two key properties of our `par` annotations:

1. Each introduced `par` sparks off a unique subexpression in the program's source
2. The semantics of `par` (as shown in Figure 12 on page 51) allow us to return its second argument, ignoring the first, without changing the semantics of the program as a whole

These two properties allow us to represent the `par`s placed by static analysis and transformation as a bitstring (one bit per call-site). Each bit represents a specific `par` in the program's AST. The second property allows us to have 2 separate *operational* intepretations of `par` that maintain its semantic properties. When a `par`'s bit is 'on' the `par` behaves as normal, sparking off its first argument to be evaluated in parallel and return its second argument. When the bit is 'off' the `par` returns its second argument, ignoring the first.

This allows us to change the *operational* behaviour of the program without altering any of the program's semantics. In other words, we are able to *try* evaluating subexpressions in parallel, without the risk of introducing behavior that was not present in the original program.

ITERATIVE IMPROVEMENT Just because an expression is *able* to be evaluated in parallel does not mean that doing so is beneficial. This is one of the critical problems in implicit parallelism [Hogen et al., 1992; Hammond and Michelson, 2000; Jones et al., 2009]. To combat this we run the program as presented in Figure 11 and collect statistics about the amount of productive work each par is responsible for. The pars that do not introduce a worthwhile amount of parallelism are disabled, freeing the program from incurring the overhead of managing threads for tasks with insufficient granularity.[6]

Luckily, the Tak program has so few pars that an exhaustive exploration of the possible settings is easy. Table 1 shows the resulting speedup (as compared to the program with all parallelism turned off) from the different possible par settings. The bitstring "10" represents the version of Tak where the par labelled 0 in Figure 11 is on and the par labelled 1 is off.

	00	01	10	11
speedup	1	1.56	0.43	0.97

Table 1: Performance of Tak with different par settings

As we can see, more pars is not necessarily better. In fact, for our interpreter the setting 10 results in quite a lot of thread collisions, meaning that a thread pays the overhead for its paralellism and is almost immediately blocked. Paying the cost of creating and managing the parallelism, but not being able to perform any actual work is why the program performs *worse* than the version of Tak with all the parallelism switched off.

So why introduce this parallelism in the first place? Because the granularity and possible interference of parallel threads is difficult to know *statically* at compile time. Not only is it difficult to know these properties statically, but these properties differ from architecture to architecture [Steuwer et al., 2015]. If we err on the side of generosity with our par annotations we can then use *runtime* profiling to gather information about the granularity and interference of threads.

[6] This can be seen as a more extreme variation of Clack and Peyton Jones' "Evaluate and die!" model of parallelism [Clack and Peyton Jones, 1986]: Evaluate *a lot* or die!

3.3 SUMMARY

This chapter provided a high-level overview of our technique for exploiting the parallelism that is inherent in functional programs. We motivated our choice of a *lazy* language (using call-by-need) by considering its emphasis on purity and the fact that the sharing of values is built in to the evaluation model.

We introduced the structure of our compiler and walked through the steps of our technique using the `Tak` program as a small example. This allows for a clearer notion of where the remaining chapters fit into our technique and provides a common vocabulary for the rest of the thesis.

Now that we have presented the high-level view of our work we will now explore each of the stages in depth. Chapter 4 introduces strictness analysis and discusses the history of its development and why we have chosen *projection-based* strictness analysis for our compiler. Chapter 5 presents how to use the results from projection-based strictness analysis to *automatically* derive parallel strategies that are then used to introduce parallelism into the source program. Chapters 7 and 8 represent the iterative phase of our compiler, discussing two methods of searching the space of `par` settings.

Part II

The Discovery and Placement of Safe Parallelism

4 FINDING SAFE PARALLELISM

> *Parallelism is introduced in our system by the programmer annotating the programs. We have not yet addressed the problem of how to automatically place such annotations.*
>
> – Augustsson and Johnsson [1989b]

Non-strictness makes it difficult to reason about when expressions are evaluated. This is due to the fact that call-by-need languages only evaluate expressions when their results are needed, and when a value is needed can depend on runtime data. One of the benefits of this approach is that it forces the programmer to avoid the use of arbitrary side-effects. The resulting purity means that functions in pure functional languages are *referentially transparent*, or the result of a function depends only on the values of its arguments (i.e. there is no global state that could affect the result of the function or be manipulated by the function).

Unfortunately this elegant evaluation model is actually at odds with the goals of performance through parallelism: if we have parallel processing resources, we wish to use them to do as much work as possible to shorten execution time [Tremblay and Gao, 1995].

Call-by-need semantics forces our compiler to take care in deciding which sub-expressions can safely be executed in parallel. Having a simple parallelisation heuristic such as 'compute all arguments to functions in parallel' can alter the semantics of a non-strict language, introducing non-termination or runtime errors that would not have occurred during a sequential execution.

The process of determining which arguments are required for a function is known as *strictness analysis* [Mycroft, 1980]. Since the early 1980's such analysis has been widely used for reducing the overheads of lazy evaluation [Sergey et al., 2014]. As strictness analysis is a form of termination analysis it is undecidable in general and therefore any results are approximate. Usually the abstract semantics are chosen so that the analysis can determine when an expression is *definitely* needed.[1]

It is possible to throw caution to the wind and *speculate* on which expressions may be needed. This itself is a rich area of research and requires the compiler to identify plausible candidates but ensure that errors and non-termination do not affect the program as a whole . For our work we chose to utilise only *safe* implicit parallelism.

Safe Implicit Parallelism

> The parallelism inherent in a program from tasks/expressions that would have been evaluated during the program's terminating sequential execution. Any parallelism determined to be safe would not result in non-termination that was not already present in the program.

Safe implicit parallelism is in line with our overall goal of providing a system that guarantees that the parallelised program has the same semantics as the original. Therefore, before we can run our automatically parallelised programs, we must develop methods and techniques for the compiler to *find* and *express* the parallelism that is implicit in our programs. Strictness analysis is suitable in aiding this task but care must be taken in choosing a specific analysis to use.

This chapter is concerned with studying the various analyses and the trade-offs that are inherent in the differing approaches. A survey of the concepts and development of strictness analysis will inform our choice of analysis and allow us to understand the drawbacks and limits of our chosen method.

Plan of the Chapter

We provide a high-level overview of the issues and motivations in Section 4.2; this should provide enough context to those who want to move quickly to the next chapter and not concern themselves with the details of strictness analysis. The ideas are then expanded in the three sections that follow. Section 4.3 explores basic strictness analysis using the original two-point domain. We will see why a two-point domain results in an analysis that is too limited for our use in deriving useful parallel strategies. Using a four-point domain, which we discuss in Section 4.4, fixes much of this issue and provides much better information for parallel programs (and has been used

[1] 'Needed' in this context means 'needed for the computation to terminate'.

toward that end) but does not allow for analysis on arbitrary datatypes. Lastly, we review the work on projection-based analysis, which solves both issues, in Section 4.5.

4.1 ORIGINAL MOTIVATION VS. OUR MOTIVATION

Purity alone is of huge benefit when dealing with parallelism. Because functions do not rely on anything but their arguments, the only necessary communication between threads is the result of the thread's computation, which is shared via the program's graph using the same mechanism used to implement laziness [Peyton Jones, 1989].

Laziness, while forcing the programmer to be pure (a boon to parallelism), is an inherently sequential evaluation strategy. Lazy evaluation only evaluates expressions when they are *needed*. This is what allows the use of infinite data structures; only what is needed will be computed.

This tension between the call-by-need convention of laziness with parallelism's desire to evaluate expressions *before* they are needed is well known [Tremblay and Gao, 1995]. The most successful method of combating this tension is through the use of *strictness analysis* [Mycroft, 1980; Wadler and Hughes, 1987; Hinze, 1995].

4.2 OVERVIEW

Because we are working in a lazy language it is not always safe to evaluate the arguments to a function before we enter the body of a function. This is easy to see with a simple example; appending two lists:[2]

```
append            :: [α] → [α] → [α]
append [] ys      = ys
append (x : xs) ys = x : append xs ys
```

append takes two list arguments. A central question that strictness analysis asks is: How *defined* must the arguments be to *append* in order for *append* to terminate?

The first hint is that *append* pattern matches on its first argument. Because the function must be able to distinguish between a (:) and

[2] Here we have used the naïve recursive version, but any correct version of append will have the same strictness properties.

a [ ] we know that the first argument must be defined *at least* to the outermost constructor. Therefore, a use of *append* that is passed $\perp$ as its first argument will result in non-termination. What about the second argument, *ys*. Determining how defined *ys* must be turns out to be impossible without taking into account the *context* that a call to *append* occurs in. If the first argument is [ ] then *ys* must be defined to WHNF. However, the (:) case guards the recursive call to *append* and, by extension, the use of *ys*.

The literature on Strictness Analysis is the story of determining these properties in the general case. We start in Section 4.2.1 with exploring the simplest strictness analysis, *Ideal Analysis* on *flat domains*. We then show how the work was extended to *non-flat domains* in Section 4.4. Lastly, we show how the notion of *contexts* mentioned above are formalised by the use of *projections* from Domain Theory in section 4.5.

4.2.1 *Ideal Strictness Analysis*

If a function uses the value of an argument within its body it is safe to evaluate that argument before, or in parallel with, the execution of the body of the function. In order to determine which arguments can be evaluated in this way modern compilers use *strictness analysis* [Mycroft, 1980]. More formally, a function f of n arguments

$$f\ x_1 \ldots\ x_i\ \ldots x_n = \ldots$$

is strict in its ith argument if and only if

$$f\ x_1 \ldots\ \perp\ \ldots x_n = \perp$$

What this states is that f is only strict in its ith argument if f becomes non-terminating[3] by passing a non-terminating value as its ith argument.

Knowing the strictness information of a function is the first step in automatic parallelisation. This is because if f is strict in its ith argument we do not risk introducing non-termination (which would not otherwise be present) by evaluating the ith argument in parallel and waiting for the result. In other words, evaluating x_i in parallel

[3] In this thesis we use the convention that $\perp$ represents erroneous or non-terminating expressions.

```
seq :: a → b → b          par :: a → b → b
seq x y = y               par x y = y
```

Figure 12: Semantics (but not the pragmatics) of seq and par.

would only introduce non-termination to the program if evaluating f with x_i would have resulted in f's non-termination anyway.

F-Lite has two primitives for taking advantage of strictness information: par and seq, as shown in Figure 12.

Both functions return the value of their second argument. The difference is in their side-effects. seq returns its second argument only *after* the evaluation of its first argument. par forks the evaluation of its first argument in a new parallel thread and then returns its second argument; this is known as sparking a parallel task [Clack and Peyton Jones, 1986].

Strictness analysis was a very active research area in the 1980's and the development of analyses that provide the type of strictness information outlined above is a well understood problem [Mycroft, 1980; Clack and Peyton Jones, 1985; Burn et al., 1986]. However, as suggested above, strictness analysis does not provide satisfactory information about complex data-structures [Wadler, 1987]. This can be remedied by the use of *projections* to represent *demand*.

4.2.2 *Abstract Interpretation*

Mycroft introduced the use of abstract interpretation for performing strictness analysis on call-by-need programs over thirty years ago [Mycroft, 1980]. Strictness analysis as originally described by Mycroft was only capable of dealing with a two-point domain (values that are definitely needed, and values that may or may not be needed). This works well for types that can be represented by a flat domain (Integer, Char, Bool, etc.)[4] but falls short on more complex data structures. For example, even if we find that a function is strict in a list argument, we can only evaluate up to the first cons safely. For many functions on lists, evaluating the entire list, or the spine of the list, is safe; canonical examples are sum and length.

[4] Any type that can be represented as an enumerated type.

In order to accommodate this type of reasoning, Wadler developed a *four-point domain* for the abstract interpretation of list-processing programs [Wadler, 1987]. However, when extended in the natural way for general recursive data structures, the size of the domains made finding fix-points prohibitively costly.

4.2.3 *Projections and Contexts*

So far our discussion of strictness has only involved two levels of 'definedness': a defined value, or ⊥. This is the whole story when dealing with *flat* data-structures such as Integers, Booleans or Enumerations. However, in lazy languages nested data-structures have *degrees* of definedness.

Take the following example function and value definitions in F-Lite

```
length []     = 0                  sum []     = 0
length (x:xs) = 1 + length xs      sum (x:xs) = x + sum xs

definedList = [1,2,3,4]            infiniteList = [1,2,3...

partialList = [1,2,⊥,4]          loop = loop
```

Both `length` and `sum` are functions on lists, but they use lists differently. `length` does not use the elements of its argument list. Therefore `length` would accept `definedList` and `partialList` (which has a non-terminating element) as arguments and still terminate. On the other hand `sum` *needs* the elements of the list, otherwise it would not be able to compute the sum. For this reason, `sum` only terminates if it is passed a fully defined list and would result in non-termination if passed `partialList`. Neither function would terminate if passed `infiniteList`, since even `length` requires the list to have a finite length (some functions do not require a finite list, such as `head`, the function that returns the first element in a list). With these examples we say that `length` *demands* a finite list, whereas `sum` *demands* a fully-defined list.

This additional information about a data-structure is extremely useful when trying to parallelise programs. If we can determine *how much* of a structure is needed we can then evaluate the structure to that depth in parallel.

The work that introduced this representation of demands was by Wadler and Hughes [Wadler and Hughes, 1987] using the idea of *projections* from domain theory. The technique we use in our compiler is a projection-based strictness analysis based on the work in Hinze's dissertation [Hinze, 1995]. Hinze's dissertation is also a good resource for learning the theory of projection-based strictness analysis.

Strategies

With the more sophisticated information provided by projection-based analysis, we require more than simply par and seq to force the evaluation of values. To this end we use the popular technique of *strategies* for parallel evaluation [Trinder et al., 1998; Marlow et al., 2010]. Strategies are designed to evaluate structures up to a certain depth in parallel with the use of those structures. Normally, strategies are written by the programmer for use in hand-parallelised code. In order to facilitate auto-parallelisation we have developed a method to *derive* an appropriate strategy from the information provided to us by projection-based strictness analysis. The rules for the derivation are presented as a denotational semantics and will be discussed in Chapter 5.

4.3 TWO-POINT FORWARD ANALYSIS

As mentioned in the Introduction, the majority of functional languages use either call-by-need or call-by-value semantics. While call-by-need has many attractive properties the delaying of computation incurs an overhead cost on all computations. Call-by-value, by contrast, has an execution model that is more easily mapped to conventional hardware, allowing for simpler implementations that achieve good performance. Mycroft used this tension to motivate his development of strictness analysis [Mycroft, 1980]:

> The above arguments suggest that call-by-value is more efficient but call-by-need preferable on aesthetic/definedness considerations. So techniques are herein developed which allow the system to present a call-by-need interface to the user but which performs a pre-pass on his program

> annotating those arguments which can validly be passed using call-by-value.

By determining which arguments can be safely passed using call-by-value we diminish the overhead of call-by-need, paying the overhead of suspending computation only when necessary to ensure that call-by-need semantics are maintained.

While this was the original motivation for strictness analysis it also serves in identifying potential parallelism in a program. When an argument is suitable to be passed as call-by-value it is also suitable to be evaluated in parallel. In this case the value is evaluated in parallel to the original thread. Synchronisation is accomplished via the same mechanism as laziness,[5] with the exception that a thread can be blocked while waiting for another thread to complete its evaluation.

4.3.1 *Safety First*

Strictness analysis is chiefly concerned with *safety*. In order to retain the origin call-by-need semantics the runtime can only alter the evaluation order when doing so guarantees the same termination properties of the program.

We will refer to this notion of safety as the *strictness* properties of an argument. Take a function f of n arguments

$$f\ x_1\ \ldots x_i\ \ldots\ x_n\ = \langle\texttt{function body}\rangle$$

The ith argument of f is said to be strict *if and only if*

$$f\ x_1\ \ldots \bot_i\ \ldots\ x_n\ = \bot \tag{1}$$

For any possible values of $x_1 - x_{i-1}, x_{i+1} - x_n$.

Equation 4.3.1 can be read as "f is strict in x_i when f fails to terminate if x_i fails to terminate". The reason this allows us to evaluate the ith argument before it is needed is because doing so would only

[5] Remember that with the spark model of parallelism the synchronisation occurs via updates to the heap. When a thread begins the evaluation of an expression it marks the heap node as 'locked', stopping any other thread from repeating the computation. When the thread completes evaluation of the expression to WHNF, it updates the locked heap node to point to the result. Therefore the way threads share computation is identical to the way a sequential lazy evaluator shares the evaluation of expressions [Clack and Peyton Jones, 1986].

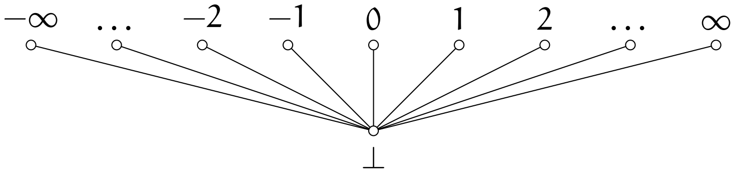


Figure 13: Flat Domain

result in introducing non-termination if the program would have resulted in non-termination otherwise.

4.3.2 *Abstract Domains*

Now that we have established what it means to be strict we can expand on how we analyse programs for this property. As with any abstract interpretation, this involves the choice of an abstract domain.

In non-strict languages types like Integers and Booleans form a flat domain; either we have a value of that type, or we have $\perp$. This is depicted in Figure 13. We can form an intuition of these orderings by thinking about how much we *know* about a certain value. While the integer value 5 maybe greater than the integer value 4, we know the same amount about each of them: their values. However, if we have a procedure that is meant to compute an integer and it loops forever, we cannot know that integer's value. Therefore we know less about a non-terminating value.

This fits nicely with call-by-need semantics: an argument to a function of type *Int* is really a computation that can either result in a value, or result in non-termination. In terms of strictness analysis, this allows us to abstract our real domain of Integers to the simple two-point domain shown in Figure 14.

This is the domain we use for basic strictness analysis. The bottom of the lattice, $\perp$, as implied above, represents *definitely non-terminating* expressions. The top of the lattice, $\top$, is used to represent *potentially*

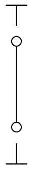


Figure 14: Two-point Domain

$$\begin{array}{lcl} \top \sqcap \top = \top & \qquad & \top \sqcup \top = \top \\ \top \sqcap \bot = \bot & & \top \sqcup \bot = \top \\ \bot \sqcap \top = \bot & & \bot \sqcup \top = \top \\ \bot \sqcap \bot = \bot & & \bot \sqcup \bot = \bot \end{array}$$

Figure 15: The *meet* ($\sqcap$) and *join* ($\sqcup$) for our lattice

terminating[6] expressions. This approximation can seem counterintuitive; why are we allowing the analysis to say some results are potentially terminating when they could be non-terminating? The reasoning is that non-terminating values do not imply a non-terminating program under non-strict semantics! If we approximated in the opposite direction (as analyses for other purposes sometimes do) we may accidentally compute a value that was never needed, defeating the purpose of call-by-need evaluation.

Abstracting Functions

Now that we know what it means to be strict and why we represent flat domains as a two-point domain the next step is to abstract the functions in our program.

The idea is simple: for every function in our program of type A we must produce a function of type $A^{\#}$ that works on the abstracted values.[7] For example, if A is the type $Int \rightarrow Int \rightarrow Int$, $A^{\#}$ would have the abstracted type $Int^{\#} \rightarrow Int^{\#} \rightarrow Int^{\#}$. This abstracted program is then interpreted using an abstract semantics that provides us with the strictness information for each function in our program.

While this separation of abstracting the program and then performing an abstract interpretation is useful from the theoretical point of view, many compilers skip the intermediate representation of an abstracted program and perform the abstract interpretation with the original AST [Hinze, 1995; Kubiak et al., 1992; Sergey et al., 2014].

We begin with the set of primitive arithmetic functions. In the case of F-Lite, each numeric primitive is strict in both arguments, providing us with the following for each of $(+)$, $(-)$, $(*)$, $(/)$:

[6] Remember that program analysis must approximate in the general case.

[7] Some texts represent an abstracted type by a number N where N is the number of points in the abstract domain. We prefer to retain the context of where this abstract domain came from.

$$
\begin{array}{ll}
\mathcal{A} & :: \mathit{Exp} \rightarrow \mathit{Env}^{\#} \rightarrow \mathit{Exp}^{\#} \\
\mathcal{A}\, [\![\mathsf{Var}\ \mathsf{v}]\!]\ \phi & = \phi\, v \\
\mathcal{A}\, [\![\mathsf{Int}\ \mathsf{i}]\!]\ \phi & = \top \\
\mathcal{A}\, [\![\mathsf{Con}\ \mathsf{c}]\!]\ \phi & = \top \\
\mathcal{A}\, [\![\mathsf{Fun}\ \mathsf{f}]\!]\ \phi & = \phi\, f \\
\mathcal{A}\, [\![\mathsf{App}\ (\mathsf{Fun}\ \mathsf{f})\ [\mathsf{a}_1,\ \ldots\ \mathsf{a}_n]]\!]\ \phi & = \mathcal{A}\, [\![\mathsf{f}]\!]\ \phi\ (\mathcal{A}\, [\![\mathsf{a}_1]\!]\ \phi)\ \ldots\ (\mathcal{A}\, [\![\mathsf{a}_n]\!]\ \phi) \\
\mathcal{A}\, [\![\mathsf{Let}\ \mathsf{b}_1\ =\ \mathsf{e}_1 \ldots \mathsf{b}_n\ =\ \mathsf{e}_n\ \mathsf{in}\ \mathsf{e}]\!]\ \phi & = \mathcal{A}\, [\![\mathsf{e}]\!]\ \phi[b_i \mapsto \mathcal{A}\, [\![\mathsf{e}_i]\!]] \\
\mathcal{A}\, [\![\mathsf{Case}\ \mathsf{e}\ \mathsf{alts}]\!]\ \phi & = \mathcal{A}\, [\![\mathsf{e}]\!]\ \phi \sqcap \mathcal{C}\, [\![\mathsf{alts}]\!]\ \phi
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}\, [\![\mathsf{c}_1\ \mathsf{vars}_1 \rightarrow \mathsf{e}_n, \ldots, \mathsf{c}_n\ \mathsf{vars}_n \rightarrow \mathsf{e}_n)]\!]\ \phi = e_1^{\#} \sqcup \cdots \sqcup e_n^{\#} \\
\quad \textbf{where} \\
\qquad e_1^{\#} = \mathcal{A}\, [\![\mathsf{e}_1]\!]\ \phi[\mathit{vars}_1 \mapsto \top] \\
\qquad\quad \vdots \\
\qquad e_n^{\#} = \mathcal{A}\, [\![\mathsf{e}_n]\!]\ \phi[\mathit{vars}_i \mapsto \top]
\end{array}
$$

Figure 16: An Abstract Semantics for Strictness Analysis on a Two-Point Domain

$$
\begin{array}{l}
(\odot) \quad :: \mathit{Int}^{\#} \rightarrow \mathit{Int}^{\#} \rightarrow \mathit{Int}^{\#} \\
\top \odot \top = \top \\
\top \odot \bot = \bot \\
\bot \odot \top = \bot \\
\bot \odot \bot = \bot
\end{array}
$$

We must also be able to combine results from different paths in a program. This requires both conjunction and disjunction. We can use the *meet* ($\sqcap$) and *join* ($\sqcup$) from our lattice which are fully described in Figure 15.

We can now define an abstract interpretation, $\mathcal{A}$, that takes expressions in our language and gives us their abstracted values. We require an environment that maps variables and functions to abstracted values, we use, $\phi :: \mathit{Env}^{\#}$ to represent this environment. We write $\phi[x \mapsto v]$ to represent extending the environment with identifier x being mapped to the value v. Lastly, looking up a value in the environment is just applying the environment to the identifier.

4.3.3 *Some Examples*

We can now use the abstract semantics from Figure 16 on some real functions.

The Constant Function:

The function *const* is defined as

$$const\ x\ y\ =\ x$$

For non-recursive functions, like *const*, we can determine the strictness properties fully with just n iterations of $\mathcal{A}$ where n is the number of arguments to the function. We run the abstract interpretation using $\bot$ as the value for the argument we are currently interested in and $\top$ for all the rest.

First we analyse the body (x) in terms of x:

$$\mathcal{A}\,[\![\mathrm{x}]\!]\,[x \mapsto \bot, y \mapsto \top] \Rightarrow \bot$$

then in terms of y:

$$\mathcal{A}\,[\![\mathrm{x}]\!]\,[x \mapsto \top, y \mapsto \bot] \Rightarrow \top$$

Remembering what it means to be strict from Equation 4.3.1, this analysis tells us that *const* is strict in x but not in y. This is exactly what we would expect.

□

Conditional

In non-strict languages we can define our own control-flow abstractions, allowing what is usually a primitive, the *if* statement, to be defined naturally in the language as

$$\begin{aligned} if\ p\ t\ e\ =\ &\mathbf{case}\ p\ \mathbf{of} \\ &\quad True \rightarrow t \\ &\quad False \rightarrow e \end{aligned}$$

Analysing *if* should determine that *if* is strict in p

$$\begin{aligned} &\mathcal{A}\,[\![\mathsf{Case}\ \mathrm{p}\ [\mathsf{True} \rightarrow \mathrm{t}, \mathsf{False} \rightarrow \mathrm{e}]]\!]\ \phi \\ &\quad \Rightarrow \mathcal{A}\,[\![\mathrm{p}]\!]\ \phi \sqcap (\mathcal{A}\,[\![\mathrm{t}]\!]\ \phi \sqcup \mathcal{A}\,[\![\mathrm{e}]\!]\ \phi) \\ &\quad \Rightarrow \bot \sqcap (\top \sqcup \top) \\ &\quad \Rightarrow \bot \\ &\quad \mathbf{where} \\ &\qquad \phi = [p \mapsto \bot, t \mapsto \top, e \mapsto \top] \end{aligned}$$

This shows how the use of *meet* and *join* are used to combine the results from the different branches of the function. Because the discriminant of a *Case* expression is always evaluated to WHNF, a non-terminating discriminant results in a non-terminating function.

Notice that if we analysed the second two arguments to *if* then we would have seen that they are not strict for *if* because only one would be $\bot$ at a given time and *meet* ($\sqcup$) only results in bottom if both arguments are bottom.

□

Calling Abstract Functions

Calling functions in the abstract interpretation is the same as a function call in the standard interpretation except that the target of the call is the abstracted function. Dependency analysis is used to ensure that callees are analysed before their callers.[8] The following example illustrates calling functions and the fact that the abstraction of *Case* is capable of determining when an argument is needed in all the branches.

$$\begin{aligned} \mathit{addOrConst}\ b\ x\ y \ &= \ \textbf{case}\ b\ \textbf{of} \\ &\qquad \mathit{True} \ \rightarrow \ x + y \\ &\qquad \mathit{False} \ \rightarrow \ x \end{aligned}$$

The analysis for the second argument proceeds as follows

$$\begin{aligned} &\mathcal{A}\,[\![\mathsf{Case}\ \mathsf{b}\ [\mathsf{True}\ \rightarrow\ \mathsf{x} + \mathsf{y},\ \mathsf{False}\ \rightarrow\ \mathsf{x}]]\!]\ \phi \\ &\quad \Rightarrow\ \mathcal{A}\,[\![\mathsf{b}]\!]\ \phi \sqcap (\mathcal{A}\,[\![\mathsf{x} + \mathsf{y}]\!]\ \phi \sqcup \mathcal{A}\,[\![\mathsf{x}]\!]\ \phi) \\ &\quad \Rightarrow\ \top \sqcap (\mathcal{A}\,[\![\mathsf{x} + \mathsf{y}]\!]\ \phi \sqcup \mathcal{A}\,[\![\mathsf{x}]\!]\ \phi) \\ &\quad \Rightarrow\ \top \sqcap ((\mathcal{A}\,[\![+]\!]\ \phi\ (\mathcal{A}\,[\![\mathsf{x}]\!]\ \phi)\ (\mathcal{A}\,[\![\mathsf{y}]\!]\ \phi)) \sqcup \top) \\ &\quad \Rightarrow\ \top \sqcap ((+^{\#}\ \bot\ \top) \sqcup \top) \\ &\quad \Rightarrow\ \top \sqcap (\bot \sqcup \top) \\ &\quad \Rightarrow\ \bot \\ &\quad \textbf{where} \\ &\qquad \phi\ =\ [b\ \mapsto\ \top, x\ \mapsto\ \bot, y\ \mapsto\ \top] \end{aligned}$$

The other language constructs are interpreted similarly. Do note that we do not attempt to analyse functions with free variables. Instead we take advantage of lambda-lifting in order to remove nested functions to the top level. We are not the first to use lambda lifting in order to avoid this problem [Clack and Peyton Jones, 1985]. Luckily, lambda lifting is done regardless for compilation to the G-Machine.

□

Recursive Functions

Our last concern is with recursive functions. We use the fact that recursive abstract functions are just recursive functions on a different

[8] This also identifies mutually recursive groups, which we explain next.

domain. This allows us to define a recursive function as the least upper bound of successive approximations of the function, i.e. an *ascending Kleene chain* (AKC).

We calculate this by starting with the 'bottom' approximation where all sets of inputs are mapped to $\perp$. For a function f we call this $f^{\#0}$. We then calculate $f^{\#n}$ by replacing each call to f with $f^{\#(n-1)}$. This series of successive approximations forms the AKC.

So for a function of the form

$$f\, x_1\, \ldots\, x_m \;=\; \langle \text{body of } f \text{ which includes a call to } f \rangle$$

We generate the following AKC

$$\begin{aligned}
f^{\#0}\, x_1\, \ldots\, x_m &= \perp \\
f^{\#1}\, x_1\, \ldots\, x_m &= \langle \text{body of } f \text{ with call to } f^{\#0} \rangle \\
f^{\#2}\, x_1\, \ldots\, x_m &= \langle \text{body of } f \text{ with call to } f^{\#1} \rangle \\
f^{\#3}\, x_1\, \ldots\, x_m &= \langle \text{body of } f \text{ with call to } f^{\#2} \rangle \\
&\vdots \\
f^{\#n}\, x_1\, \ldots\, x_m &= \langle \text{body of } f \text{ with call to } f^{\#(n-1)} \rangle
\end{aligned}$$

We can stop the calculation of this AKC when $f^{\#n} \equiv f^{\#(n-1)}$ for *all combinations* of x_1 to x_m. This means that for each iteration of the AKC we must interpret f 2^m times!

Fortunately, there are clever ways of avoiding much of this expense for typical cases. Clack and Peyton Jones developed an algorithm for the efficient calculation of an AKC [Clack and Peyton Jones, 1985].

□

Discussion of Two-Point Strictness Analysis

We have now seen how to use semantic function $\mathcal{A}$ from Figure 16 to analyse functions in our programs. Now we can see how the results help us in our ultimate aim of implicit parallelism. For this discussion we will forget that parallelism is not always beneficial and focus on how we would utilise all possible parallelism.

Imagine we have a function f with a call to g in its body.

$$f\, \ldots \;=\; \ldots\, g\, e_1\, e_2\, e_3\, \ldots$$

Our analysis may determine that g is strict in its first two arguments, providing us with an opportunity for parallelism. This would allow us to safely rewrite f as the following

```
f ... = ... let
              x = e1
              y = e2
            in
            x 'par' y 'par' g x y e3 ...
```

The expressions e_1 and e_2 are bound to the names x and y in a *let* expression so that the results of parallel evaluation are shared with the body of g.

As mentioned above, the two-point domain informs us about strictness up to WHNF. So if e_1 or e_2 are values from a flat domain, this transformation will provide all the benefits possible to g.[9] But what if these arguments are of a non-flat type, like pairs or lists?

Because we aim for safe parallelism, we cannot evaluate e_1 or e_2 any further than WHNF, already eliminating a vast quantity of potential parallelism. Take the function *sum* for example:

```
f ... = ... let
              xs = e1
            in
            xs 'par' sum xs ...
```

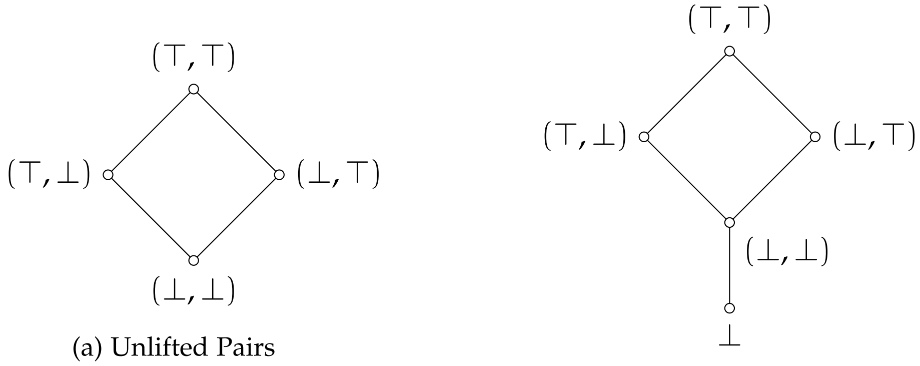
When a programmer wants to express this idiom in their code they often use *parallel strategies* as illustrated in Section 2.5. This allows the programmer to write an expression similar to *xs 'using'parList*. This forces the evaluation of the list beyond WHNF. Because our two-point domain does not guarantee the safety of evaluating beyond WHNF we are not able to use a strategy like *parList*. This means that even though we 'know' that *sum* requires the list fully, the analysis has no way to represent this, and can only determine that the outermost constructor is needed!

Because of this shortcoming, strictness analysis in this form is inappropriate for discovering the parallelism in a program that uses arbitrary algebraic data structures (e.g. lists or trees). In the next section we will see how this deficiency is overcome by choosing suitable domains for non-flat structures.

4.4 four-point forward analysis

As noted in the last section, two-point domains are quite limited when dealing with lazy structures. A more formal explanation for

[9] Again, ignoring the fact that it may not actually be a positive benefit.



(a) Unlifted Pairs

(b) Lifted Pairs

Figure 17: Domain for pairs of flat-domain values

this limitation is that the two-point domain really represents reduction to WHNF or $\perp$, and nothing else. In the case of flat domains this is sufficient because WHNF is all there is. For nested data types the reality is much different.

For functions that work on lists, like *sum* or *append*, strictness up to WHNF is not much benefit. Strictness analysis as described in the previous section would be able to tell us that *sum* requires its argument to be defined, allowing us to evaluate it before entering the function (or in parallel). But it is only safe up to WHNF. Once the first *Cons* is reached we must stop evaluation of the list or risk introducing non-termination. Through the lens of implicit parallelism it seems that we are unlikely to benefit from introducing parallel evaluation when we are limited to WHNF.[10] This is clearly a problem.

The solution seems clear: we must extend the abstract domains for non-flat data types so that we can have more detailed strictness information. For some data types, extending the technique is straightforward. In F-Lite, pairs can be defined as follows.

$$\textit{data Pair}\ \alpha\ \beta\ =\ \textit{MkPair}\ \alpha\ \beta$$

Many languages, such as Haskell, Clean, and the ML family provide the following syntactic sugar for pairs (and other N-tuples): (α, β).

A first try at representing pairs of values from the two-point domain could give us the lattice in Figure 17a.

The meaning of this lattice is fairly intuitive. When we possess a pair of flat-domain values there are four possibilities, we can have

[10] Indeed most uses of the basic strictness information were for improving the code generation to avoid building unnecessary suspensions.

1. The pair structure itself (*MkPair*), but accessing either value results in non-termination

2. The pair structure itself, but accessing the *fst* element results in non-termination

3. The pair structure itself, but accessing the *snd* element results in non-termination

4. The pair structure itself and both values are fully defined

Notice that possibilites 2 and 3 are similar in that there are one defined and one undefined item in each. Suggesting that one of 2 or 3 is more defined than the other would make little sense. For this reason we say that they are *incomparable*, i.e. they are neither more nor less defined than each other.

However, the lattice in Figure 17a is only valid for *unlifted* pairs, where the constructor value, $(,)$, is always defined. The reality for non-strict languages is that *any* value may be undefined, including the constructors for product types.

This means that we must *lift* the domain, adding a further bottom value that represents a failure to construct the pair's outermost constructor (*MkPair* in the F-Lite case). The result, shown in Figure 17b, is typical of domains for strictness analysis on finite types. You can construct an appropriate domain assuming that the structure is itself defined, then lift the resulting lattice with an additional $\perp$ that represents a failure to construct the structure.

Because this domain is still finite we are able to incorporate it into the framework developed by Mycroft without much issue, simply defining the appropriate *meet* and *join* on the lattice and the strictness properties for any primitives that work with pairs. The main issue is that extending the technique to non-flat domains in the obvious way introduces infinite domains for recursive types, losing a lot of the power of abstract interpretation.

The first practical solution was proposed by Wadler involving a four-point domain for lists [Wadler, 1987]. Instead of representing the recursive structure of lists directly, which creates an infinite domain, Wadler chose a domain that represents four degrees of definedness for lists.

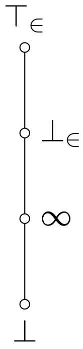


Figure 18: Wadler's Four-point Domain

The result, as shown in Figure 18, can be described, from least to most defined as follows:

1. $\bot$ represents all undefined lists
2. ∞ represents all undefined lists, lists with undefined tails and all infinite lists
3. $\bot_\in$ represents all of the above in addition to all finite lists with at least one undefined element
4. $\top_\in$ represents fully defined lists along with all of the above

Because we are now concerning ourselves with values from different domains in our analysis we must now know the types of expressions in our program. This ensures that we do not accidentally try to *meet* or *join* values from different domains.

To incorporate the four-point domain into the abstract interpretation from the previous section we need a few new primitives. The *Cons* constructor is given the abstract definition shown in Figure 19. *Nil*, being a fully defined list, is always abstracted as $\top_\in$. There are a few points worth mentioning about the definition of *cons*$^\#$. First, none of the equations result in $\bot$. This makes sense with our understanding of lazy evaluation, if we have the outermost constructor we

$$\begin{array}{llll}
cons^\# \; \top \; \top_\in = \top_\in & \qquad & cons^\# \; \bot \; \top_\in = \bot_\in \\
cons^\# \; \top \; \bot_\in = \bot_\in & \qquad & cons^\# \; \bot \; \bot_\in = \bot_\in \\
cons^\# \; \top \; \infty \; = \infty & \qquad & cons^\# \; \bot \; \infty \; = \infty \\
cons^\# \; \top \; \bot \; = \infty & \qquad & cons^\# \; \bot \; \bot \; = \infty
\end{array}$$

Figure 19: Definition of $cons^\#$ for a Four-Point Domain

$$
\begin{array}{l}
\mathcal{A}\,[\![\mathrm{Con\ c}]\!]\ \phi \\
\qquad \mid\ c\ ==\ \text{“}Nil\text{”} = \top_{\in} \\
\qquad \mid\ \mathit{otherwise} \quad\ \ = \top \\
\mathcal{A}\,[\![\mathrm{App\ (Con\ “Cons”)\ [x,\ xs]}]\!]\ \phi = \mathit{cons}^{\#}\ (\mathcal{A}\,[\![\mathrm{x}]\!]\ \phi)\ (\mathcal{A}\,[\![\mathrm{xs}]\!]\ \phi)
\end{array}
$$

Figure 20: Modification to $\mathcal{A}$ for List Constructors

have a value in WHNF and therefore it is definitely *not* $\perp$. Additionally, notice that *cons*ing a defined value onto $\perp_{\in}$ also results in $\perp_{\in}$, this keeps $\mathit{cons}^{\#}$ monotonic in addition to aligning with our intuitions (*cons*ing a defined element to the beginning of a list with possibly undefined elements does not suddenly make the list fully defined).

We must therefore alter the $\mathcal{A}$ rules for nullary constructors and add a pattern for when *Cons* is used. The modified *Con c* rule and the new rule for *Cons* are shown in Figure 20.

In addition to *Cons* and *Nil*, we need to define new interpretations for *Case* expressions. Pattern matching on a value from the four-point domain will require a different interpretation than the previous section. The new domain introduces two problems that must be dealt with:

1. Abstracting the alternatives of a *Case* expression naively can approximate too much, making the results less effective

2. Choosing appropriate approximations for the bindings introduced with the *Cons* alternative

We will show the solutions to these obstacles one at a time.

PROBLEM 1: The first point has to do with pattern matching and preventing the *Nil* alternative from weakening our approximations. For now we will ignore the question of how to approximate the bindings introduced with the *Cons* alternative.

Take the following template for pattern matching on lists:

```
case ⟨a list⟩ of
  Nil       → ⟨Nil branch⟩
  Cons x xs → ⟨Cons branch with possible occurrences of x and xs⟩
```

If we name the *Nil* branch $a^{\#}$ and treat the *Cons* branch as a function $f^{\#}$ of x and xs we have the following form

$$
\begin{array}{l}
\textbf{case } \langle \text{a list} \rangle \textbf{ of} \\
\quad \mathit{Nil} \qquad\quad \to a^{\#} \\
\quad \mathit{Cons}\ x\ xs \to f^{\#}\ x\ xs
\end{array}
$$

If we were to naively use the *Case* rule from the previous section we would have[11]

$$
\begin{array}{l}
\mathcal{A}\,[\![\mathsf{Case}\ \mathsf{xs}\ [\mathsf{Nil}\ \to\ \mathsf{e}_1,\ \mathsf{Cons}\ \mathsf{y}\ \mathsf{ys}\ \to\ \mathsf{e}_2]]\!]\ \phi = xs^{\#} \sqcap (a^{\#} \sqcup f^{\#}\ \top\ \top_{\in}) \\
\quad \textbf{where} \\
\qquad xs^{\#} = \mathcal{A}\,[\![\mathsf{xs}]\!]\ \phi \\
\qquad a^{\#} \;\, = \mathcal{A}\,[\![\mathsf{e}_1]\!]\ \phi \\
\qquad f^{\#} \;\, = [ys][y](\mathcal{A}\,[\![\mathsf{e}_2]\!]\ \phi)
\end{array}
$$

The issue is that the *meet*ing of $a^{\#}$ with $f^{\#}\ y\ ys$ will often prevent the analysis from providing useful information. Take the function *sum* for example:

$$
\begin{array}{lcl}
\mathit{sum}\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad \mathit{Nil} \qquad\quad \to 0 \\
 & & \quad \mathit{Cons}\ y\ ys \to y + \mathit{sum}\ ys
\end{array}
$$

In this case, $a^{\#}$ will always be $\top$, when we abstract the function to get $xs^{\#} \sqcap (\top \sqcup f^{\#}\ y\ ys)$ the result would only ever be $\bot$ if $xs \equiv \bot$. Therefore it is only safe for us to evaluate the list up to WHNF, when *sum* clearly needs a fully-defined list. Moreover, because we may lack information about the definedness of xs we must be safe and approximate y and ys to the top of their lattices ($\top$ and $\top_{\in}$, respectively).

Wadler's key insight was that the use of pattern matching allowed us to retain information that would otherwise be lost when performing abstract interpretation [Wadler, 1987]. Whereas in our previous abstract interpretation from Section 4.3 we had to *join* ($\sqcup$) all of the branches in a *Case* expression, we can now use the fact that we know *Nil* is always the $\top_{\in}$ value in our domain. Why is this? Because *Nil* is a fully defined list with *no bottom elements*!

This means that we only have to consider the value of $a^{\#}$ when our *Case* expression matches on $\top_{\in}$. This prevents the definedness of $a^{\#}$ from preventing more accurate approximations for when the list is less defined than $\top_{\in}$, solving our first issue.

[11] We use the notation found in [Turner, 2012] for abstracting a variable from an expression: $[x]e$ denotes abstracting occurrences of x out of e. This results in a function equivalent to $(\lambda x \to e)$.

$$
\begin{array}{l}
\mathcal{A}\,[\![\mathrm{Case}\; xs\; [\mathrm{Nil} \;\to\; e_1,\; \mathrm{Cons}\; y\; ys \;\to\; e_2]]\!]\;\phi \\
\quad \begin{array}{lll}
\mid xs^\# == \top_\in & = & a^\# \sqcup f^\#\;\top\;\top_\in \\
\mid xs^\# == \bot_\in & = & f^\#\;\bot\;\top_\in \sqcup f^\#\;\top\;\bot_\in \\
\mid xs^\# == \infty & = & f^\#\;\top\;\infty \\
\mid \mathit{otherwise} & = & \bot
\end{array} \\
\quad \textbf{where} \\
\qquad \begin{array}{lll}
xs^\# & = & \mathcal{A}\,[\![xs]\!]\;\phi \\
a^\# & = & \mathcal{A}\,[\![e_1]\!]\;\phi \\
f^\# & = & [ys][y](\mathcal{A}\,[\![e_2]\!]\;\phi)
\end{array}
\end{array}
$$

Figure 21: Abstraction of Case Expressions on Lists

PROBLEM 2: The second problem was choosing appropriate approximations for the bindings introduced with the *Cons* alternative. Happily, this turns out to be quite easy to solve.

In our two-point analysis all bindings introduced by an alternative to a *Case* expression are approximated by $\top$ because we do not 'know' how to approximate non-flat structures. When evaluating a *Case* on our four-point domain we can use the knowledge we have of what lists each point in the domain corresponds to. We can use the definition of the primitive $cons^\#$ as a lookup table, switching the right hand and left hand sides of each equation. This gives us the following:

$$
\begin{array}{ll}
\top_\in & \to cons^\#\;\top\;\top_\in \\
\bot_\in & \to (cons^\#\;\bot\;\top_\in) \sqcup (cons^\#\;\top\;\bot_\in) \sqcup (cons^\#\;\bot\;\bot_\in) \sqcup \\
\infty & \to (cons^\#\;\top\;\infty) \sqcup (cons^\#\;\top\;\bot) \sqcup (cons^\#\;\bot\;\infty) \sqcup (cons^\#\;\bot\;\bot)
\end{array}
$$

The absence of a rule for $\bot$ is due to the fact that we would never match on an undefined value, resulting in $\bot$ regardless of the values of the alternatives. We can also remove several of the alternatives in the cases for ∞ and $\bot_\in$ due to the necessity for the abstraction of the alternative branches to be monotonic [Wadler, 1987].[12]

Taking these insights into account leaves us with the rule for pattern matching on lists seen if Figure 21.

meet and *join* are easily defined for the four-point domain. If we assign each point in the domain a value according to its position in the lattice, with $\bot$ being 0 and $\top_\in$ being 3, we can define *meet* ($\sqcap$) as *min* and *join* ($\sqcup$) as *max*.

[12] For example, with $\bot_\in$ we will always have $f^\#\bot\bot_\in \sqsubseteq f^\#\top\bot_\in$, making $f^\#\bot\bot_\in$ unnecessary since $x \sqcup y \equiv y$ when $x \sqsubseteq y$. Applying this reasoning to ∞ leaves us with only $f^\#\top\infty$ to consider.

With everything in place, we can now determine whether an analysis using this four-point domain is more suitable for implicit parallelism.

Length and Sum

We will use the simple recursive *length* function for our first example of using this analysis (*sum* is defined similarly, replacing the 1 with *y*).

```
length xs = case xs of
                Nil        → 0
                Cons y ys → 1 + length ys
```

Because *length* and *sum* take only one argument, which is a list, we must analyse the functions at each of the four points in our domain for lists. Recursion can be dealt with in the same manner as shown in Section 4.3. Once a fixed-point is reached we are left with the following results.

$xs^\#$	$length^\#\ xs^\#$	$sum^\#\ xs^\#$
$\top_\#$	$\top$	$\top$
$\bot_\#$	$\top$	$\bot$
∞	$\bot$	$\bot$
$\bot$	$\bot$	$\bot$

Table 2: Analysis of $\mathtt{length}^\#$ and $\mathtt{sum}^\#$ Using 4-point Domains

The results in Table 2 are exactly what we would expect. If *length* or *sum* are passed infinite lists then program will result in non-termination. *sum* has the additional constraint that all *elements* of its input list must also be defined. This analysis would allow us to evaluate the argument to *sum* fully, in parallel, making it a significant improvement to the simple two-point analysis from Section 4.3.

Discussion of Four-Point Strictness Analysis

Because lists are one of the most common structures in functional programming, this development allowed strictness analysis to be useful in a wide variety of 'real' systems. This also makes strictness analysis' use for parallelism more realistic. We can now tell the machine to evaluate lists in parallel up to the degree that it is safe to do so. Some of the more successful attempts at implicit parallelism were based on using this strictness information, most notably Burn's work on parallelisation of functional programs for a Spineless G-Machine [Burn, 1987] and the work of Hogen et al. [1992] on automatically parallelising programs for a distributed reduction machine.

While this four-point domain made strictness analysis much more flexible it suffers from a few considerable shortcomings:

1. An argument is only considered strict for a function if it is strict in *all* possible contexts[13] of that function

2. For other structures similar domains must be *designed*, i.e. there does not seem to be straightforward way to derive a 'good' finite domain for every recursive type

3. Even when the compiler writer designs additional abstract domains for other recursive types, the calculation of fixed points becomes prohibitively expensive with more complex abstract domains

To illustrate the first problem we can study the results of applying this analysis to the *append* function, which can be seen in Table 3.

		$ys^\#$			
		$\top_\in$	$\bot_\in$	∞	$\bot$
$xs^\#$	$\top_\#$	$\top_\in$	$\bot_\in$	∞	∞
	$\bot_\#$	$\bot_\in$	$\bot_\in$	∞	∞
	∞	∞	∞	∞	∞
	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table 3: Analysis of $append^\#$ $xs^\#$ $ys^\#$ Using 4-point Domains

We can see that while the first list is always strict up to WHNF, the second list is not strict. This is unfortunate because we know that *append* is strict in both arguments under certain conditions.

For example, if we pass the result of *append* to *length* then we know that *both* argument lists for *append* must be finite for the result of the call the *length* to terminate. The inability for this analysis to express that form of strictness is a major weakness.

The limitations due to this first point were well known at the time of Wadler's paper on the four-point domain. However, the solutions seemed ad-hoc and were on shaky theoretical grounds [Hughes, 1985, 1987]. The introduction of the four-point domain was successful, in part, due to it fitting naturally in the strictness analysis techniques

[13] In other words, an argument for a function is only strict if it is strict for all possible uses of that function. We explore this further in the next section.

that were already understood. Fortunately, we have the benefit of time and work on the analysis of strictness that takes into account the *use* of a function, using *projections*, is much better understood [Hinze, 1995; Sergey et al., 2014].

The need to design a suitable domain for each recursive type is unfortunate. Ideally the strictness analysis in a compiler would work on whatever types the programmer decides to define. Functional languages are often lauded for their ability to have few primitive types and allow the programmer to define their own 'first class' types. Having strictness analysis that only functions well on lists subverts this ideal, creating a leaky abstraction. Programmers will use lists even when inappropriate because the compiler is so much better at optimising them than any custom type. While not motivated by the second issue, projection-based analysis solves it anyway, allowing strictness analysis to be performed on arbitrary types with very few restrictions.

As for the third shortcoming, projection-based analysis does not make calculating fixed points free. It does however shift the complexity of the analysis. Instead of being exponential in the number of arguments, it grows relative to the size of a function's *return* type. While not a panacea in this regard it does make projection-based analysis practical.

Overall, strictness analysis using the four-point domain is a significant improvement over a basic two-point domain, particularly for use in exploiting implicit parallelism. While having solved several of the downsides of using a simple two-point domain, the four-point analysis still suffers from significant problems when taking our use-case into account.

4.5 PROJECTION-BASED ANALYSIS

The shortcomings of the analyses based on the abstract interpretation of programs motivated Wadler and Hughes to propose using *projections* from domain theory to analyse strictness [Wadler and Hughes, 1987].

For our purposes projection-based analysis provides two benefits over abstract interpretation: simple formulation of domains to analyse functions over arbitrary structures, and a correspondence with parallel strategies [Marlow et al., 2010; Trinder et al., 1998]. This allows us to use the projections provided by our analysis to produce an appropriate function to compute the strict arguments in parallel.

We can frame the differences in the two approaches by thinking about what each analysis is answering. Strictness analysis by abstract interpretation asks “When passing $\perp$ as an argument, is the result of the function call $\perp$?”. Projection-based strictness analysis instead asks “If there is a certain degree of demand on the result of this function, what degree of demand is there on its arguments?”.

What is meant by ‘demand’? As an example, the function *length* requires that the input list be finite, but no more. We can therefore say that *length demands* the spine of the argument list. The function *append* is a more interesting example:

```
append             :: [α] → [α] → [α]
append [] ys       = ys
append (x : xs) ys = x : append xs ys
```

As mentioned in the previous section the first argument must be defined to the first cons, but we cannot know whether the second argument is ever needed. However, what if the calling context of *append* requires the *result* of *append* to be a finite list? For example:

```
lengthOfBoth       :: [α] → [α] → Int
lengthOfBoth xs ys = length (append xs ys)
```

In this case *both* arguments to *append* must be finite. Projections can be used to formalise this type of context [Wadler and Hughes, 1987; Hinze, 1995], which we call a *demand context*.

Demand Context

> The depth of a structure that is needed by the consumer of a function’s result.

Demand Contexts allow us to reason about the various *uses* of a function’s result, letting us reason about functions like *append* more accurately. This, combined with their ability to analyse functions of arbitrary types without the need to design abstract domains by hand, make projection-based analysis the most realistic for our purposes of utilising implicit parallelism.

4.5.1 *Semantics of Projections*

Given a domain D, a projection on D is a continuous function $\pi : D \to D$ that satisfies

$$\pi \sqsubseteq \mathrm{ID} \tag{2}$$

$$\pi \circ \pi = \pi \tag{3}$$

Equation (2) ensures that a projection can not add any information to a value, i.e. all projections approximate the identity function. Idempotence (3) ensures that projecting the same demand twice on a value has no additional effect. This aligns with our intuition of demand. If we demand that a list is spine-strict, demanding spine-strictness again does not change the demand on the list.

Because we want the introduction of parallelism to be semantics-preserving we use the following safety condition for projections:

$$\gamma \circ f = \gamma \circ f \circ \pi \tag{4}$$

Given a function $f : X \to Y$, and demand γ on the *result* of f we wish to find a safe π such that the equality in Equation (4) remains true. Projection-based analysis propagates the demand given by γ to the arguments of f. This results in the demand on the *arguments* of f given by π. The analysis aims to find the *smallest* π for each γ, but approximating towards ID (as it is always safe to project the identity).

DEMANDS ON PRIMITIVES On unlifted base types, such as unboxed integers, there are two demands, ID and BOT, with the following semantics

$$\mathrm{ID}\, x = x \tag{5}$$

$$\mathrm{BOT}\, x = \bot \tag{6}$$

When an expression is in a BOT context it means that non-termination is inevitable. You can safely evaluate an expression in this context because there is no danger of *introducing* non-termination that is not already present.

DEMANDS ON LIFTED TYPES Haskell's non-strict semantics means that most types we encounter are *lifted* types. Lifted types represent possibly unevaluated values. Given a demand π on D, we can form two possible demands on $D_\bot$, $\pi!$ and $\pi?$; strict lift and lazy lift respectively. To paraphrase Kubiak et al.: $\pi!$ means we will definitely need

$\pi ::= \mathsf{BOT}$	Bottom (hyperstrict)
$\mid \mathsf{ID}$	Top (the identity)
$\mid \langle \pi_1 \otimes \pi_2 \cdots \otimes \pi_n \rangle$	Products
$\mid [C_1\ \pi_1 \mid C_2\ \pi_2 \ldots \mid C_n\ \pi_n]$	Sums
$\mid \mu\beta.\pi$	Recursive Demands
$\mid \beta$	Recursive Binding
$\mid \pi?$	Strict Lift
$\mid \pi!$	Lazy Lift

Figure 22: Abstract Syntax for Contexts of Demand

the value demanded by this projection, and we will need π's worth of it [Kubiak et al., 1992]. $\pi?$ does not tell us whether we need the value or not, but if we *do* need the value, we will need it to satisfy π's demand.

DEMANDS ON PRODUCTS A projection representing a demand on a product can be formed by using the $\otimes$ operator with the following semantics

$$\begin{aligned} \langle \pi_1 \otimes \cdots \otimes \pi_n \rangle\ \bot &= \bot \\ \langle \pi_1 \otimes \cdots \otimes \pi_n \rangle\ \langle x_1, \ldots, x_n \rangle &= \langle \pi_1 x_1, \ldots, \pi_n x_n \rangle \end{aligned}$$

DEMANDS ON SUMS If projections are functions on a domain, then $|$, the operator that forms projections on sum types performs the case-analysis. Each summand is tagged with the constructor it corresponds to. Sometimes we will omit the constructor name when presenting projections on types with a single summand (such as anonymous tuples).

$$\begin{aligned} [\mathsf{True}\ \mathsf{ID} \mid \mathsf{False}\ \mathsf{BOT}]\ \mathsf{True} &= \mathsf{True} \\ [\mathsf{True}\ \mathsf{ID} \mid \mathsf{False}\ \mathsf{BOT}]\ \mathsf{False} &= \bot \end{aligned}$$

Figure 22 presents a suitable abstract syntax for projections representing demand. This form was introduced by Kubiak et al. and used in Hinze's work on projection-based analyses [Kubiak et al., 1992; Hinze, 1995]. We have omitted the details on the representation of context variables (for polymorphic demands). For a comprehensive discussion we suggest Chapter 6 of Hinze's dissertation [Hinze, 1995].

In short, projections representing demand give us information about how defined a value must be to satisfy a function's demand on that value. Knowing that a value is definitely needed, and to what degree, allows us to evaluate the value before entering the function.

Example Projections

Because our primitive values can be modelled by a flat domain (just ID and BOT), our lattice of projections corresponds with the two-point domain used in abstract interpretation.

□

For pairs of primitive values, possible contexts include:

$$[\langle \mathrm{ID}? \otimes \mathrm{ID}? \rangle] \tag{7}$$

$$[\langle \mathrm{ID}! \otimes \mathrm{ID}? \rangle] \tag{8}$$

As Haskell's types are sums of products, pairs are treated as sums with only one constructor. For product types each member of the product is lifted. Context 7 is the top of the lattice for pairs, accepting all possible pairs. Context 8 requires that the first member be defined but does not require the second element. This is the demand that *fst* places on its argument.

□

For polymorphic lists there are 7 principal contexts[14] [Hinze, 1995]; 3 commonly occurring contexts are:

$$\mu\beta.[\mathtt{Nil}\ \mathrm{ID}|\mathtt{Cons}\ \langle \gamma? \otimes \beta? \rangle] \tag{9}$$

$$\mu\beta.[\mathtt{Nil}\ \mathrm{ID}|\mathtt{Cons}\ \langle \gamma? \otimes \beta! \rangle] \tag{10}$$

$$\mu\beta.[\mathtt{Nil}\ \mathrm{ID}|\mathtt{Cons}\ \langle \gamma! \otimes \beta! \rangle] \tag{11}$$

Here μ binds the name for the 'recursive call' of the projection and γ is used to represent an appropriate demand for the element type of the list. An important point is that this representation for recursive contexts restricts the representable contexts to *uniform projections*: projections that define the same degree of evaluation on each

[14] All possible demands on polymorphic lists are instances of one of the 7 principal contexts.

ID: accepts all lists

T (tail strict): accepts all finite lists

H (head strict): accepts lists where the head is defined

HT: accepts finite lists where every member is defined

Figure 23: Four contexts on lists as described in [Wadler and Hughes, 1987].

---

of their recursive components as they do on the structure as a whole. The detailed reason for this restriction is given on page 89 of Hinze [1995]. This limitation does not hinder the analysis significantly as many functions on recursive structures are themselves uniform.

With this in mind, Context 9 represents a lazy demand on the list, Context 10 represents a *tail strict* demand, and Context 11 represents a *head and tail* strict demand on the list.

□

It will be useful to have abbreviations for a few of the contexts on lists. These abbreviations are presented in Figure 23.

We can now say more about the strictness properties of *append*. The strictness properties of a function are presented as a *context transformer* [Hinze, 1995].

$$\begin{array}{lcl}
\mathrm{append}(\mathrm{ID}) & \to & \mathrm{ID}!;\mathrm{ID}? \\
\mathrm{append}(\mathrm{T}) & \to & \mathrm{T}!;\mathrm{T}! \\
\mathrm{append}(\mathrm{H}) & \to & \mathrm{H}!;\mathrm{H}? \\
\mathrm{append}(\mathrm{HT}) & \to & \mathrm{HT}!;\mathrm{HT}!
\end{array}$$

This can be read as "If the demand on the result of *append* is ID then the first argument is strict with the demand ID and the second argument is lazy, but if it *is* needed, it is with demand ID.

□

Following Hinze [Hinze, 1995] we construct projections for every user-defined type. Each projection represents a specific strategy for evaluating the structure, as we shall define in section 5.2. This provides us with the ability to generate appropriate parallel strategies for arbitrary types.

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\mathsf{BOT} \,\&\, \gamma & = & \mathsf{BOT} & \mathsf{BOT} \sqcup \gamma & = & \gamma \\
\gamma \,\&\, \mathsf{BOT} & = & \mathsf{BOT} & \gamma \sqcup \mathsf{BOT} & = & \gamma \\
\mathsf{ID} \,\&\, \mathsf{ID} & = & \mathsf{ID} & \mathsf{ID} \sqcup \mathsf{ID} & = & \mathsf{ID} \\
 & & & & & \\
\alpha! \,\&\, \gamma! & = & (\alpha \,\&\, \gamma)! & \alpha! \sqcup \gamma! & = & (\alpha \sqcup \gamma)! \\
\alpha! \,\&\, \gamma? & = & (\alpha \sqcup \alpha \,\&\, \gamma)! & \alpha! \sqcup \gamma? & = & (\alpha \sqcup \gamma)? \\
\alpha? \,\&\, \gamma! & = & (\alpha \,\&\, \gamma \sqcup \gamma)! & \alpha? \sqcup \gamma! & = & (\alpha \sqcup \gamma)? \\
\alpha? \,\&\, \gamma? & = & (\alpha \sqcup \gamma)? & \alpha? \sqcup \gamma? & = & (\alpha \sqcup \gamma)?
\end{array}
$$

Figure 24: Conjunction & and Disjunction ⊔ for Projections on Basic and Lifted Values

4.5.2 *Lattice of Projections*

Having an intuition of what projections are we can now define how we combine differing demands on values. In the previous analyses we used the *meet* (⊓) and *join* (⊔) operations directly. Projections also have *meet* (⊓) and *join* (⊔) , but because our projections are representing demand contexts we do not actually want to use *meet* (⊓) . Instead we use &, where $\alpha \,\&\, \gamma$ represents the *joint* demand of both α's and γ's demand taken together. In other words, the projection $\alpha \,\&\, \gamma$ only accepts values that are accepted by α and γ, and returns $\perp$ otherwise (motivating the use of 'conjunction' to describe the operation). Using & instead of ⊓ is standard when dealing with projections that act on demands [Wadler and Hughes, 1987; Hinze, 1995; Sergey et al., 2014].

Figure 24 shows the rules for performing conjunction and disjunction of projections on basic values (either BOT or ID) and for lifted values. Note that when we perform & on two projections with different lifts we must ensure that the resulting projection is not more strict than the strictly lifted input, this ensures that we maintain our desired safety condition.

Figure 25 shows the same operations for projections on sum and product types. The only surprising aspect of the definitions is that we are forced to normalise the result of a conjunction on product types. This is because it possible for & to form a projection denoting BOT even when both arguments are not BOT. For example, applying & to a projection that only accepts True and a projection that only accepts False results in the BOT projection. This is because there is

$$
\begin{aligned}
&[C_1\alpha_1|\ldots|C_n\alpha_n]\,\&[C_1\gamma_1|\ldots|C_n\gamma_n] \\
&\qquad\qquad = [C_1(\alpha_1\,\&\,\gamma_1)|\ldots|C_n(\alpha_n\,\&\,\gamma_n)] \\
&[C_1\alpha_1|\ldots|C_n\alpha_n]\sqcup[C_1\gamma_1|\ldots|C_n\gamma_n] \\
&\qquad\qquad = [C_1(\alpha_1\sqcup\gamma_1)|\ldots|C_n(\alpha_n\sqcup\gamma_n)]
\end{aligned}
$$

$$
\begin{aligned}
&\langle\alpha_1\otimes\cdots\otimes\alpha_n\rangle\,\&\langle\gamma_1\otimes\cdots\otimes\gamma_n\rangle \\
&\qquad\qquad = \mathrm{norm}\big(\langle(\alpha_1\,\&\,\gamma_1)\otimes\cdots\otimes(\alpha_n\,\&\,\gamma_n)\rangle\big) \\
&\langle\alpha_1\otimes\cdots\otimes\alpha_n\rangle\sqcup\langle\gamma_1\otimes\cdots\otimes\gamma_n\rangle \\
&\qquad\qquad = \langle(\alpha_1\sqcup\gamma_1)\otimes\cdots\otimes(\alpha_n\sqcup\gamma_n)\rangle
\end{aligned}
$$

Figure 25: Conjunction and Disjunction for Projections on Products and Sums

no possible Boolean value that the resulting projection will accept, despite neither constituent projection denoting BOT.

The *norm* function recognises these projections and transforms them to the direct representation of BOT.[15]

4.5.3 *Recursive Types*

For conjunction of projections on recursive types we have to perform additional analysis to ensure that we maintain uniformity, which is the property that the demand on the 'recursive call' of the type is equal to the demand on the type itself (as mentioned in Section 4.5.1). The subtlety is due to the fact that performing conjunction on two recursive types might result in demands that differ on the 'head' of the value from the demand on the recursive call [Kubiak et al., 1992; Hinze, 1995].

A simple example is when we perform conjunction on the *H* and *T* projections on lists (from Figure 23). If we naively perform conjunction as $(\mu\beta.\alpha)\ \&\ (\mu\beta.\gamma) = \mu\beta.\alpha\ \&\ \gamma$, we arrive at *HT*, while this may seem like the correct result it is actually unsafe! This is clear when applying these projections to the list $xs = 1 : \bot : []$

[15] norm is defined in Hinze [1995] Section 6.3.

$$
\begin{array}{l}
(\mu\beta.\alpha) \sqcup (\mu\beta.\gamma) = \ \mu\beta.\alpha \sqcup \gamma \\
(\mu\beta.\alpha)\ \&(\mu\beta.\gamma) \begin{array}[t]{l@{\ }l}
|\ \mathit{conj} \sqsubseteq \beta_1\ \&'\ \beta_2 & = \mathit{norm}\,(\mu\beta.\alpha\ \&\ \gamma\,) \\
|\ \mathit{conj} \sqsubseteq \beta_1 \sqcup'\ \beta_1\ \&'\ \beta_2 & = \mu\beta.\alpha \sqcup \alpha\ \&\ \gamma \\
|\ \mathit{conj} \sqsubseteq \beta_1\ \&'\ \beta_2 \sqcup'\ \beta_2 & = \mu\beta.\alpha\ \&\ \gamma \sqcup \gamma \\
|\ \mathit{conj} \sqsubseteq \beta_1 \sqcup'\ \beta_2 & = \mu\beta.\alpha \sqcup \gamma
\end{array} \\
\quad \textbf{where} \\
\qquad \mathit{conj} = \ (\mathit{norm}(\alpha[\beta\ \mapsto\ \beta_1]))\ \&'\ (\mathit{norm}(\gamma[\beta\ \mapsto\ \beta_2]))
\end{array}
$$

Figure 26: Conjunction and Disjuntion for Projections on Recursive Types

$$
\begin{array}{lll}
H\,xs & \equiv 1\ :\ \_ & \text{-- Head strictness forces the first element} \\
T\,xs & \equiv \_\ :\ \_\ :\ [\,] & \text{-- Tail strictness forces the spine} \\
HT\,xs & \equiv \bot & \text{-- } HT \text{ forces all elements and the spine}
\end{array}
$$

The reason for the differing results is that H is not strict in the recursive part of the list, but T is, and being head strict is *not* the same as requiring all elements of a structure, as evidenced by the following small program

$$
\begin{array}{l}
f\,xs\ = a + b \\
\quad \textbf{where} \\
\qquad a = \mathit{head}\,xs \\
\qquad b = \mathit{length}\,xs
\end{array}
$$

The demand on the input list *xs* is the conjunction of the demands for *head* and *length* but it is clear to see that *f* does not require its input list to be fully defined, making it unsafe for $H\ \&\ T$ to result in HT.

□

The simplest way to maintain uniformity is to take the least upper bound ($\sqcup$) of the two projections. However, this would be too conservative and we would lose out on some strictness information that is present. Fortunately, Hinze provides us with a method that allows us to use more accurate approximations when it is safe to do so, relying on *join* ($\sqcup$) only when necessary (the last guard in Figure 26). The idea is to use versions of $\sqcup$ and & that ignore all demands except those on the recursive calls; these are written as $\sqcup'$ and $\&'$. We then see where in the lattice the result (*conj*) resides, performing the corresponding approximation, defaulting to the always safe $\mu\beta.\alpha \sqcup \gamma$. For details on how this method was derived and a proof of its safety, see Hinze [1995] Section 6.4.

We are now able to present the analysis itself. Being a backward analysis means that the *result* of our analysis is an environment that maps variables to demand contexts on those variables. Disjunction and conjunction on these environments simply performs the operations on the elements in the environment with the same key. If a key is not present in an environment it is equivalent to having the lazy demand (top of the demand context lattice for its type).

In order to understand the rules in Figure 27 we must introduce a few small operators. The ↓ operator takes a projection on sum types and a constructor tag and returns the projection corresponding to that constructor:

$$[C_1\ \alpha_1 \mid \ldots \mid C_i\ \alpha_i \mid \ldots \mid C_n\ \alpha_n] \downarrow C_i \;=\; \alpha_i$$

The ↑ operator performs the dual operation, injecting a projection on one constructor into a projection on the corresponding sum type. In the equation below BOT? (also known as the *absent* demand) represents the lazy lift of the bottom projection for the corresponding sum type:

$$C_i \uparrow \pi \;=\; [C_1\ BOT? \mid \ldots \mid C_i\ \pi \mid \ldots \mid C_n\ BOT?]$$

When analysing expressions wrapped in *Freeze* we have to be careful because any demands that originate from suspended values are not *strict* demands. The *guard* (▷) operator accomplishes this:

$$! \rhd \pi = \pi$$
$$? \rhd \pi = \pi \sqcup abs(typeOf(\pi))$$

The $abs(typeOf(\pi))$ above gets the absent demand for the corresponding type of the projection π.

The *wrap* and *unwrap* functions ensure that we handle recursive types appropriately. *unwrap* is used to 'unwrap' the recursive knot one level, so that

$$unwrap\ \alpha@(\mu\beta.[Nil\ \pi_1 \mid Cons\ \langle\ \pi_2 \otimes \beta_\ell\ \rangle]) \;=$$
$$[Nil\ \pi_1 \mid Cons\ \langle\ \pi_2 \otimes \alpha_\ell\ \rangle]$$

wrap is the inverse operation, retying the recursive knot [Hinze, 1995, pg. 117].

Lastly, *getProd* takes a list of identifiers and a projection environment and returns the projection on the product made up by those identifiers.

```
$\mathcal{P} :: Exp \to FunEnv^{\#} \to Context \to Env^{\#}$
$\mathcal{P}\,[\![\mathsf{Var}\ v]\!]\ \phi\ \pi \qquad = \{v \mapsto \pi\}$
$\mathcal{P}\,[\![\mathsf{Int}\ i]\!]\ \phi\ \pi \qquad = \emptyset$
$\mathcal{P}\,[\![\mathsf{Con}\ c]\!]\ \phi\ \pi \qquad = \emptyset$
$\mathcal{P}\,[\![\mathsf{Freeze}\ e]\!]\ \phi\ \pi_{\ell} \qquad = \ell \rhd \mathcal{P}\,[\![e]\!]\ \phi\ \pi$
$\mathcal{P}\,[\![\mathsf{Unfreeze}\ e]\!]\ \phi\ \pi \qquad = \mathcal{P}\,[\![e]\!]\ \phi\ \pi!$
$\mathcal{P}\,[\![\mathsf{App}\ (\mathsf{Con}\ c)\ as]\!]\ \phi\ \pi$
      $\mid null\ as \quad = \emptyset$
      $\mid otherwise = overList\ \phi\ (unwrap(\pi) \downarrow c)\ as$
$\mathcal{P}\,[\![\mathsf{App}\ (\mathsf{Fun}\ f)\ as]\!]\ \phi\ \pi$
      $\mid null\ as \quad = \emptyset$
      $\mid otherwise = overList\ \phi\ (\phi\ f\ \pi)\ as$
$\mathcal{P}\,[\![\mathsf{Let}\ b\ =\ e_1\ \mathsf{in}\ e]\!]\ \phi\ \pi = env$
  where
      $\rho \ \ = \mathcal{P}\,[\![e]\!]\ \phi\ \pi$
      $\rho' \ = \rho \setminus \{b\}$
      $env = $ case $lookup\ b\ \rho$ of
                   $Nothing \to \rho'$
                   $Just\ \gamma_{\ell} \ \to \rho'\ \&\ (\ell \rhd \mathcal{P}\,[\![e_1]\!]\ \phi\ \gamma)$
$\mathcal{P}\,[\![\mathsf{Case}\ e\ [C_1\ cs_1 \to e_1,\ \ldots C_n\ cs_n \to e_n]]\!]\ \phi\ \pi$
                        $= \rho'_1\ \&\ \mathcal{P}\,[\![e]\!]\ \phi(wrap(C_1 \uparrow \pi_1))$
                          $\sqcup \cdots \sqcup$
                          $\rho'_n\ \&\ \mathcal{P}\,[\![e]\!]\ \phi(wrap(C_n \uparrow \pi_n))$
  where
      $\rho_1 \qquad = \mathcal{P}\,[\![e_1]\!]\ \phi\ \pi$
            $\vdots$
      $\rho_n \qquad = \mathcal{P}\,[\![e_n]\!]\ \phi\ \pi$

      $(\rho'_1, \pi_1) = (\rho_1 \setminus cs_1,\ getProd\ cs_1\ \rho_1)$
            $\vdots$
      $(\rho'_n, \pi_n) = (\rho_n \setminus cs_n,\ getProd\ cs_n\ \rho_n)$
```

Figure 27: Projection-based Strictness Analysis

4.6 SUMMARY

This chapter explored the most common static analyses used for strictness analysis. Because the initial placement of parallelism is crucial to our technique we explored each possible analysis in depth, highlighting the drawbacks of the two-point and four-point analyses. While projection-based analysis is significantly more complex, its ability to determine the strictness properties of functions based on the *demand* placed on their results provides too many benefits to ignore.

Many of the previous attempts and using strictness analysis for implicit parallelism used the four-point analysis. Combined with *evaluation transformers* (discussed in the next section) the four-point analysis is able to identify a significant amount of parallelism. Unfortunately, the analysis is limited to functions on lists, which while ubiquitous, significantly restricts the potential of identifying parallelism in more complex programs.

The projection-based analysis not only provides more insight into functions like *append*, as discussed in Section 4.5.1, but it also allows us to determine a useful set of strictness properties for functions on arbitrary types. This greatly expands the applicability of the strictness analysis for finding potential parallelism.

The major drawback of Hinze's projection-based analysis is that we, as compiler writers, no longer know in advance the set of demands our analysis will return. With the four-point analysis we can hardcode the parallel Strategies that correspond to each of the points in the domain. If we expand our analysis to include pairs, we can again add the corresponding strategies. With the projection-based analysis we no longer have that foresight. This is the issue we address in the next chapter.

5 DERIVATION AND USE OF PARALLEL STRATEGIES

> *It would be premature to claim that* `par` *and Strategies are redundant; [...]. Still, the difficulties with* `par` *indicate that it may be more appropriate as a mechanism for automatic parallelisation, than as a programmer-level tool.*
>
> – Marlow et al. [2011]

The information that projection-based strictness analysis provides us is concerned with how defined a value must be for a function to be defined given a certain demand on the result of that function. This is reflected in the safety condition for this analysis, which we remind ourselves of below:

$$\gamma \circ f = \gamma \circ f \circ \pi \tag{12}$$

The projection-based analysis attempts to determine the smallest π that retains the semantics of f for a given γ. This tells us which arguments, if any, are safe to evaluate before entering f. Our goal now is to take a given π and transform a call to f so that as much evaluation as π allows is done in parallel.

This chapter presents our method of achieving this goal automatically. We describe this process as the *derivation* of parallel strategies from projections representing demand on a value, and it forms a core part of our contribution.

Plan of the Chapter

We discuss some parallels with Burn's *Evaluation Transformers* in Section 5.1 which can be seen as a limited, manual version of our technique. We then present our strategy-derivation rules in Section 5.2. Section 5.3 demonstrates how we introduce the derived strategies to the input program.

Burn introduced the idea of *evaluation transformers* as a way to specify how much of a value can be reduced safely before it is needed [Burn, 1987]. By using the results of a four-point strictness analysis Burn was able to annotate expressions based on how much of the structure could be safely evaluated. The main insight was that each annotation represented a demand on the values and that demand could be propagated to the sub-expressions in a well defined way.

Burn defined four 'evaluators', each for lists, with the following meanings

- ξ_0 performs no evaluation
- ξ_1 evaluates its argument to WHNF
- ξ_2 evaluates the spine of its argument
- ξ_3 evaluates the spine of its argument and each element to WHNF

These correspond directly to each point in Wadler's four-point domain. The compiler can apply the relevant evaluation transformer to each expression that it is safe to do so. Burn realised that the strictness of certain functions can be dependent on which evaluation transformer forced the evaluation of the function itself. With hindsight we can see that this is very similar to the motivation behind projection based analysis (and indeed Burn noted this relationship in later work. Additionally, the runtime system can keep track of which evaluation transformer is used on each expression. This allows the runtime system to propagate the evaluation transformers where possible.

5.1.1 *Gaining Context*

As noted above, Burn's main insight was that strictness analysis using Wadler's four-point domain did not provide any information about the *demand context* of a function application. This prevented the results of strictness analysis from identifying a substantial amount of implicit parallelism, as noted in results for analysing *append* using the four-point domain (Section 4.4, Table 3).

Burn was able to regain some of this context using a *reverse* four-point domain analysis after first analysing the program with the original four-point domain analysis [Burn, 1987, Section 4]. This results in an *evaluation transformer* for each function in the program (if the types do not have evaluators defined for them then only ξ_0 and ξ_1 are used, which correspond to no evaluation and evaluation to WHNF, respectively). The evaluation transformer for *append* is shown in Table 4, the columns for $append_1$ and $append_2$ represent the appropriate evaluator for the first and second argument to *append*, respectively.

E	$append_1(E)$	$append_2(E)$
ξ_0	ξ_0	ξ_0
ξ_1	ξ_1	ξ_0
ξ_2	ξ_2	ξ_2
ξ_3	ξ_3	ξ_3

Table 4: The Evaluation Transformer for append

Knowing the evaluation transformers themselves is not enough, there must also be a way to utilise the information at runtime. Burn's resolution to this problem was to annotate the application nodes of a program *on the heap*, i.e. dynamically. Each application node is tagged with which evaluator from above is safe to use. The evaluation of a program would proceed as normal with one addition: Whenever the evaluator enters the code for a function it checks the evaluator tag of the application node and then evaluates the argument nodes appropriately.

Some functions, such as *length* or primitive functions, always propagate the same demand to their arguments; in the case of *length* it is always ξ_2. These functions form the starting points for the propagation of the evaluation transformers. Given the expression *length* (*append xs ys*) the propagation of the evaluator on *length*'s argument to *appends* arguments is shown in Figure 28. For a more detailed exposition of how evaluation transformers are used at runtime see Hogen et al. [1992, Section 3.1].

There are two downsides with this approach: the level of evaluation is limited to pre-determined forms (in this case lists and basic values) and it is the *runtime system* that determines how the evaluation transformers are propagated. Ideally the propagation of demand would be static so that the runtime system would not have the additional bookeeping and management involved in the method introduced by Burn.

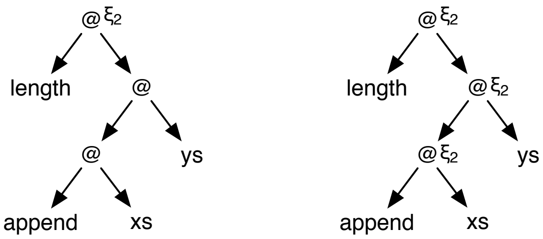


Figure 28: The propagation of an evaluator using evaluation transformers: when entering the code for `length` the evaluation transformer for `append` is indexed by ξ_2, resulting in `append`'s arguments being tagged by their appropriate evaluator

5.2 DERIVING STRATEGIES FROM PROJECTIONS

One of the reasons that projections were chosen for our strictness analysis is their correspondence to parallel strategies. Strategies are functions whose sole purpose is to force the evaluation of specific parts of their arguments [Marlow et al., 2010; Trinder et al., 1998]. All strategies return the unit value (). Strategies are not used for their computed result but for the evaluation they force along the way.

Some Examples

The type for strategies is defined as **type** $Strategy\ \alpha\ = \alpha\ \rightarrow\ ()$.

The simplest strategy, named *r0* in [Marlow et al., 2010], which performs no reductions is defined as $r0\ x\ =\ ()$. The strategy for weak head normal form is only slightly more involved: $rwhnf\ x\ =\ x\ `seq`\ ()$

The real power comes when strategies are used on data-structures. Take lists for example. Evaluating a list sequentially or in parallel provides us with the following two strategies

$$\begin{aligned}
&seqList\ s\ []\ &&=\ ()\\
&seqList\ s\ (x:xs)\ &&=\ s\ x\ `seq`\ (seqList\ s\ xs)
\end{aligned}$$

$$\begin{aligned}
&parList\ s\ []\ &&=\ ()\\
&parList\ s\ (x:xs)\ &&=\ s\ x\ `par`\ (parList\ s\ xs)
\end{aligned}$$

Each strategy takes another strategy as an argument. The provided strategy is what determines how much of each element to evaluate. If the provided strategy is *r0* the end result would be that only the spine of the list is evaluated. On the other end of the spectrum, providing a

$$\begin{aligned}
&\mathcal{C} :: \llbracket \mathit{Demand} \rrbracket \to \mathit{Names} \to \mathit{Exp} \\
&\mathcal{C}\, \llbracket c? \rrbracket\, \phi && = \lambda x \to () \\
&\mathcal{C}\, \llbracket c! \rrbracket\, \phi && = \llbracket c \rrbracket\, \phi \\
&\mathcal{C}\, \llbracket \mu\beta.c \rrbracket\, \phi && = \mathit{fix}\ (\lambda n \to \llbracket c \rrbracket\, (n : \phi)) \\
&\mathcal{C}\, \llbracket \beta \rrbracket\, (n : \phi) && = n \\
&\mathcal{C}\, \llbracket [cs] \rrbracket\, \phi && = \lambda x \to \mathit{Case}\ x\ \mathit{of}\ \mathcal{A}\, \llbracket cs \rrbracket\, \phi \\
&\mathcal{C}\, \llbracket c \rrbracket\, \phi && = \lambda x \to x\ \text{‘}\mathit{seq}\text{‘}\ ()
\end{aligned}$$

$$\begin{aligned}
&\mathcal{A} :: \llbracket (\mathit{Constructor}, \mathit{Demand}) \rrbracket \to \mathit{Names} \to (\mathit{Pat}, \mathit{Exp}) \\
&\mathcal{A}\, \llbracket (C, \mathit{ID}) \rrbracket\, \phi && = (C, ()) \\
&\mathcal{A}\, \llbracket (C, \mathit{BOT}) \rrbracket\, \phi && = (C, ()) \\
&\mathcal{A}\, \llbracket (C, \langle cs \rangle) \rrbracket\, \phi && = (C\ vs, \mathcal{F}\, \llbracket ss \rrbracket\, \phi) \\
&\quad \mathit{where}\ \ ss = \mathit{filter}\ (\mathit{isStrict} \circ \mathit{fst})\ \$\ \mathit{zip}\ cs\ vs \\
&\qquad\qquad\ \ vs = \mathit{take}\ (\mathit{length}\ cs)\ \mathit{freshVars}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F} :: \llbracket [(\mathit{Demand}, \mathit{Exp})] \rrbracket \to \mathit{Names} \to \mathit{Exp} \\
&\mathcal{F}\, \llbracket [] \rrbracket\, \phi && = () \\
&\mathcal{F}\, \llbracket ((c, v) : []) \rrbracket\, \phi && = \mathit{App}\ (\mathit{Fun}\ \text{“seq”})\ [\mathit{App}\ (\mathcal{C}\, \llbracket c \rrbracket\, \phi)\ [v], ()] \\
&\mathcal{F}\, \llbracket ((c, v) : cs) \rrbracket\, \phi && = \mathit{App}\ (\mathit{Fun}\ \text{“par”})\ [\mathit{App}\ (\mathcal{C}\, \llbracket c \rrbracket\, \phi)\ [v], ls] \\
&\quad \mathit{where}\ \ ls = \mathcal{F}\, \llbracket cs \rrbracket\, \phi
\end{aligned}$$

Figure 29: Rules to generate strategies from demand contexts

strategy that evaluates a value of the list-item type *a* to WHNF would result in list's spine *and* elements being evaluated fully.

□

Already we can see a correspondence between these strategies and the contexts shown in Figure 23. The T context (tail strict) corresponds to the strategy that only evaluates the spine of the list, while the HT context corresponds to the strategy that evaluates the spine and all the elements of a list.

In our derived strategies it is not necessary to pass additional strategies as arguments because the demand on a structure's elements makes up part of the context that describes the demand on that structure.

Because projections *already* represent functions on our value domain, translating a projection into a usable strategy only requires that we express the projection's denotation in a syntactic form. The rules we use are shown in Figure 29. Rule $\mathcal{C}$ constructs a strategy for all of the context constructors except for products. This is because product types are only found within constructors in the source language and are therefore wrapped in sums as constructor-tag context pairs. These pairs are handled by the $\mathcal{A}$ rule.

One aspect of strategies that does *not* directly correspond to a context is the choice between *seq* and *par*. Every context can be fully described by both sequential and parallel strategies. When a constructor has two or more fields, it can be beneficial to evaluate *some* of the fields in parallel. It is not clear, generally, which fields should be evaluated in parallel and which should be evaluated in sequence. As shown in rule $\mathcal{F}$ we evaluate all fields in parallel *except* for the last field in a structure. This means that if a structure has only one field then its field will be evaluated using *seq*.

5.2.1 *Specialising on Demand*

The key reason for performing a strictness analysis in our work is to know when it is *safe* to perform work before it is needed. This work can then be sparked off and performed in parallel to the main thread of execution. Using the projection-based analysis allows us to know not only *which* arguments are needed, but *how much* (structurally) of each argument is needed. We convert the projections into strategies and then spark off those strategies in parallel.

Assume that our analysis determines that function *f* is strict in both of its arguments. This allows us to convert

```
f e1 e2
```

into

```
let
  a = e1
  b = e2
in
(s1 a) `par` (s2 b) `seq` (f a b)
```

where $s1$ and $s2$ are the strategies generated from the projections on those expressions.

Different Demands on the Calling Function

If a function has different demands on its result at different calling sites, that is dealt with 'for free' using the transformation above. However, there may be multiple *possible* demands *at the same call site*.

This can happen when there are different demands on the calling function, for example:

$$func\ x \ = f\ e1\ e2$$

Different demands on the result of *func* may mean different demands on the result of f. This in turn means that different transformations would be appropriate. Assume this results in having two different demands on f. One demand results in the first transformation (*funcD*1) and the other results in the second (*funcD*2). How do we reconcile this possible difference?

Specialisation by Demand

To accommodate this possibility we can clone the function *func*. One clone for each demand allows us to have the 'more parallel' version when it is safe, and keep the 'less parallel' version in the appropriate cases. Note, we do not have to clone all functions with versions for every possible demand. Instead we can do the following for each function:

1. Determine which demands are actually present in the program
2. In the body of the function, do the different demands result in differing demands for a specific function call?
3. If no, no cloning
4. If yes, clone the function for each demand and re-write the call-sites to call the appropriate clone

Applying the above procedure to our hypothetical expression would result in the following

```
funcD1 x = let
               a = e1
               b = e2
           in
           s1 a `par` s2 b `seq` f a b
funcD2 x = let
               a  = e1
           in
           s1 a `par` f a e2
```

Upgrading Hinze's Analysis

Hinze's projection-based analysis on its own is a *context-insensitive* analysis.[1] Each function is analysed in isolation, without concern for how it is actually used. Therefore, while the projection-based analysis gives us strictness properties for every possible *demand context*, we are only able to use that information at differing call sites and do not have contextual information about the demand at identical call sites (as discussed above in the Section "Different Demands on the Calling Function").

By performing the specialisation outlined above we are able to use the calling context of a function, making the combination of the projection-based strictness analysis and the demand specialisation *context senstive* [Nielson et al., 1999]. Of course, this extra flexibility came with some costs. The specialisation on demand requires another pass of the program's AST and results in some additional code. This trade-off is well known in the static analysis community, see Nielson et al. [1999, page 95]. Luckily, we have not found the additional cost to be prohibitive in this instance.

5.3 USING DERIVED STRATEGIES

5.3.1 *The Granularity Problem*

We have now explained how we find the parallelism that is implicit in our program, but none of the analysis we provide determines whether the safe parallelism is *worthwhile*. Often static analysis will determine that a certain structure is *safe* to compute in parallel, but it is very difficult to know when it is actually of any benefit. Parallelism

[1] This context refers to the call-string of a function and not to a *demand context*.

has overheads that require the parallel tasks to be substantial enough to make up for the cost. A *fine-grained* task is unlikely to have more computation than the cost of sparking and managing the thread, let alone its potential to interrupt productive threads [Hammond and Michelson, 2000; Hogen et al., 1992].

One of the central arguments in our work is that static analysis *alone* is insufficient at finding both the implicit parallelism and determining whether the introduced parallelism is substantial enough to warrant the overheads.

Our proposal is that the compiler should *run* the program and use the information gained from running it (even if it only looks at overall execution time) to *remove* the parallelism that is too fine-grained. By doing this we shift the burden of the granularity problem away from our static analysis and onto our search techniques. This way our static analysis is only used to determine the safe parallel expressions, and not the granularity of the expressions.

Here we will describe the method by which we identify the *safe* parallelism in F-Lite programs and arrange for the evaluation of these expressions in parallel. The *strictness* properties of a function determine which arguments are definitely needed for the function to terminate, whereas the *demand* on an argument tells us *how much* of the argument's structure is needed. *Strategies* are functions that evaluate their argument's structure to a specific depth. By analysing the program for strictness and demand information, we can then generate strategies for the strict arguments to a function and evaluate the strategies in parallel to the body of the function. The strategies we generate will only evaluate the arguments to the depth determined by the demand analysis.

5.3.2 *Introducing* `par`*s*

While Section 4.5 shows how to derive parallel strategies from projections representing demand, we still require a method to *introduce* the use of the strategies within the program. Happily, we can reuse the results of the projection analysis for this task as well. The general approach taken is to apply the generated strategies to *the strict* arguments of a function. As discussed earlier, the strict arguments are *safe* for evaluation in parallel. However, there are arguments to functions that are *clearly* not worthwhile: constants, etc.. Because of this we introduce an *oracle* that will determine which arguments should be parallelised.

Example

Take the famous `fib`:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

The results of the strictness analysis show us that both arguments to + have the same demand: ID!. We therefore evaluate the recursive calls to `fib` in parallel:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = let a = fib (n - 2)
            b = fib (n - 1)
        in (s1 a) `par` (s2 b) `seq` a + b
```

There are two points to consider in the transformed program. One is that by lifting subexpressions into `let` bindings we preclude the possibility of certain compiler optimisations. The sharing of values is essential for parallel strategies to be beneficial. In particular, thunk elimination becomes more difficult. The other point is that we utilise the common technique of combining `par`s and `seq`s in order to prevent collisions between threads.

□

In order to address the granularity problem [Hammond and Michelson, 2000] we use a simple oracle to determine whether a subexpression should be evaluated in parallel. Recall that our oracle should be generous in 'allowing' subexpressions to be evaluated in parallel. Our iterative improvement *reduces* the amount of parallelism introduced by static analysis. As the oracle's only job is to determine whether a subexpression is 'worth' the overhead of parallel evaluation it has the type `type Oracle = Exp -> Bool`. The two trivial oracles are

```
allYes :: Oracle
allYes = const True
```

```
allNo :: Oracle
allNo = const False
```

allNo clearly defeats the purpose of an auto-parallelising compiler, but allYes can serve as a stress-test for the iterative process. The medium oracle used in our results returns True if the expression contains a non-primitive function call, False otherwise.

```
mediumOracle e = or $ map f (universe e)
  where
    f (App (Fun n) as)
        | n `elem` prims = False
        | otherwise      = True
    f _ = False
```

Here, universe takes an expression e and provides a list of all the valid subexpressions of e, reaching the leaf nodes of the AST.

The transformation we apply is simple. For each function application $f\ e_1 \ldots e_n$:

1. Gather all the strict argument expressions to a function

2. Pass each expression to the oracle

3. Give a name (via let-binding) to each of the oracle-approved expressions

4. Before calling f, spark the application of the derived strategy to the appropriate binding

5. If there are multiple arguments that are oracle approved, ensure that the last argument has its strategy applied with seq

We now have the necessary mechanisms in place for the *introduction* of parallelism into a program.

Part III

Experimental Platform, Benchmark Programs, and Results

6 EXPERIMENTAL PLATFORM

Compilers for functional languages are not new. More importantly, implementations of functional compilers have been described in detail in the literature and provide a strong foundation for us to build upon. Most of what we have implemented uses standard techniques for the compilers of lazy languages and, apart from the automatic derivation of strategies detailed in the last chapter, our contribution in the implementation is the incorporation of runtime feedback.

For our compiler and runtime system we have opted to use a bytecode compiler that produces code for a variant of the G-Machine [Augustsson and Johnsson, 1989a]. We then pass our G-code to a virtual machine that executes the program. While more efficient methods of compilation are known (see the STG-Machine and its improvements [Peyton Jones, 1992; Marlow and Peyton Jones, 2006]) an interpreter works for our purposes because we aim to *simulate* parallel execution. The use of simulation for profiling, debugging, and analysing parallel programs is not new [Loidl, 1998] , and it provides a few important benefits:

1. Avoids non-determinism introduced by the operating system scheduler

2. We are able to log information without affecting the runtime of the program

3. Ability to use simple memory management schemes without worry of thread-safety

Of course while simulation makes certain implementation tasks simpler, it is only worthwhile if there is some correspondence between the simulated runtime and the actual runtime when run on a truly parallel system (such as GHC). Luckily, the work on GranSim and on the Quasi-Parallel evaluator for GRIP [Peyton Jones et al., 1987] show that this is indeed the case.

Being freed from the non-determinism of the OS scheduler is a significant benefit for an iterative compiler. This allows the compiler to assume that a single execution is representative of the current program. With non-determinism in the running of the program, the compiler would have to perform repeat executions *for each compiler iteration*. This increases the cost of iterative compilation.

Plan of the Chapter

As mentioned in Section 4.5, a higher-order analysis that is suitable for the derivation of parallel strategies does not yet exist. As a consequence of this, our compiler must ensure that all programs are first-order before strictness analysis is performed. Section 6.1 describes the method we use to convert our higher-order input programs into a first-order equivalent. Section 6.2 presents the runtime statistics that we are able to measure in our runtime system. Section 6.3 discusses the method we use to incorporate the runtime profile data along with an alternative method that may be useful in future work. Lastly, Section 6.4 provides an overview of the set of benchmark programs we use in the chapters to come.

6.1 DEFUNCTIONALISATION (HIGHER-ORDER SPECIALISATION)

After parsing, the next stage of the compiler applies a defunctionalising[1] transformation to the input programs. Our defunctionalisation method is limited in scope, but sufficient for our purposes. It specialises higher-order functions defining separate instances for different functional arguments. We are careful to preserve sharing during this transformation. Here we give our motivation for introducing this transformation.

A significant motivator is that our chosen strictness analysis cannot cope with higher-order programs. However, it would certainly be possible to extend such an analysis to higher-order functions if required, but our use-case provides other incentives to remove higher-order functions. When taken together we do not see defunctionalisation as a compromise but as an enabling mechanism for implicit parallelism. Defunctionalisation increases the number of call sites and therefore increases the number of verb-par- sites available to the iterative portion of the compiler.

[1] Some have taken issue with our use of the term 'defunctionalisation'. Many see defunctionalisation as the *specific* transformation introduced by Reynolds [Reynolds, 1972] to remove higher-order functions from programs. However, we feel that defunctionalisation is the *concept* of transforming a higher-order program to a first-order program. Reynold's transformation is but one instantiation of this concept.

Central to our thesis is the concept of `par` placement within a program. Each `par` application can be identified by its *position* in the AST. In a higher-order program basing our parallelism on the location of a `par` would very likely lead to undesirable consequences. This is because parallelising the application of a function to its arguments becomes more difficult when the function in question is unknown. Take *foo* below, which takes a functional argument g.

$$\begin{array}{lcl} & & \text{-- 'g' is a functional argument} \\ \mathit{foo}\, g & = & \ldots\; g\, e_1\, e_2\; \ldots \end{array}$$

Our goal is to parallelise the evaluation of e_1 and e_2 when it is safe to do so, but because we lack concrete information about the strictness of g it is not possible to know when it is.

By defunctionalising we gain a new instance of *foo* for every unique functional argument. If *foo* was passed the functions *bar* and *qux* in our program that would leave us with the following function defintions.

$$\begin{array}{lcl} \mathit{foo}_{\mathit{bar}} & = & \ldots\; \mathit{bar}\, e_1\, e_2\; \ldots \\ \mathit{foo}_{\mathit{qux}} & = & \ldots\; \mathit{qux}\, e_1\, e_2\; \ldots \end{array}$$

Now we are able to analyse the instantiations of *foo* independently. If *bar* is strict in both its arguments but *qux* is not, we face no dilemma.

$$\begin{array}{lcl} \mathit{foo}_{\mathit{bar}} & = & \ldots\; \mathbf{let} \qquad\qquad\qquad\quad \ldots \\ & & \qquad e_1^{'} = e_1 \\ & & \qquad e_2^{'} = e_2 \\ & & \quad\;\mathbf{in} \\ & & \quad\; e_1^{'}\ \text{`}\mathit{par}\text{`}\ e_2^{'}\ \text{`}\mathit{par}\text{`}\ \mathit{bar}\, e_1\, e_2 \\ \mathit{foo}_{\mathit{qux}} & = & \ldots\; \mathit{qux}\, e_1\, e_2\; \ldots \end{array}$$

We can have our cake and eat it too! We retain our safety but are able to maximise the parallelism that our compiler introduces.

In addition to allowing the compiler to introduce more parallelism, defunctionalisation aids in the iterative portion of our work. For example, a common pattern in parallel programs is to introduce a parallel version of the `map` function

```
parMap :: (a -> b) -> [a] -> [b]
parMap f []     = []
parMap f (x:xs) = let y = f x
                  in y ‘par‘ y : parMap f xs
```

There is inevitably some overhead associated with evaluation of a par application, and of sparking off a fresh parallel thread. So if the computation f x is inexpensive, the parallelism may not provide any benefit and could even be detrimental. As parMap may be used throughout a program it is possible that there are both useful and detrimental parallel applications for various functional arguments: parMap f may provide useful parallelism while parMap g may cost more in overhead than we gain from any parallelism. Unfortunately when this occurs we are unable to switch off the par for parMap g without losing the useful parallelism of parMap f. This is because the par annotation is within the body of parMap. By specialising parMap we create two separate functions: parMap_f and parMap_g, with distinct par annotations in *each* of the instances of parMap.

```
parMap_f []     = []
parMap_f (x:xs) = let y = f x
                  in y ‘par‘ y : parMap_f xs

parMap_g []     = []
parMap_g (x:xs) = let y = g x
                  in y ‘par‘ y : parMap_g xs
```

After defunctionalisation we can determine the usefulness of parallelism in each case independently. The plan is to deactivate the par for the inexpensive computation, g x, without affecting the parallel application of the worthwhile computation, f x.

The code duplication from this defunctionalisation is allowing the compiler to simulate the results of a *context-sensitive* higher-order analysis with the results of Hinze's *context-insensitive* first-order analysis. This echos the specialisation by deman from Section 5.2.1, which also used code duplication in order to simulate some context-sensitivity.

6.1.1 *How We Defunctionalise*

Our defunctionaliser makes the following set of assumptions:

- Algebraic data structures are first-order (no functional components)
- The patterns on the left-hand side of a declaration have been compiled into case expressions

- Functions may have functional arguments but their definitions must be arity-saturated and return data-value (i.e. not function) results
- No explicit lambdas in the program, but partial applications are permitted

With these assumptions in mind, the rules for defunctionalisation are presented in Figure 30. These rules are applied to the AST in a *bottom-up* fashion. This allows the transformation to assume that the arguments to partially applied functions (like e_1' in (1)) have already been defunctionalised.

EXAMPLE Take `reverse` defined as an instance of `foldl`:

```
reverse xs  =  foldl (flip Cons) Nil xs
```

this becomes

```
reverse xs  =  foldl_flip_Cons Nil xs

foldl_flip_Cons z xs
    = case xs of
        Nil       -> z
        Cons y ys ->
            foldl_flip_Cons (flip_Cons z y) ys

flip_Cons xs x = Cons x xs
```

□

The defunctionalisation rules are admittedly simple and we do not claim a contribution with this method. A production system would likely need to develop a more robust method of defunctionalisation or opt for a higher-order strictness analysis.

Now that our programs are defunctionalised we are able to take advantage of our chosen strictness analysis.

6.2 KEEPING TRACK OF ALL THE THREADS

Before we discuss the techniques used for disabling the parallelism that is too costly, we must first discuss *how* we determine which `par` sites are not worthwhile. This involves recording a significant amount

$$\begin{aligned} &f\ e_1 \ldots e_{i-1}\ (g\ e'_1 \ldots e'_m)\ e_{i+1} \ldots e_{\#f} \qquad 0 \leqslant m < \#g \\ &\quad \Longrightarrow f_{\langle i,g,m \rangle}\ e_1 \ldots e_{i-1}\ e'_1 \ldots e'_m\ e_{i+1} \ldots e_{\#f} \end{aligned} \tag{13}$$

$$\begin{aligned} &f\ x_1 \ldots\ x_n \ = e \\ &\quad \Longrightarrow f_{\langle i,g,m \rangle}\ x_1 \ldots x_{i-1}\ y_1 \ldots y_m\ x_{i+1} \ldots x_n \\ &\qquad\qquad = e[g\ y_1 \ldots y_m / x_i] \end{aligned} \tag{14}$$

Figure 30: Rules for Defunctionalisation. #f and #g represent the arities of the functions. (1) refers to the transformation at the *call site*, (2) describes the transformation of the definition, creating a new version of f that has been specialised at its ith argument with function g and m arguments to g.

of runtime information and a method for safely switching off the `par` annotations.

As mentioned earlier, our runtime system is designed in the tradition of projects like GranSim [Loidl, 1998]. The goal is to have as much control of the execution substrate as possible. This allows us to investigate certain trade-offs while ensuring that we minimise confounding variables.

Logging

The runtime system maintains records of the following global statistics:

- Number of reduction cycles
- Number of threads
- Number of blocked threads
- Number of active threads

These statistics are useful when measuring the overall performance of a parallel program, but tell us very little about the usefulness of the threads themselves.

In order to ensure that the iterative feedback system is able to determine the overall ‘health’ of a thread, it is important that we collect some statistics pertaining to each individual thread. We record the following metrics for each thread:

- Number of reduction cycles
- Number of threads generated
- Number of threads blocked by this one
- Which threads have blocked the current thread

This allows us to reason about the productivity of the threads themselves. An ideal thread will perform many reductions, block very few other threads, and be blocked rarely. A 'bad' thread will perform few reductions and be blocked for long periods of time.

6.2.1 *Adjusting the Cost of Parallelism*

Because our simulator abstracts away from many of the bookkeeping details of the runtime system the creation and management of a thread is very close to free. In fact, the creation of a parallel task only costs the time of the `par` function itself. This only requires a handful of instructions. This is clearly too optimistic. In order to better model the fact that creating and managing parallel tasks incurs *real* cost, we must implement a method of simulating this overhead. We do this by increasing the number of reductions required to execute the `par` function, this creates additional overhead for each new thread. In Chapter 8 we will see the effect of manipulating the amount of overhead for each new thread.

6.3 TRYING FOR `PAR`: SWITCHING OFF PARALLELISM

We are now able to describe the last piece of the puzzle. Once we have recorded our runtime feedback we must then *modify* the amount of parallelism in the program. There are at least two suitable methods for accomplishing this task:

1. Have the compiler use the feedback before bytecode generation and transform the program accordingly
2. Incorporate the feedback by modifying the bytecode itself

We have chosen the latter option as it suits our needs and provides sufficient foundation for further exploration. That being said there is an explicit trade-off that has occured.

6.3.1 *Switchable* `pars`

In order to take advantage of runtime profiles we must be able to adapt the compilation based on any new information. One choice is to recompile the program completely and create an oracle that uses the profiles. This way the oracle can better decide which subexpressions to parallelise. Our approach is to modify the runtime system so that it is able to *disable* individual `par` annotations. When a specific `par` in the source program is deactivated it no longer creates any parallel tasks while still maintaining the semantics of the program. The method has two basic steps:

- `par`'s are identified via the G-Code instruction `PushGlobal "par"` and each par is given a unique identifier.
- When a thread creates the heap object representing the call to `par` the runtime system looks up the status of the `par` using its unique identifier. If the `par` is 'on' execution follows as normal. If the `par` is 'off' the thread will ignore the G-Code instruction `Par`.

There is one exception to the above rules. If a `par` is *within* a derived strategy switching it 'off' turns it into a `seq`. This is because the demand for that part of the structure has not gone away. If the overall strategy is not worthwhile than its top-level `par` will be switched off completely.

6.4 BENCHMARK PROGRAMS

In this section we will give a brief introduction to the benchmark programs that are used in experimenting with our platform. We have also provided the source listing for each program in Appendix A.

SUMEULER SumEuler is a common parallel functional programming benchmark first introduced with the work on the $\langle \nu, G \rangle$-Machine in 1989 [Augustsson and Johnsson, 1989b]. This program is often used as a parallel compiler benchmark, making it a 'sanity-check' for our work. We expect to see consistent speed-ups in this program when parallelised (9 `par` sites).

QUEENS AND QUEENS2 We benchmark two versions of the nQueens program. Queens2 is a purely symbolic version that represents the board as a list of lists and does not perform numeric computation (10 `par` sites for Queens and 24 for Queens2). The fact that Queens2 has more than double the number of `par` sites for the same problem shows that writing in a more symbolic style provides more opportunity for *safe* parallelism.

SODACOUNT The SodaCount program solves a word-search problem for a given grid of letters and a list of keywords. Introduced by Runciman and Wakeling, this program was chosen because it exhibits a standard search problem and because Runciman and Wakeling hand-tuned and profiled a parallel version, demonstrating that impressive speed-ups are possible with this program [Runciman and Wakeling, 1996] (15 `par` sites).

TAK Small recursive numeric computation that calculates a Takeuchi number. Knuth describes the properties of Tak in [Knuth, 1991] (2 `par` sites).

TAUT Determines whether a given predicate expression is a tautology. This program was chosen because the algorithm used is *inherently sequential*. We feel that it was important to demonstrate that not all programs have implicit parallelism within them, sometimes the only way to achieve parallel speed-ups is to rework the algorithm (15 `par` sites).

MATMUL List-of-list matrix multiplication. Matrix multiplication is an inherently parallel operation; we expect this program to demonstrate speed-ups when parallelised (7 `par` sites).

7 RUN-TIME DIRECTED SEARCH

> *[...] even in a conservative regime, too much parallelism may be generated. This can raise serious resource-management problems*
>
> – Peyton Jones [1987]

Having explored the design of our experimental platform we can now begin describing some of the experiments that we have conducted. In this chapter we use standard bitstring search techniques during our iterative step, i.e. we do not utilise *any* runtime information other than the actual *running time* of the program. In other words, this chapter presents 'black-box' exploration of the search space.

The intuition is that because each `par` site is a switch, our technique lends itself to representing the 'setting' of our parallel program as a bitstring, each index in the string representing a single `par`. The fitness function, in our case, is the program's execution time. The healthier a `par` setting, the faster our program will run.

Plan of the Chapter

The rest of this chapter describes our technique in more detail. Section 7.1 presents the two heuristic algorithms that we will use and the expected trade-offs for each. We describe our empirical method and results in Section 7.3. Lastly, we offer our conclusions and discuss related work in Section 7.4.

7.1 HEURISTIC ALGORITHMS

Because we have chosen to perform the iterative step of compilation after bytecode generation, we can know *statically* how many `pars` are in a program. This allows us to represent a specific *setting* of a program's parallelism by a bitstring of static length, allowing us to perform standard search techniques. The motivation for doing so is simple: while our platform can record runtime statistics and profiling data, the overall goal is to produce a program that performs better than the input program. The collection of profiling data is simply one means of accomplishing this (which we will investigate in Chapter 8). Using the target goal, the improved performance of our program, as a fitness function allows this method to be used even when a runtime system is not able to record such profiling data.

There are a number of heuristic search techniques that perform search-space exploration using the evaluation of a fitness function [Russell and Norvig, 1995]. For this thesis we have chosen to explore two algorithms: a greedy algorithm, guaranteed to take linear time in the number of bits, and a hill-climbing algorithm that explores more of the search space but at the cost of being *potentially* exponential in the number of bits.

REPRESENTATION We represent the choice of enabled `par`s as a bitstring where a 1 indicates that the `par` is applied at a site, and 0 that it is not. The length of the bitstring is the number of `par` annotations introduced by the static analysis and transformation stage of the compiler.

Each index points to a unique address in the bytecode of the program, and the order of bits does not have any semantic meaning, i.e. the bit at index 1 does does not necessarily have any relationship to the bit at index 2.

FITNESS For both algorithms we use the same fitness function: the overall runtime of the program (measured in bytecode reductions). Therefore we aim to *minimise* the result of a fitness evaluation.

7.1.1 *Greedy Algorithm*

The greedy algorithm is designed to be simple but effective. The intuition is that each bit is either beneficial to the program's performance, or detrimental. Therefore the greedy algorithm visits each bit *exactly once*. By never revisiting a bit after a decision has been made about it we can guarantee a linear time complexity.

The greedy algorithm considers the bits in our representation in a random order. This avoids any potential bias toward the early bits in a bitstring. As each bit is considered, the bit is flipped from its current setting and the program is evaluated using the settings of the resulting bitstring; the setting of the bit—current or flipped—with the

```
Algorithm: Greedy bitstring search
Data: An initial par setting as a bitstring of size $N$
Result: The best performing bitstring
best.fitness $\leftarrow$ evaluateProg()
best.setting $\leftarrow$ setting
for $i \leftarrow 1$ to $N$ do
    $j \leftarrow$ uniqueRand($N$)
    flipSwitch(setting[j])
    fitness $\leftarrow$ evaluateProg()
    if fitness $<$ best.fitness then
        best.fitness $\leftarrow$ fitness
        best.setting $\leftarrow$ setting
    else
        flipSwitch(setting[j])
return best.setting
```

Algorithm 1: Greedy par-Setting Search

better fitness is retained. The algorithm terminates once all the bits have been evaluated.

The listing in Algorithm 1 provides the actual algorithm used in performing our greedy search. There are a few points to be aware of:

- The function `uniqueRand` provides random numbers but ensures that the same index will not be visited twice.
- The function `evaluateProg` runs the loaded bytecode file, returning the global reduction count
- `flipSwitch` modifies the bytecode pointed at by an index so that the function calls switches from `par` to `parOff` or vice versa
- best is a structure containing a fitness value and a bitstring of `par` settings

7.1.2 *Hill-Climbing Algorithm*

We use a simple hill-climbing algorithm in which the neighbours of the current bitstring are those formed by flipping a single bit. At each iteration, these neighbours of the current bitstring are considered in a random order, and the fitness evaluated for each in turn. The first neighbour that has a better fitness, i.e. fewer reductions are made by the main thread, than the current bitstring becomes the current

```
Algorithm: Hill-Climbing bitstring search
Data: An initial par setting as a bitstring of size $N$
Result: The best performing bitstring
best.fitness ← ∞; best.setting ← setting
while searching do
    searching ← False
    for $i$ ← 1 to $N$ do
        j ← uniqueRand(N)
        flipSwitch(setting[j])
        fitness ← evaluateProg()
        if fitness < best.fitness then
            best.fitness ← fitness
            best.setting ← setting
            searching ← True
            break
        else
            flipSwitch(setting[j])
    refreshUniques()
return best.setting
```

Algorithm 2: Hill-Climbing `par`-Setting Search

bitstring in the next iteration. The algorithm terminates when no neighbour of the current bitstring has a better fitness.

Our hill-climbing algorithm (shown in Algorithm 2) is only slightly more complex than our greedy algorithm, but much more powerful. Its power comes from its ability to revisit indices in the bitstring. The function `refreshUniques` resets the state within `uniqueRand` to allow for already visited indices to be generated again. However it does *not* reset the key in the psuedorandom number generator.

7.1.3 *Initial `par` Setting*

While we now have both algorithms in hand, there is still an important decision to be made. As shown in the algorithm listings, neither algorithm initialises the bitstring representing our `par` setting. There are three obvious choices available to us:

1. All bits on

2. All bits off

3. A random bitstring

Based on our problem domain, option 1 seems the most intuitive. We would like to begin with as much parallelism as possible, and prune out detrimental `pars`. For this reason, we will not consider option 2. Additionally, many `par` sites are only meaningful when other `pars` are turned on.[1] However, it is possible that by starting in a random location in the search space, we may avoid a local optimum that our search techniques may encounter. So we experiment with both an initial setting with all `pars` on, and with random initial settings.[2]

7.2 RESEARCH QUESTIONS

Because the two algorithms we are exploring have stochastic aspects it is important that we are careful in measuring the results and ensuring that we use proper statistical methods when deciding whether our technique 'works'.

Part of this process is deciding on what we are testing *before* we run the experiments and perform any statistical analysis. This ensures that we compare the appropriate statistics.

The main thesis of our work is that by adding an iterative step to the automatic parallelisation we get better performance than through static analysis alone. Therefore, the most important question to test is the following:

[1] The `pars` that we *within* strategies are only meaning when the strategy itself is sparked off in parallel.

[2] We realise that by being random we actually include option 2 as well, but it becomes a rare edge-case.

RQ1 What speed-up is achieved by using search to enable a subset of `pars` compared to enabling all the `pars` found by static analysis?

While the previous question is important, it is also important that we gain speedups *as compared to the sequential version*. With this in mind, it is important to ask the following question:

RQ2 What speed-up is achieved by parallelisation using search compared to the sequential version of the software-under-test (SUT)?

As discussed in the last section, we consider two algorithms: a simple hill-climbing algorithm and a greedy algorithm:

RQ3 Which search algorithm achieves the larger speed-ups, and how quickly do these algorithms achieve these speed-ups?

Because we have decided to use two methods for the initial `par` settings we have one final research question:

RQ4 Which form of initialisation enables the algorithm to find the best speed-ups: all `pars` enabled (we refer to this as *'all-on'* initialisation), or a random subset enabled (*'random'* initialisation)?

7.3 EXPERIMENTAL SETUP AND RESULTS

In this section we evaluate the use of search in finding an effective enabling of `pars` that achieves a worthwhile speed-up when the parellelised program is run in a simulated multi-core architecture. As a reminder, the starting point for our proposed technique is a program that was originally written to be run sequentially on a single core; static analysis identifies potential sites at which `par` functions *could* be applied; and then search is used to determine the subset of sites at which the `par` is actually used.

7.3.1 *Method*

The following four algorithm configurations were evaluated:

- hill-climbing with all-on initialisation
- greedy with all-on initialisation

- hill-climbing with random initialisation
- greedy with random initialisation

Each algorithm configuration was evaluated for four settings of the number of cores: 4, 8, 16 and 24 cores. Each algorithm / core count combination was evaluated against each of the seven benchmark programs described above.

Since both search algorithms are stochastic, multiple runs were made for each algorithm / core count / benchmark combination, each using 30 different seeds.[3]for the pseudo-random number generator. For all runs, after each fitness evaluation, the best bit string found and its fitness (the number of reductions made by the main thread), was recorded.

In addition, the fitness (number of reductions) was evaluated for a bit string where all bits are set to 1. This evaluation was made for each combination of core count and benchmark. Finally, the fitness was evaluated for the sequential version of each benchmark.

OVERHEADS Our runtime system allows us to set the cost of creating a parallel task, this models the overhead present in real systems. Using the Criterion [O'Sullivan, 2009] benchmarking library we found an approximate cost for the creation of a `par` in GHC's runtime.[4] For the experiments we have chosen an overhead of 300 reductions for each call to `par`.

7.3.2 *Results*

The results are summarised in Table 5. This table compares the speed-up, calculated as the ratio of the medians of the reduction counts, of hill-climbing with all-on initialisation compared to (a) the parallelisation that would result from the static analysis without optimisation; (b) the sequential version of the program; (c) the greedy algorithm with all-on initialisation; and (d) the hill-climbing algorithm with random initialisation. The speed-up is calculated as the factor by which the number of reductions is reduced, and so values greater than 1 indicate that the program parallelised using hill-climbing with all-on initialisation would be faster in the multi-core environment. Values in bold in the table indicate that differences between the algorithms

[3] All seeds were obtained from `www.random.org`.

[4] The code for benchmarking the cost of a `par` is available at `https://github.com/jmct/par-experiments`

used to calculate the speed-up are statistically significant at the 5% level using a one- or two-sample Wilcoxon test as appropriate.[5] The values bolded in blue are both statistically significant at the 5% level *and* exhibit a speedup for more than 5%. This separates the statistically significant speedups that are unlikely to manifest on a real machine. The results for `Taut`, for example, are statistically faster, but the difference is so minute (less than one tenth of a percent in all cases!) that the non-determinism of a real architecture is likely to render the speedup irrelevant.

7.3.3 *Discussion of Research Questions*

We can now look again at our four research questions from Section 7.2 and determine whether our experiments have given us meaningful results to any of them.

RQ1 For most of our benchmarks there is a relatively large speed-up of the hill-climbing algorithm compared to the default parallelisation where all `pars` are enabled. The largest speed-ups are for `Queens2` where we might expect a wall-clock run time that is more than 6 times better than the default parallelisation. For `Queens` and `Taut` the speed-ups are closer to 1, but are in all cases statistically significant.

An interesting property regarding the speedups as compared to the static placement is that they decrease as the number of cores increases. This aligns with our intuition for parallel programs. As the number of cores goes up, the static placement *gets away* with non-optimal `par` placement. With more cores, there is less contention for the resources of the machine. A `par` that does more work than its overheads is less likely to interrupt another, more productive thread. When the number of cores is lower, the contention makes even productive `pars` detrimental because they interrupt threads that are even more productive.

We conclude that both the greedy and the hill-climbing algorithm can improve parallel performance across a range of benchmarks and

[5] Since in the following we discuss the results for each benchmark program, or combination of benchmark program and number of cores, individually as well as for the entire set of results as a family, we do not apply a Bonferroni or similar correction to the significance level. Nevertheless we note here that most of the currently significant differences would remain significant if such a correction were applied.

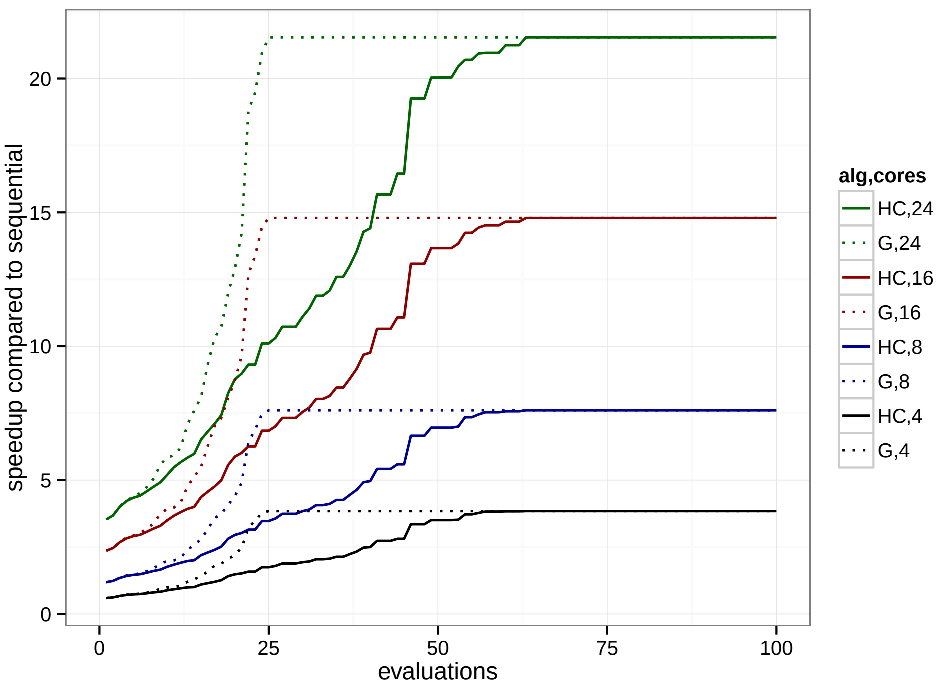
		hill-climbing search speed-up compared to:			
SUT	Cores	Static Parallel	Sequential	Greedy	Random Init
MatMul	4	4.903	1.021	1	1
	8	4.625	1.021	1	1
	16	4.485	1.021	1	1
	24	4.439	1.021	1	1
Queens	4	1.080	1.294	1	1
	8	1.043	1.369	1	1
	16	1.017	1.401	1	1
	24	1.003	1.401	1.000	1
Queens2	4	6.479	3.843	1	1
	8	6.421	7.607	1	1
	16	6.263	14.79	1	1
	24	6.101	21.54	1	1
SodaCount	4	4.237	3.773	1.001	1.055
	8	3.544	6.207	1.007	1.071
	16	3.110	10.40	1.081	1.072
	24	2.810	13.26	1.004	1
SumEuler	4	1.494	3.948	1	1
	8	1.486	7.773	1	1
	16	1.460	14.77	1	1
	24	1.432	20.69	1	1
Tak	4	1.609	1.560	1	1
	8	1.609	3.118	1	1
	16	1.608	6.230	1	1
	24	1.608	9.330	1	1
Taut	4	1.000	1.000	1.000	1
	8	1.000	1.000	1.000	1.000
	16	1.000	1.000	1.000	1
	24	1.000	1.000	1.000	1

Table 5: The speed-up, calculated as the ratio of the medians of the reduction counts, achieved by the hill-climbing algorithm using all-on initialisation.

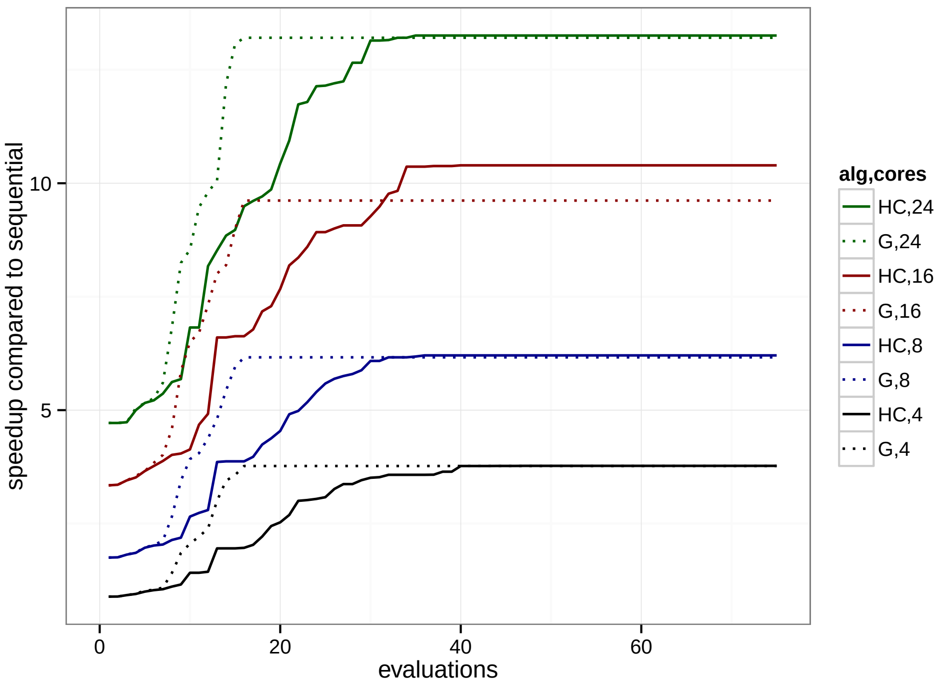
across a range of core counts when compared to using the static placement of parallelism.

RQ2 For Queens2 and SumEuler, the speed-up compared to the sequential version of these benchmarks is almost linear: it approaches the number of cores available. For example, for SumEuler on 4 cores, the speed-up compared to the sequential version is 3.95. A linear speed-up is the best that can be achieved, and so these results are indicative that our proposed technique could be very effective in practice. Meanwhile, for other benchmarks such as MathMaul and Taut, there is little speed-up over the sequential version of the benchmark.

RQ3 The results show that for most benchmarks, there is little difference in the speed-up achieved by the hill-climbing and greedy algorithm. (For clarity, the table shows the comparison only between the two algorithms using all-on initialisation, but similar results are obtained when initialisation is random.) Only for SodaCount is there a non-trivial and statistically significant difference between the hill-climbing and greedy algorithm for all core sizes. Figure 31 performs a further analysis for this research question: for two of the benchmarks, it plots the best speed-up (compared to sequential) obtained so far by the algorithm against the number of fitness evaluations. For Queens2 at all core counts, the greedy algorithm finds the same best speed-up as the hill-climbing, but finds it in fewer fitness evaluations, i.e. the search is faster. For SodaCount, the greedy algorithm finds its best speed-up in relatively few evaluations. The hill-climber takes longer but finds a better speed-up at all cores counts; the difference is most noticeable in the results for 16 cores. For the goal of having a compiler that provides you with a faster program while you take a tea break, the greedy algorithm seems satisfactory. For frequently-used benchmarks that account for a significant part of a system's performance, the additional effort required to find the best parallelisation using hill-climbing may be justified, but will depend on context. In the end this is a subjective trade off; we feel that the results support the use of the greedy algorithm unless finding the optimal solution is absolutely necessary.



(a) Queens2



(b) SodaCount

Figure 31: The speed-up, calculated as the ratio of the medians of the reduction counts, obtained so far by the algorithm plotted against the number of fitness evaluations. HC and G indicate the hill-climbing and greedy algorithm respectively, both using all-on initialisation. The numbers following the algorithm abbreviation indicate the number of cores.

RQ4 For most benchmarks there is no statistically significant difference between all-on and random initialisation. For `SodaCount`, the all-on initialisation is slightly better for core counts of 4, 8, and 16. This result provides evidence that all-on initialisation may be beneficial, but requires further investigation to confirm the generality.

The only results elided in Table 5 are the runtimes for the greedy search with a random initialisation. This is because the random initialisation produces inferior results in all cases and the same insight can be gathered from studying the hill-climbing results for random initialisation.

7.4 SUMMARY OF BITSTRING SEARCHING

We feel that this chapter has provided evidence that the combination of static analysis and search can parallelise programs more effectively than through static analysis alone. For some programs we are able to achieve close to linear speed-ups which is as performant as can expected. These results are promising for those looking to add iterative capabilities to their compiler but without adapting the runtime system. By choosing an appropriate representation of the available parallelism in a program, we are able to use standard search techniques to search the possible parallel configurations.

The success of the greedy algorithm was somewhat surprising. It further supports the idea that the vast majority of potential parallelism in any given program does not make up for the overhead costs associated with creating and managing that parallelism.

8 PROFILE DIRECTED SEARCH

> *When concurrent evaluation is used to gain efficiency one actually would like to have an analyser that* automatically *marks expressions that can safely be executed in parallel. [. . .] But one often also needs to know whether parallel evaluation is* worthwhile*.*
>
> – Plasmeijer and Eekelen [1993]

The results from the previous chapter, while promising, assume that the runtime system is a 'black box'. In some cases it may be necessary or desirable to make this assumption but we feel that with access to information from the runtime system our technique can produce meaningful results with fewer iterations.

The work in this chapter uses profile information and other statistics from each iteration of the program before determining where to proceed in the search space.

Plan of the Chapter

We begin by defining the concept of `par` site health in Section 8.1. This metric provides us with a way to make decisions about how to proceed at each iterative step. The algorithm for incorporating `par` site health is presented in Section 8.2. We then present and discuss the results of using this algorithm in Section 8.3, we also use the opportunity of having profiling information to experiment with various simulated overheads for the 'cost' of a `par`. In Section 8.4 we show how the benchmark programs perform when we naively transfer the resulting programs to be compiled by GHC. Lastly, we provide a summary of our results from these experiments and some conclusions that we can draw from them in Section 8.6.

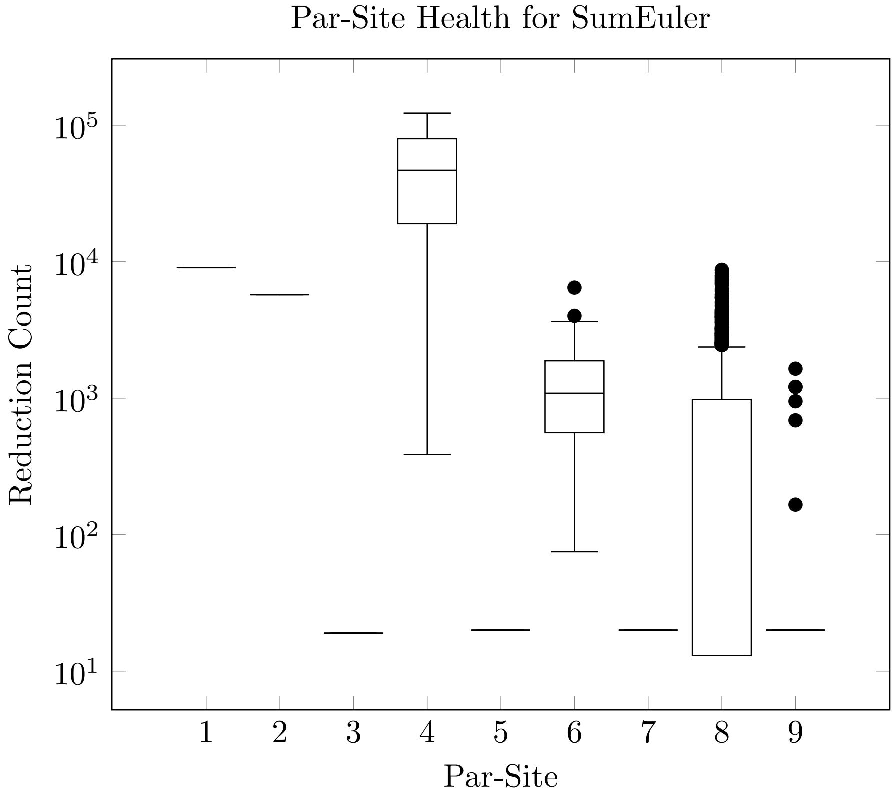


Figure 32: Statistics on the number of reductions carried out by the threads a `par` site sparks off

8.1 PAR-SITE HEALTH

In this section we will present what it means for a `par` site to be worthwhile. We do this by measuring the *health* of each `par` site.

The runtime system provides us with a variety of statistics about threads and the `pars` that sparked them. However, our current approach focuses on reduction count as a guide to determine which `par` sites are beneficial to the program. The reasoning is simple; our motivation for parallelism is to do more work at once while ensuring that each unit of work done by a thread makes up for the cost of creating and managing that thread, so measuring the amount of work undertaken by each thread is the most direct measure of this desirable property.

Because we record how productive each thread in a program is and we keep track of which `par` site created each thread, we can easily visualise how useful each `par` site is. Figure 32 gives an overview of the health of each `par` site for the `SumEuler` benchmark. The plot shows us the statistics for this data with the median (line), inter-quartile range (IQR, box), and ±1.5∗IQR (whiskers). Statistical outliers are shown as independent points. The par sites that only show a line as their plot either have only one child thread (the case for par-site 1) or have little variance in the distribution of reduction counts.

Even by just plotting this information we gain a much better understanding of SumEuler's parallel behaviour. Much like using Threadscope [Jones et al., 2009], having this information helps the programmer (and the compiler in our case) make better decisions about where to look for performance improvements. In the case of the profile shown in Figure 32 we can clearly see that some par sites do not spark hard-working threads.

Additionally, we can see that the variance between the productivity of the threads sparked by a par site can be quite high (par site #4 sparks some threads that perform hundreds of reductions and some that perform hundreds of thousands of reductions).

We define a pars **health** as the *mean* of the reduction counts for *all* the threads sparked off by the par site in question. This simple view provides a rough estimate of how worthwhile a par site is overall while requiring very little computation on its own. Another important aspect of this measure is that par site health is more of a *relative* measure than an absolute measure. Trying to define what is healthy or not healthy for all programs is difficult. Instead we aim to rank a par's health as compared to the other par's in a program.

For the SumEuler program's par sites as shown in Figure 32, this means that par site numbers 3, 5, 7, 8, and 9 are less healthy than site numbers 1, 2, 4, and 6.

8.2 SEARCH ALGORITHM

The iterative step of our compiler must make a decision about which par setting to provide the runtime with for the next execution of the program. In the previous chapter we used heuristic techniques to provide this decision making. Now we will use the notion of par site health described in the previous section.

After every execution of the program, turn off the par site with the lowest health (the lowest average reduction count). In the case of the execution statistics displayed in Figure 32 we would disable par site #8.[1] allowing us to avoid the overhead of all the unproductive

[1] Notice that par site #8 also has some very productive threads, but even so, its median is the lowest in that run of the program.

threads it sparked off. Then repeat this process until switching a par site off increases the overall runtime of the program.

```
Algorithm: Profile-Driven Search
Data: An initial par setting as a bitstring of size N (all bits on)
Result: The best performing bitstring
last.runtime ← ∞
for i ← 1 to N do
    current ← evaluateProg()
    if current.runtime > last.runtime then
        break
    else if allOff?(current.setting) then
        return current.setting
    last ← current
    weakest ← calculateHealth()
    flipSwitch(current.setting[weakest])
return last.setting
```

Algorithm 3: Greedy par-Setting Search

It is worth noting that our algorithm is really a hill-climbing algorithm with an oracle. Instead of randomly evaluating neighbours, the search moves to the neighbour where the weakest par site is switched off. The success of this search algorithm will depend on how well this corresponds to an increase in overall performance.

8.3 experimental results and discussion

In this section we present some preliminary results and point out certain patterns that appear in our data.

Overheads

Whether an expression is worthwhile to evaluate in parallel is directly tied to *cost* of creating a parallel task. Because our search algorithm is not deterministic, we do not have the statistical issues that were present in the previous chapter. Therefore we took the opportunity to experiment with various overheads as well.

Based on the benchmarking of GHC's par we have chosen a lower and upper bound. While the lower bound (10) and the upper bound (1000) are both unrealistic (i.e. benchmarking GHC points to the actual overhead being somewhere in the hundreds of reductions) this will help us see how sensitive our approach is to the overhead of a `par`.

Experimental Results

For each program we set our runtime system to simulate 4, 8, and 16 cores. First, let us examine Table 6 which displays the results of setting the cost of task creation to 10 reductions.

Already there are a few interesting results. `SumEuler` performs as expected and manages to eliminate the majority of the introduced `par` sites. Interestingly, the `par` sites that remain are, when taken together, equivalent to applying `parMap euler` over the input list.[2] When this program is parallelised explicitly, that `parMap` is usually the only addition to the program [Augustsson and Johnsson, 1989b]. It is reassuring that our technique converges on the same result.

The two implementations of `nQueens` vary drastically in their improvement, with the more symbolic solution (`Queens2`) achieving much better results. Search problems are known to be problematic for techniques involving strictness analysis and usually benefit from the introduction of *speculative* parallelism [Hammond and Michelson, 2000].

`Taut` was chosen as a benchmark program specifically because the program (as written) did not have many opportunities for parallelism. Had our technique managed to find any useful parallelism, we would have been surprised.

`MatMul` is, to us, the most surprising of the results so far. Matrix multiplication is famously parallelisable and yet our implementation barely breaks even! Notice that of the 7 `par` sites in `MatMul`, only 2 are being switched off. We will investigate `MatMul`'s performance in a bit more depth in Section 8.5.

While the results in Table 6 are revealing, it could be argued that an overhead of 10 reductions to spark off a thread is unrealistically low. Therefore we repeat the experiments with the more realistic 100 reduction overhead (Table 7) and the pessimistic case of 1000 reduction overheads (Table 8).

The results in Table 7 mostly align with what we would expect to happen if creating a parallel task incurred higher overheads: we see reduced speedup factors and adding more cores is less likely to benefit.

[2] See Appendix A for the source of the program.

Program	4-core		8-cores		16-cores	
	Runs	Final	Runs	Final	Runs	Final
SumEuler	6	3·77	6	6·84	6	10·27
Queens	5	1·30	5	1·37	5	1·41
Queens2	22	3·91	22	7·74	22	15·07
SodaCount	3	2·42	3	4·72	3	8·95
Tak	1	3·39	1	6·79	1	13·58
Taut	4	1·00	0	1·00	9	1·00
MatMul	2	1·02	2	1·07	2	1·10

Table 6: Speedups relative to sequential computation when the cost of sparking a task is set to 10 reductions. The number of runs corresponds to the number of `par` sites that have been switched off.

Program	4-core		8-cores		16-cores	
	Runs	Final	Runs	Final	Runs	Final
SumEuler	6	3·74	6	6·81	6	10·23
Queens	5	1·29	5	1·37	5	1·41
Queens2	22	3·83	22	7·57	22	14·76
SodaCount	3	2·17	3	4·23	3	8·02
Tak	1	2·36	1	4·71	1	9·42
Taut	9	1·00	0	1·00	9	1·00
MatMul	2	0·93	2	1·06	2	1·09

Table 7: Speedups relative to sequential computation when the cost of sparking a task is set to 100 reductions. The number of runs corresponds to the number of `par` sites that have been switched off.

Program	4-core		8-cores		16-cores	
	Runs	Final	Runs	Final	Runs	Final
SumEuler	6	3·51	6	6·40	6	9·73
Queens	5	1·26	5	1·35	5	1·40
Queens2	22	3·14	22	6·22	22	12·18
SodaCount	12	1·85	3	2·08	1	1·39
Tak	1	0·57	1	1·15	1	2·32
Taut	12	1·00	12	1·00	7	1·00
MatMul	5	1·00	5	1·00	5	1·01

Table 8: Speedups relative to sequential computation when the cost of sparking a task is set to 1000 reductions. The number of runs corresponds to the number of `par` sites that have been switched off.

The key point to take away from this set of results is that while lower speedups are achieved, the *same* par sites are eliminated in the same number of iterations.[3]

Now we try the same experiment again but with the less realistic 1000 reduction overhead to create a new thread.

While the speedups are now much more moderate (when there is a speedup at all) these results are interesting for a few reasons.

In particular, the number of cores now has a greater influence on how many par sites are worthwhile. SodaCount, for instance, now eliminates 12 of its 15 par annotations in the case of 4-core execution. This fits with our intuition that when there are fewer processing units the threads require coarser granularity to be worthwhile. In the cases of 8 and 16-core executions we observe that fewer par sites are disabled, reinforcing this intuition.

MatMul also sees a jump in the number of disabled par sites. Sadly, this results in even worse performance for MatMul, which should be a highly parallelisable program.

Static vs. Iterative

While the results presented in Tables 6, 7, and 8 are promising for preliminary results they are based on an admittedly simple search heuristic. Part of our argument is that static analysis *alone* is not sufficient for good gains from implicit parallelism. Figures 33, 34, 35, and 36 present a selection of results that show how the iterative improvement affects the static placement of par annotations.

[3] Except for Taut, which in the 4-core case now takes 9 runs to determine that there is no parallelism in the program.

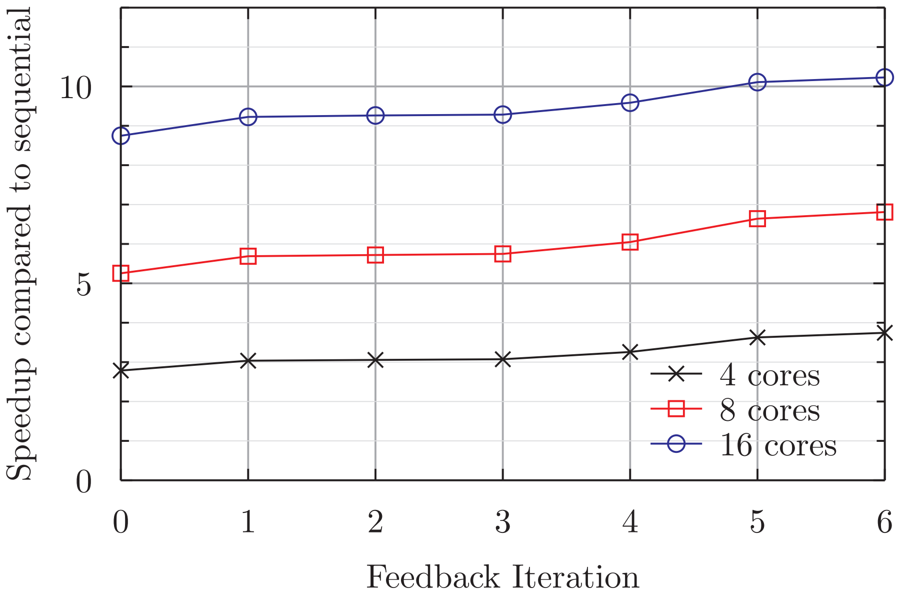


Figure 33: `SumEuler` speedup

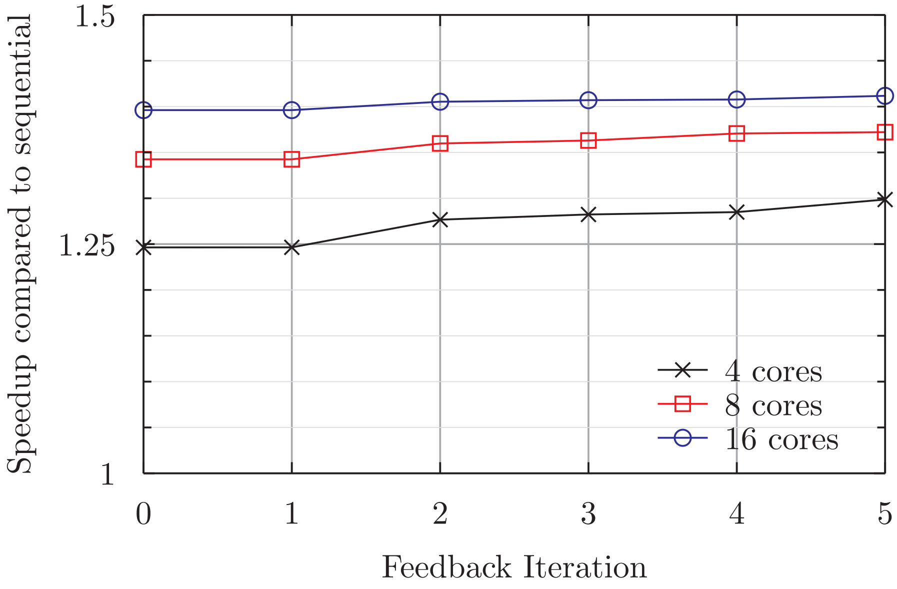


Figure 34: `Queens` speedup

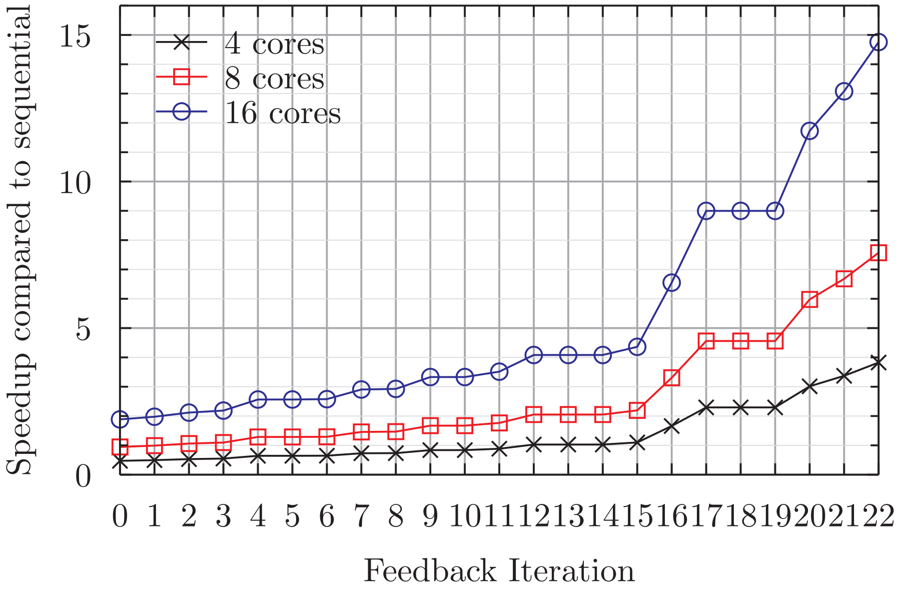


Figure 35: Queens2 speedup

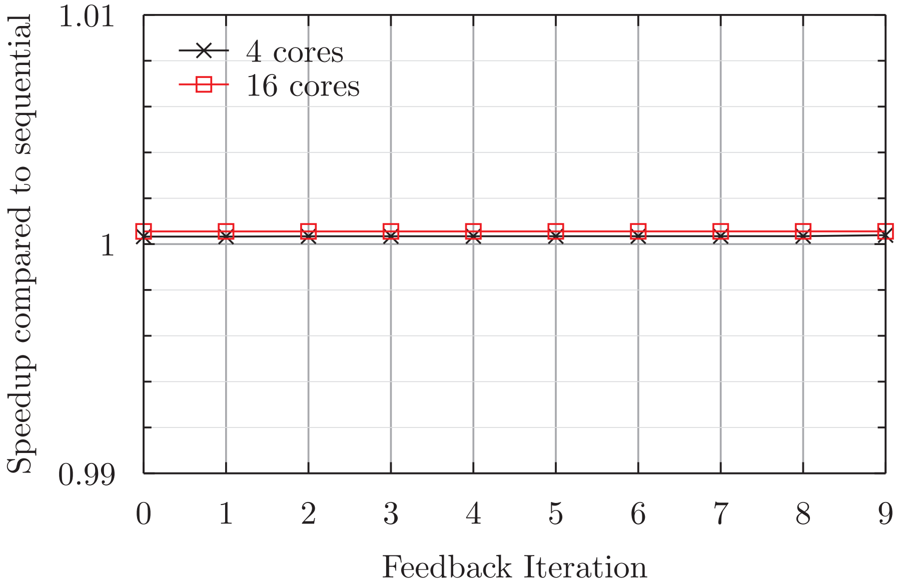


Figure 36: Taut speedup

Even in the cases where the final speedup is lower than anticipated, such as Queens in Figure 34, the program still benefits from the iterative improvement. Queens2 sees the highest payoff from iterative improvement. Many of the pars introduced by the static analysis do not contribute significantly to the computation even though it is semantically safe to introduce them. The iterative loop converges on the few par sites that make a significant difference.

It may be noticed that the iterative steps sometimes plateau for a few iterations, this is particularly evident in Figures 33 and 35. The reason for this is that a top-level strategy has been switched off, making all of the pars that make up that strategy into dead code. Luckily we can determine which pars are descended from other pars statically and skip these plateaus when they occur. This reduces the number of iterative steps even further for some programs; Queens2 then takes only 14 iterative steps instead of 22, for example.

8.4 TRANSFER TO GHC

While the results above are encouraging,, we would like to see how the resulting programs perform when compiled by a modern high-performance Haskell compiler. To do this we extract the final par settings from each program and translate that to Haskell suitable for compilation by GHC.

For the versions parallelised by hand we use the par placements found in the literature [Augustsson and Johnsson, 1989b; Runciman and Wakeling, 1994].

Program	4-core	
	Hand	Auto
SumEuler	3.32	3.31
Queens	1.76	0.97
Queens2	2.29	0.61
SodaCount	1.25	0.64
Tak	1.77	1.64
MatMul	1.75	0.80

Table 9: Speedups compared to the sequential program as compiled by GHC for manually and automatically parallelised versions

As Table 9 makes clear, the results are not impressive. In fact, except for SumEuler and Tak, all of the parallel benchmarks performed *worse* than their sequential counterparts.

However, we feel that not all hope is lost. There are a few recurring issues in the generated program. A common issue is that the generated strategies will not be what forces the evaluation of a value. Take the following example as an illustration

```
foo n = let ys = gen n n
        in par (tailStrict1 ys) (bar ys)

tailStrict1 xs = case xs of
    y:ys -> tailStrict1 ys
    []   -> ()
```

In the function `foo` we spark off a strategy that is meant to force the spine of the list `ys`. The catch is that GHC's `par` is fast enough for `bar ys` to be what forces the evaluation of `ys`. So we're paying the overhead and reaping none of the benefits. In some programs changing a `par` like the one found in `foo` to a `seq` is enough to solve the issue and make the parallel version competitive with the manually parallelised version. `Queens`, `Queens2`, and `SodaCount` all benefit from this adaptation. The issues with `MatMul` are more subtle and will be discussed in the next section.

8.5 LIMITATIONS

Our approach falls short on many programs that should be easily parallelisable. In this section we will explore the main limits of our technique by examining what causes the `MatMul` benchmark to perform so poorly. `MatMul` was expected to be a success story of this approach and understanding why the technique does not properly parallelise the program is important in improving this work in the future. The exercise of hand parallelising the programs in Section 8.4 illuminated two main issues: the transfer of parallelism from the callee to the caller and lack of speculation. These two issues are interrelated but we will present them separately for clarity.

8.5.1 *Caller and Callee Parallelism*

Strategies offer an elegant method of separating the logic of an algorithm from how it should be parallelised. The way it accomplishes this is by removing the burden of the *callee* to parallelise the structure and instead ensures that the *caller* parallelises its result. This distinction is easy to see with the classic *parMap* function (remember that in our work we must defunctionalise):

$$
\begin{array}{lll}
\mathit{parMap}_f & :: & [\alpha] \rightarrow [\beta] \\
\mathit{parMap}_f\ [\,] & = & [\,] \\
\mathit{parMap}_f\ (x : xs) & = & \mathit{fx}\ \text{`}\mathit{par}\text{'}\ (\mathit{fx}\ :\ \mathit{parMap} f\ xs) \\
\quad \textbf{where} & & \\
\qquad \mathit{fx}\ = f\ x & &
\end{array}
$$

The above version of *parMap* is *callee* parallel, and the function itself is responsible for the parallelisation of the structure. Compare the above to the version using the parallelisation technique using basic strategies:

$$
\begin{array}{lll}
\mathit{parList} & :: & [\alpha] \rightarrow () \\
\mathit{parList}\ [\,] & = & () \\
\mathit{parList}\ (x : xs) & = & x\ \text{`}\mathit{par}\text{'}\ (\mathit{parList} f\ xs)
\end{array}
$$

$$\mathit{map}_f\ xs\ \text{`}\mathit{using}\text{'}\ \mathit{parList}$$

Now our parallel map_f is *caller* parallel, the responsibility to parallelise lies with the function that *uses* the result of map_f.

Why is this an Issue?

As detailed in Chapter 5, the decision about whether to parallelise an expression is made based on the strictness properties of a function; additionally all parallelisation must be *known to be safe*. Using *caller* parallelism forces us to be *too safe*. For example, imagine that we are parallelising the function f which takes two list arguments and returns a list; importantly f only uses its first argument when the second argument is a *cons*:

$$
\begin{array}{lll}
f\ xs\ [\,] & = & [\,] \\
f\ xs\ ys & = & \langle \text{some expression using } xs \rangle
\end{array}
$$

Using *callee* parallelism we could spark the evaluation of *xs* within f's body. But with *caller* parallelism we are stuck *even if the demand on the result of f requires the full list!* This is because the second argument to f could be the empty list, making it unsafe for the *caller* to spark the evaluation of the first argument. This is quite a loss! And it is exactly what is happening in `MatMul`. To be more concrete, in `MatMul` we have *two* functions of the following form.[4]

[4] The functions are both defunctionalised *maps*: $\mathit{map}_{\mathit{mulRow}}$ and $\mathit{map}_{\mathit{dotProduct}}$.

```
map_f xs ys   -- xs is only used if ys is a cons
```

In both cases we are not able to spark a strategy to evaluate *xs* because doing so would be unsafe if *ys* is *nil*.

As an aside, converting the calls to these *map*s to *callee* parallelism (manually) is only useful for one of the functions ($map_{dotProduct}$). However, we would expect our iterative step to manage that aspect for us by turning off the parallelism that is not useful. The result of converting $map_{dotProduct}$ to *callee* parallelism was the only change needed in achieving the better speedup in Table 9 for `MatMul`.

8.5.2 *Lack of Speculation*

The other limitation in our technique is the lack of speculation. For example, take the following common idiom (present in our `Taut` benchmark):

```
case forall longlist of
  True  → e1
  False → e2

  where
    longlist = map f xs
```

Because we are in a non-strict language it is not safe to evaluate the *longlist* eagerly. This is because the very first element may be *False*, allowing us to ignore the rest of the list. However, when *f* is very expensive, it can be useful to 'look ahead' in the list and compute some elements of *longlist speculatively*. This is because if *f x* is expensive enough we would likely save time when elements of the list are *True*, necessitating us to continue evaluating the list.

Unlike the *caller/callee* distinction above, there is no way of making this idiom safe. Instead the compiler must annotate certain expressions as speculative, and allow the runtime system to ensure that the evaluation of these expressions does not introduce non-termination [Checkland, 1994; Mattson Jr, 1993]. One possible hint to the compiler is when an expression is not strictly needed, but *if* it is needed, it is needed to its full degree (such as in the *longlist* example above).

8.6 SUMMARY OF PROFILE-DIRECTED SEARCH

This chapter has provided a method to utilise runtime profile data in order to better search for an optimal `par` setting of a given program.

The use of profile information gives the hill-climbing algorithm an oracle that predicts which neighbour will be the most performant. The oracle is based on the concept of `par` site health, the mean of the work undertaken by all threads sparked by a `par`. This proves to be a good metric for our runtime system but does not translate naively to GHC.

The naive transfer to GHC is a disappointing result. That being said, iterative compilation techniques are usually performed on the same substrate as the final program. Using one runtime system (a simulator) and then transferring the results to another, completely different, runtime system proves to be too optimistic and does not correspond to the performance increases we would like.

We feel that performing the iterative step on GHC itself would likely provide better results, but testing this hypothesis requires adapting the compiler, static analysis, and runtime system of GHC in non-trivial ways.

In addition to performing the iteration on GHC itself it may be beneficial to abandon the bitstring representation of a program's `par` settings. In Section 8.4 we note that many of the performance issues when transferring to GHC can be fixed by switching certain `par`s to `seq`s. We may be able to adapt the search to begin by turning off the *obviously bad* `par`s and then perform the remaining search with each `par` site having three modes: on, off, and `seq`.

8.6.1 *Comparison to Heuristic Search*

In this chapter we explored using two heuristic search algorithms to accomplish our iterative step. While the results were promising, the ability to examine profile data should not be underestimated. The first benefit is that our search is guaranteed to be linear *in the worst case* and usually sub-linear, whereas even the greedy algorithm required exactly linear time.

As the programs grow larger the combinatorial explosion for the hill-climbing will become more and more detrimental. For the hill-climbing algorithm to terminate all neighbours of the current best candidate must be explored. This means that when a program has twenty `par` sites, the *last* iteration of the hill-climbing algorithm will require twenty evaluations of the program! The profile-based technique's ability to guarantee that this program will require *at most* twenty iterations is a huge advantage. This drastically reduces the cost of finding the worthwhile parallelism.

Part IV

Conclusions and Future Directions

9 CONCLUSIONS

> *In short, I think things are changing* [for implicit parallelism].
>
> – Peyton Jones [2005]

This chapter provides a summary and review of our contributions, allowing us to determine what was accomplished and what still requires work.

Because of the nature of our work, and of compilers in general, decisions about earlier phases of the compiler have consequences for the later stages, often forcing the hand of the latter phases of the compiler. For this reason we will review our work backwards, discussing the choices and trade-offs made in the reverse order of the compiler pipeline. This will allow us to consider alternative design choices and then consider what requirements are placed on the earlier stages of the compiler.

In Chapter 8 we showed that it is possible to use profiling data from the execution of functional programs to *refine* the parallelism that is introduced by a static analysis. Importantly, despite being a bitstring search, we were able to ensure that on average the search time is sublinear to the amount of parallelism (in the form of `par` annotations) that is introduced.

However, not all programs perform well and for this technique to work more generally it is clear that some form of *speculative* parallelism is needed. This is made clear by the disappointing performance of the `MatMul` program.

Additionally, when all things are considered, our method restricts the use of the profiling information passed back to the compiler after an iteration. The compiler can use profiling data to make decisions about *already existing* `par` sites, but can not decide to *create* new `par` sites or transform the program further. There is no fundamental reason this can not be done, but it would require a re-working of the static analysis and transformation phases of the compiler (Chapter

4 and 5) to somehow incorporate that information. Doing so, while maintaining flexibility, is more complex than our approach of simply modifying the bytecode produced by the compiler.

Chapter 7 searched over bitstrings using only the total running time of the program's execution. We experimented with two heuristic search algorithms: a greedy search that only considered each bit in the bitstring once and a traditional hill-climbing algorithm. This technique showed that even without the ability to measure detailed profiling information, iterative compilation can be beneficial. The downside of this simplicity is the additional number of iterations that must be performed. Using the simpler greedy algorithm allows us to cap the number of iterations to the number of `par` sites introduced, but as we saw with the `SodaCount` program, this can result in finding non-optimal (but pretty good) solutions.

Our experimental platform, as described in Chapter 6, is a fairly standard implementation of a lazy functional language. The limiting aspect of this implementation seems to be our need to defunctionalise. Our higher-order specialisation is suitable for exploratory purposes but is not appropriate for wider use. The Ur/Web compiler also requires a similar form of defunctionalisation (i.e. not Reynold's style defunctionalisation) [1] and has found success in using a modified version of a call-pattern specialisation [Peyton Jones, 2007] to produce first-order programs from higher-order programs. While admittedly not total, the author of Ur/Web has stated that they are not aware of a single 'real' program that this technique is unable to accommodate [Chlipala, 2015].

Chapter 5 introduces a method of automatically deriving parallel strategies from arbitrary demands. It would be desirable to have a proof that our technique is total and sound (all valid projections result in valid strategies). Before our work on automatically deriving strategies, parallelising programs automatically was limited by the evaluation methods that were hard-coded into the compiler, often based on Wadler's four-point domain [Hogen et al., 1992; Burn, 1987]. This limited the applicability of these techniques to list-based programs; an unnecessary limitation given the additional information that demand contexts provide.

[1] This was discussed during at hallway conversation at ICFP 2015.

Despite the lack of a formal proof, we feel that this is a useful contribution even outside the scope of implicit parallelism, we will discuss other applications in Chapter 10.

If we desire the ability to introduce new `pars` (or other similar extensions) during the iterative step (as mentioned above), the static-analysis phase of the compiler would need to be adapted. One idea would be to change the order of the `pars` in a strategy to reduce thread collisions, possibly using a *path analysis* [Bloss and Hudak, 1987]. Mostly, however, the principal of translating a projection into a strategy would remain the same.

Implicit parallelism is often cited as a pipe-dream that is unlikely to provide any real benefit to programmers [Peyton Jones, 1999; Marlow, 2013; Lippmeier, 2005; Hammond and Michelson, 2000]. We feel that we have demonstrated that the topic is worth pursuing with fresh eyes, and that in some cases a combination of static analysis and feedback-directed compilation can achieve speedups 'for free'. Processor utilisation[2] is much less important than it was when multiprocessor hardware was niche and expensive, pragmatic approaches that provide useful speedups, even if not for all programs, are worthwhile in the multicore era.

[2] Full utilisation of each processor in a multi-core system.

10

FUTURE DIRECTIONS

> *Parallelism is initiated by the* `par` *combinator in the source program. (At present these combinators are added by the programmer, though we would of course like this task to be automated.)*
>
> – Trinder et al. [1996]

One of the main motivators of this work was the lack of attention that implicit parallelism has been receiving in the functional programming community. Early on, functional programmers were optimistic about the feasibility and benefits of systems designed to exploit the inherent parallelism in our programs. The software and hardware ecosystem has changed beneath our feet since the 1980s and 1990s. Increasingly, software companies and developers prioritise developer productivity over software performance [Atwood, 2008]. The exploitation of implicit parallelism will allow developers to regain *some* of the performance without the cost of developer time.

One of our aims with this work was to show that despite setbacks in the past, there are still techniques and analyses worth exploring in this research area. We therefore turn our attention to the future and discuss some of the possible avenues of exploration.

Plan of the Chapter

There are many possible extensions and improvements that can be made to our general technique. The most direct would be to use the runtime information for function specialisation, similar to what we already do for the different demands on a function in Section 5.2.1. We discuss this idea in Section 10.1.

This thesis is predicated on the idea that implicit parallelism is our goal, and we still believe that this is a worthy pursuit. But it does not have to be *limited to* implicit parallelism. There may be situations where the programmer would like to specify that certain expressions

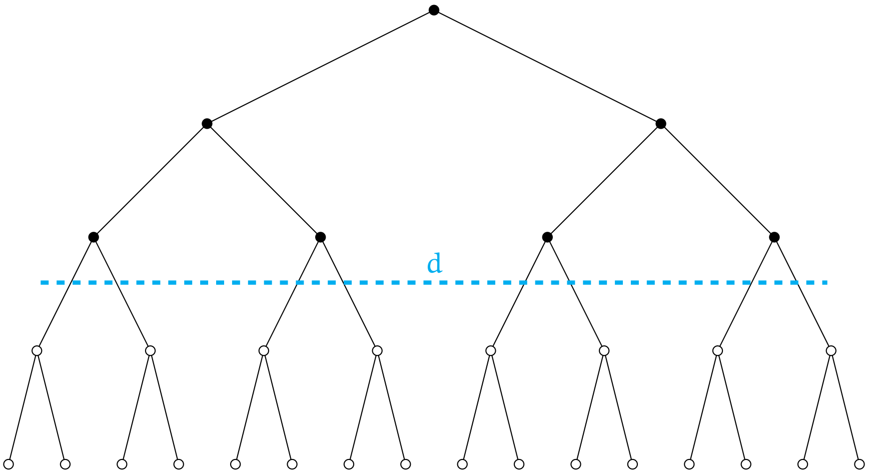


Figure 37: A tree representation of a recursive computation

are evaluated in parallel. We explore what such hybrid systems might look like in Section 10.2.

It is also possible that the demand properties of a program would be useful in identifying pipeline-parallelism automatically. Section 10.3 explores how one might design such a system.

10.1 SPECIALISING ON DEPTH

In our implementation we specialised functions in two ways: higher-order to first-order and `par` placement based on varied demands. There are, of course, many forms of specialisation that could be added: monomorphisation, call-pattern specialisation, and even specialisation to some depth of a recursive function.

When writing parallel programs we often write recursive functions, particularly divide-and-conquer algorithms, to take into account how deep into a computation a call is. This allows the program to avoid the creation of parallel threads when the computation is obviously (to the programmer) not worthwhile. An example of this was shown in Section 2.5.2 with our definition of *quicksort*. This pattern is not only limited to parallelising the more 'shallow' levels of the computation and then switching to a sequential version. It is possible that the leaves of a computation are the expensive part, and to limit the number of generated threads, one should begin with a sequential version and switch to a parallelised version below some depth. The general shape of the technique is drawn in Figure 37.

Given some depth d of the computation (usually measured in the number of recursive calls) for the function f, the shaded nodes represent calls to one specialised form of f and the unshaded nodes represent another specialisation of f. We can attain these two versions simply with the following transformation

$$
\begin{array}{lcl}
f\ x\ =\ \langle e \rangle & \Longrightarrow & f_1\ d\ x = \textbf{if}\ d\ >\ \mathit{depth}\ \textbf{then} \\
 & & \qquad f\ x \\
 & & \quad \textbf{else} \\
 & & \qquad \langle e_{[f \mapsto f_1\ (d+1)]} \rangle
\end{array}
$$

All original calls to f in the source program become f_1 0 and only below a certain depth do we begin to call the original function. What benefit does this give us? Now any `par` sites in f are duplicated in f_2 and can be switched independently based on the depth. This gives the iterative step the flexibility to switch off the `par`s for the levels of the call tree that are not worthwhile to perform in parallel.

In order to choose an appropriate value for *depth* in f_1 the runtime system must be equipped with some proxy for call-depth. This could take the form of something similar to what is used to get stack traces for lazy functional languages [Allwood et al., 2009].

We hypothesise that the value of *depth* in f_1 does not have to be perfect on the first attempt. By ensuring that a *reasonable depth* is chosen the compiler can then determine which `par`s, the ones in f_1 or the ones lower in the call-depth in f, should remain on. Once the on/off decisions have been made the compiler could attempt to *tune* the value of *depth*.

10.1.1 *Knowing When to Specialise on Depth*

A subtle point is knowing *when* this form of specialisation is useful. As with the main argument of this thesis, we would use both static and dynamic information about the program. We believe that candidate functions for specialisation on depth must exhibit *at least* the following properties:

1. The function must be recursive
2. The function must parallelise a recursive call to itself
3. The `par` site accomplishing the above has a wide distribution of `par` health (as exemplified by `par` site #8 in 32)

We believe that for large scale programs, specialising on depth will be necessary as divide-and-conquer algorithms are quite common.

10.2 HYBRID APPROACHES

We have presented our work as being fully automated and requiring no intervention by the programmer, but if we loosen that requirement on our technique we may be able to find a fruitful 'middle-ground' between fully-automated parallelism and programmer effort.

Super Strategies

One technique that we find promising is the idea of an automatic strategy.

It is common for a programmer to know that an algorithm should be parallelisable but does not want to invest the effort in parallelising the code by hand. A 'super-strategy' would allow the programmer to annotate the program, telling the compiler 'this expression should be parallelisable', but without specifying how. This saves the compiler from searching for parallelism throughout the whole program and iterating over the large resulting search space and instead focus its static analysis and iteration to the annotated expression.

This could be exposed to the programmer with an interface similar to parallel strategies [Trinder et al., 1998].

Something along the lines of:

$\langle$expression Alice would like to parallelise$\rangle$ *'using' autoStrat*

Or more conventionally:

autoPar $\langle$expression Alice would like to parallelise$\rangle$

The auto-strategy would still use the demand information available at compile time, and proceed in the manner outlined in this thesis, but the compiler would benefit from a much reduced search space.

10.3 AUTOMATING PIPELINE PARALLELISM

Using the projection-based strictness analysis to discover producer-consumer pairs we may be able to automatically utilise pipeline parallelism. Because of lazy evaluation we already benefit from a form of pipeline parallelism [Hughes, 1989]. However, because parallel expressions are not tied to specific execution contexts the producer and consumer threads can easily interrupt each other due to the runtime system scheduler. To prevent this we can use ideas from Hudak's *para-functional programming* that allow for expressions to be tagged with operational information. This could take the form of where in memory the expression should allocate data or which processors an expression should be tied to [Hudak, 1986].[1] If the functions are strict enough we could employ the techniques introduced by Marlow et al. in the Par Monad, which automatically schedules pipeline parallelism for values that can be fully evaluated [Marlow et al., 2011].

[1] Known as processor affinity in many modern systems.

A BENCHMARK PROGRAMS

A.1 SUMEULER

```haskell
import Prelude hiding ( foldr, map, sum
                      , length, filter)

myGcd x y = if y == 0
          then x
          else if x > y
               then myGcd (x - y) y
               else myGcd x (y - x)

relPrime x y = myGcd x y == 1

euler n = length $ filter (relPrime n) xs
  where xs = [1..n]

filter p []     = []
filter p (x:xs) = case p x of
                       True  -> x:filter p xs
                       False -> filter p xs

length [] = 0
length (x:xs) = (+) 1 (length xs)

map f []     = []
map f (x:xs) = f x : map f xs

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

sum xs  =  foldr plus 0 xs

main = print $ sum $ map euler [1..1000]
```

A.2 MATMUL

```
import Prelude hiding ( foldr, map, sum, null
                      , transpose, zipWith, replicate
                      )

null []      =  True
null (x:xs)  =  False

matMul xss yss  =  map (mulRow (transpose yss)) xss

mulRow yssTrans xs  =  map (dotProduct xs) yssTrans

dotProduct xs ys  =  sum (zipWith (*) xs ys)

transpose (r:rs) =  case null rs of
                        True -> map (:[]) r
                        False -> zipWith (:) r (transpose rs)

zipWith f []         []          =  []
zipWith f (x:xs) (y:ys)  =  f x y : zipWith f xs ys

sum xs  =  foldr (+) 0 xs

foldr f z []     =  z
foldr f z (x:xs)  =  f x (foldr f z xs)

map f []          =  []
map f (x:xs)  =  f x : map f xs

onesMat n  =  replicate n (replicate n 1)

replicate n x  =  case ((==) n 0) of
                  True  -> []
                  False -> x:(replicate ((-) n 1) x)

main  = print $ matMul (onesMat 50) (onesMat 50)
```

A.3 QUEENS

```
import Prelude hiding ( concatMap, length, and)

and False a = False
and True  a = a

append []     ys = ys
append (x:xs) ys = x : (append xs ys)

concatMap f []     = []
concatMap f (x:xs) = append (f x) (concatMap f xs)

length []     = 0
length (x:xs) = (+) 1 (length xs)

gen nq n = case (==) n 0 of
    True  -> []:[]
    False -> concatMap (gen1 nq) (gen nq ((-) n 1))

gen1 nq b = concatMap (gen2 b) (toOne nq)

gen2 b q = case safe q 1 b of
             True  -> (q:b) : []
             False -> []

safe x d [] = True
safe x d (q:l) =
  and ((/=) x q) (
  and ((/=) x ((+) q d)) (
  and ((/=) x ((-) q d)) (
  safe x ((+) d 1) l)))

toOne n = case (==) n 1 of
            True  -> 1:[]
            False -> n : toOne ((-) n 1)

nsoln nq = length (gen nq nq)

main = print $ nsoln 10
```

A.4 QUEENS2

```
import Prelude hiding ( foldr, foldl, map, concatMap, sum, tail
                      , null, length, transpose, reverse, zipWith
                      , const, replicate, flip, and)

data Test a  = A a | B a a | C

tail (x : xs) = xs

const a b = a

one p []       = []
one p (x : xs) = const (case p x of
                         True -> x : []
                         False -> one p xs) 0

map f []     = []
map f (x:xs) = f x : map f xs

append []     ys = ys
append (x:xs) ys = x : (append xs ys)

concatMap f []     = []
concatMap f (x:xs) = append (f x) (concatMap f xs)

length []     = 0
length (x:xs) = (+) 1 (length xs)

replicate n x =
  case (==) n 0 of
    True  -> []
    False -> x : replicate ((-) n 1) x

l = 0
r = 1
d = 2

eq x y = (==) x y
```

```
left  xs = map (one (eq l)) (tail xs)
right xs = [] : map (one (eq r)) xs
down  xs = map (one (eq d)) xs

merge []     ys     = []
merge (x:xs) []     = x : xs
merge (x:xs) (y:ys) = append x y : merge xs ys

next mask = merge (merge (down mask) (left mask)) (right mask)

fill []     = []
fill (x:xs) = append (lrd x xs) (map (x:) (fill xs))

lrd []     ys = [[l,r,d]:ys]
lrd (x:xs) ys = []

solve n mask =
  case (==) n 0 of
    True  -> []:[]
    False -> concatMap (sol ((-) n 1)) (fill mask)

foo n =
  case (<=) n 5 of
    True  -> B True False
    False -> C

sol n row = map (row:) (solve n (next row))

nqueens n = length (solve n (replicate n []))

main = print $ nqueens 10
```

A.5 SODACOUNT

```haskell
import Prelude hiding ( foldr, foldl, map, sum, tail, null
                      , transpose, reverse, zipWith
                      , flip)

tail (x:xs) = xs

null []     = True
null (x:xs) = False

single x = [x]

main = print $ map gridCount hidden

gridCount word  =
  let d = transpose grid
  in let r = grid
     in let dr = diagonals grid
        in let ur = diagonals (reverse grid)
           in let dirs = [r,d,dr,ur]
              in let drow  = reverse word
                 in (+) (sum (map (dirCount word) dirs))
                        (sum (map (dirCount drow) dirs))

sum xs  =  foldr plus 0 xs

plus x y = (+) x y

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

map f []     = []
map f (x:xs) = f x : map f xs

transpose (r:rs) = case null rs of
                        True  -> map single r
                        False -> zipWith (:) r (transpose rs)
```

```
diagonals (r:rs) = case null rs of
                       True  -> map single r
                       False -> zipInit r ([] : (diagonals rs))

reverse xs  =  foldl (flip (:)) [] xs

foldl f a []          =  a
foldl f a (x:xs)  =  foldl f (f a x) xs

flip f x y  =  f y x

zipWith f []         []          =  []
zipWith f (x:xs) (y:ys)  =  f x y : zipWith f xs ys

zipInit []         ys           =  ys
zipInit (x:xs) (y:ys)  =  (x:y) : zipInit xs ys

dirCount xs yss  =  sum (map (rowCount xs) yss)

rowCount xs ys  =  count (prefix xs) (suffixes ys)

count p []  =  0
count p (x:xs)  =
  let c  =  count p xs in
  case p x of
  True -> (+) 1 c
  False -> c

suffixes xs   =  case null xs of
                   True  -> []
                   False -> xs : suffixes (tail xs)

prefix []      ys      = True
prefix (x:xs) []       = False
prefix (x:xs) (y:ys)  = case ((==) x y) of
                             True  -> prefix xs ys
                             False -> False
```

```
grid =
  [ "YIOMRESKST"
  , "AEHYGEHEDW"
  , "ZFIACNITIA"
  , "NTOCOMVOOR"
  , "ERDLOCENSM"
  , "ZOURPSRNDA"
  , "OYASMOYEDL"
  , "RNDENLOAIT"
  , "FIWINTERRC"
  , "FEZEERFTFI"
  , "IIDTPHUBRL"
  , "CNOHSGEION"
  , "EGMOPSTASO"
  , "TGFFCISHTH"
  , "OTBCSSNOWI"
  ]

hidden =
  [ "COSY"
  , "SOFT"
  , "WINTER"
  , "SHIVER"
  , "FROZEN"
  , "SNOW"
  , "WARM"
  , "HEAT"
  , "COLD"
  , "FREEZE"
  , "FROST"
  , "ICE"
  ]
```

A.6 TAK

```
tak x y z = case (<=) x y of
            True  -> z
            False -> tak (tak ((-) x 1) y z)
                                     (tak ((-) y 1) z x)
                                     (tak ((-) z 1) x y)

main = print $ tak 24 16 8
```

A.7 TAUT

```
import Prelude hiding ( foldr1, map, length
                      , zip, filter, flip, and
                      )

data Pair a b = P a b
data Prop     = And Prop Prop
              | Const Bool
              | Implies Prop Prop
              | Not Prop
              | Var Char

find key ((P k v):t) = case (==) key k of
                               True  -> v
                               False -> find key t

eval s (Const b)       = b
eval s (Var x)         = find x s
eval s (Not p)         = case eval s p of
                           True  -> False
                           False -> True
eval s (And p q)       = case eval s p of
                           True  -> eval s q
                           False -> False
eval s (Implies p q)   = case eval s p of
                           True  -> eval s q
                           False -> True

vars (Const b)     = []
vars (Var x)       = [x]
vars (Not p)       = vars p
vars (And p q)     = append (vars p) (vars q)
vars (Implies p q) = append (vars p) (vars q)

bools n = case (==) n 0 of
          True  -> []:[]
          False -> let bss = bools ((-) n 1) in
                   append (map (False:) bss)
                          (map (True:)  bss)
```

```
neq x y = (/=) x y

rmdups []     = []
rmdups (x:xs) = x:rmdups (filter (neq x) xs)

substs p = let vs = rmdups (vars p) in
           map (zip vs) (bools (length vs))

isTaut p = and (map (flip eval p) (substs p))

flip f y x = f x y

length []     = 0
length (x:xs) = (+) 1 (length xs)

append []     ys = ys
append (x:xs) ys = x:append xs ys

map f []     = []
map f (x:xs) = f x : map f xs

and []     = True
and (b:bs) = case b of
                  True  -> and bs
                  False -> False

filter p []     = []
filter p (x:xs) = case p x of
                       True  -> x:filter p xs
                       False -> filter p xs

null []     = True
null (x:xs) = False

zip []     ys     = []
zip (x:xs) []     = []
zip (x:xs) (y:ys) = (P x y) : (zip xs ys)
```

```
foldr1 f (x:xs) = case null xs of
                       True  -> x
                       False -> f x (foldr1 f xs)

imp v = Implies (Var ’p’) (Var v)

names = "abcdefghijklmn"

testProp = Implies
             (foldr1 And (map imp names))
             (Implies (Var ’p’) (foldr1 And (map Var names)))

main = print $ case isTaut testProp of
       True  -> 1
       False -> 0
```

B

LAZIFY EXPRESSIONS

Because our projection-based strictness analysis requires that our programs have the suspension and forcing of thunks *explicitly* annotated in the AST we require a method to convert the standard F-lite AST from Section 3.1 into an intermediate language with explicit thunks (via *Freeze* and *Unfreeze*. The functions in Figure 38 accomplish this task. The rules are adapted from a function performing the same task in Hinze [1995, Section 2.3].

$$
\begin{aligned}
&\mathcal{L} && :: \mathit{Exp} \rightarrow \mathit{Exp} \\
&\mathcal{L}\,\llbracket \mathsf{Var\ v} \rrbracket && = \mathit{Unfreeze}\ (\mathit{Var}\ v) \\
&\mathcal{L}\,\llbracket \mathsf{Int\ i} \rrbracket && = \mathit{Int}\ i \\
&\mathcal{L}\,\llbracket \mathsf{Con\ c} \rrbracket && = \mathit{Con}\ c \\
&\mathcal{L}\,\llbracket \mathsf{App\ (Fun\ f)\ args} \rrbracket && = \mathit{App}\, f\ \$\ \mathit{map}\ (\mathit{Freeze}\ .\ \mathcal{L})\ \mathit{args} \\
&\mathcal{L}\,\llbracket \mathsf{Let\ bs\ in\ e} \rrbracket && = \mathit{Let}\ (\mathcal{L}'\,\llbracket \mathsf{bs} \rrbracket)\ (\mathcal{L}\,\llbracket \mathsf{e} \rrbracket) \\
&\mathcal{L}\,\llbracket \mathsf{Case\ e\ alts} \rrbracket && = \mathit{Case}\ (\mathcal{A}\,\llbracket \mathsf{e} \rrbracket)\ (\mathcal{L}'\,\llbracket \mathsf{alts} \rrbracket)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{L}' && :: [(\alpha, \mathit{Exp})] \rightarrow [(\alpha, \mathit{Exp})] \\
&\mathcal{L}'\,\llbracket \mathsf{ts} \rrbracket && = [(x, \mathcal{L}\,\llbracket \mathsf{e} \rrbracket) \mid (x, e) \leftarrow \mathit{ts}]
\end{aligned}
$$

Figure 38: Convert an expression to one with explicit suspension and forcing of thunks

Sometimes performing this AST pass results in redundant sequences of $\mathit{Freeze}(\mathit{Unfreeze}\ e)$. These can be safely converted to e [Hinze, 1995, pg. 36].

BIBLIOGRAPHY

Samson Abramsky. The Lazy Lambda Calculus. *Research Topics in Functional Programming*, 65, 1990.

Tristan O.R. Allwood, Simon Peyton Jones, and Susan Eisenbach. Finding the Needle: Stack Traces for GHC. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 129–140, New York, NY, USA, 2009. ACM.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

Jeff Atwood. Coding Horror: Hardware is Cheap, Programmers are Expensive. http://blog.codinghorror.com/hardware-is-cheap-programmers-are-expensive/, Dec 2008. [Online; accessed 25-September-2015].

Lennart Augustsson and Thomas Johnsson. The Chalmers Lazy ML-compiler. *Computer Journal*, 32(2):127–141, April 1989a.

Lennart Augustsson and Thomas Johnsson. Parallel Graph Reduction with the $\langle \nu, G \rangle$-Machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 202–213, New York, NY, USA, 1989b. ACM.

Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, and Hans-Wolfgang Loidl. A Survey of High-Level Parallel Programming Models. Technical Report HW-MACS-TR-0103, 2013.

Richard Bird. *Thinking functionally with Haskell*. Cambridge University Press, 2014.

A. Bloss and Paul Hudak. Path Semantics. In *Proceedings of Third Workshop on the Mathematical Foundations of Programming Language Semantics*, pages 476–489. Tulane University, Springer-Verlag LNCS Volume 298, 1987.

Geoffrey L. Burn. Evaluation Transformers–A Model for the Parallel Evaluation of Functional Languages. In *Functional Programming Languages and Computer Architecture*, pages 446–470. Springer, 1987.

Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness Analysis for Higher-order Functions. *Science of Computer Programming*, 7:249–278, 1986.

Geoffrey L. Burn, Simon Peyton Jones, and J.D. Robson. The Spineless G-Machine. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, pages 244–258. ACM, 1988.

Iain Gavin Checkland. *Speculative Concurrent Evaluation in a Lazy Functional Language*. PhD thesis, University of York, 1994.

Adam Chlipala. An Optimizing Compiler for a Purely Functional Web-application Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 10–21, New York, NY, USA, 2015. ACM.

Alonzo Church and J Barkley Rosser. Some Properties of Conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

Chris Clack. Realisations for Non-Strict Languages. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 6. Springer, 1999.

Chris Clack and Simon Peyton Jones. Strictness Analysis-A Practical Approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49. Springer, 1985.

Chris Clack and Simon Peyton Jones. The Four-Stroke Reduction Engine. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 220–232. ACM, 1986.

Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The Architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA, 2009. ACM.

Benjamin Goldberg. Buckwheat: Graph Reduction on a Shared-Memory Multiprocessor. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 40–51, New York, NY, USA, 1988. ACM.

Kevin Hammond. Parallel Functional Programming: An Introduction. www-fp.dcs.st-and.ac.uk/~kh/papers/pasco94/pasco94.html, 1994.

Kevin Hammond and Greg Michelson. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 2000.

Kevin Hammond and Álvaro J. Rebón Portillo. Haskskel: Algorithmic skeletons in haskell. In Pieter Koopman and Chris Clack, editors, *Implementation of Functional Languages*, volume 1868 of *Lecture Notes in Computer Science*, pages 181–198. Springer Berlin Heidelberg, 2000.

Tim Harris and Satnam Singh. Feedback Directed Implicit Parallelism. 42(9):251–264, October 2007. ISSN 0362-1340.

Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a Shared-memory Multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 49–61, New York, NY, USA, 2005. ACM.

P. Harrison and M. Reeve. The Parallel Graph Reduction Machine, ALICE. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 181–202. Springer Berlin / Heidelberg, 1987.

Ralf Hinze. Projection-based Strictness Analysis: Theoretical and Practical Aspects, 1995. Inaugural Dissertation, University of Bonn.

Guido Hogen, Andrea Kindler, and Rita Loogen. Automatic Parallelization of Lazy Functional Programs. In *ESOP'92*, pages 254–268. Springer, 1992.

Paul Hudak. Para-functional Programming. *Computer Journal*, 19(8), 1986.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

John Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Programming Research Group, Oxford University, July 1983.

John Hughes. Strictness detection in non-flat domains. In *Programs as Data Objects*, pages 112–135. Springer, 1985.

John Hughes. Analysing Strictness by Abstract Interpretation of Continuations. In *Abstract Interpretation of Declarative Languages*, pages 63–102. Ellis Horwood Chichester, 1987.

John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

Clèment Hurlin. Automatic Parallelization and Optimization of Programs by Proof Rewriting. Technical Report RR-6806, 2009.

Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel Performance Tuning for Haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 81–92, New York, NY, USA, 2009. ACM.

Mark Jones and Paul Hudak. Implicit and Explicit Parallel Programming in Haskell. *Research Report YALEU/DCS/RR-982*, 1993.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, Shape-Polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, volume 45 of *ICFP 2010*, pages 261–272. ACM, 2010.

Donald E. Knuth. Textbook Examples of Recursion. In *In Artificial Intelligence and Theory of Computation*, pages 207–229. Academic Press, 1991.

Ryszard Kubiak, John Hughes, and John Launchbury. Implementing Projection-Based Strictness Analysis. In *Functional Programming, Glasgow 1991*, pages 207–224. Springer, 1992.

Ben Lippmeier. [Haskell] Implicit parallel functional programming. https://mail.haskell.org/pipermail/haskell/2005-January/015213.html, Jan 2005. [Online; accessed 21-September-2015].

Hans Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Citeseer, 1998.

Rita Loogen. Programming Language Constructs. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 3. Springer, 1999.

Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

Simon Marlow and Simon Peyton Jones. Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

Simon Marlow, A.R. Yakushev, and Simon Peyton Jones. Faster Laziness Using Dynamic Pointer Tagging. pages 277–288, 2007.

Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, volume 44 of *ICFP 2009*, pages 65–78. ACM, 2009.

Simon Marlow, P. Maier, Hans Wolfgang Loidl, M.K. Aswad, and Philip W. Trinder. Seq No More: Better Strategies for Parallel Haskell. In *Proceedings of the 3rd ACM Symposium on Haskell*, pages 91–102. ACM, 2010.

Simon Marlow, Ryan Newton, and Simon Peyton Jones. A Monad for Deterministic Parallelism. In *ACM SIGPLAN Notices*, volume 46, pages 71–82. ACM, 2011.

James S Mattson Jr. *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction*. PhD thesis, University of California at San Diego, 1993.

Matthew Might and Tarun Prabhu. Interprocedural Dependence Analysis of Higher-Order Programs via Stack Reachability. In *Proceedings of the 2009 Workshop on Scheme and Functional Programming*, pages 10–22, 2009.

Alan Mycroft. The Theory and Practice of Transforming Call-by-Need Into Call-by-Value. In *International Symposium on Programming*, pages 269–281. Springer, 1980.

Matthew Naylor and Colin Runciman. The reduceron reconfigured. pages 75–86, 2010.

Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

B. O'Sullivan, D.B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2009.

Bryan O'Sullivan. Criterion: A Haskell Microbenchmarking kibrary. https://hackage.haskell.org/package/criterion, 2009.

Ross Paterson. Compiling Laziness Using Projections. In Radhia Cousot and DavidA. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 255–269. Springer Berlin Heidelberg, 1996.

Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

Simon Peyton Jones. Parallel Implementations of Functional Programming Languages. *Computer Journal*, 32(2):175–186, April 1989.

Simon Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

Simon Peyton Jones. Foreward. In Kevin Hammond and Greg Michaelson, editors, *Research Directions in Parallel Functional Programming*. Springer, 1999.

Simon Peyton Jones. [Haskell] Implicit parallel functional programming. https://mail.haskell.org/pipermail/haskell/2005-January/015218.html, Jan 2005. [Online; accessed 21-September-2015].

Simon Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 327–337, New York, NY, USA, 2007. ACM.

Simon Peyton Jones and David R. Lester. *Implementing Functional Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *Journal of Functional Programming*, 12(4-5):393–434, 2002.

Simon Peyton Jones and André L M Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1):3–47, 1998.

Simon Peyton Jones, Chris Clack, and J. Salkild. GRIP: A High Performance Architecture for Parallel Graph Reduction. In *Functional Programming Languages and Computer Architecture: Third International Conference (Portland, Oregon)*. Springer Verlag, 1987.

Rinus Plasmeijer and Marko Van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1993.

John C. Reynolds. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

Colin Runciman and David Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines). In *Functional Programming, Glasgow 1993*, pages 236–251. Springer, 1994.

Colin Runciman and David Wakeling, editors. *Applications of Functional Programming*. UCL Press Ltd., London, UK, 1996.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones. Modular, Higher-order Cardinality Analysis in Theory and Practice. In *Proceedings of the 41st ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '14, pages 335–347, New York, NY, USA, 2014. ACM.

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 205–217, New York, NY, USA, 2015. ACM.

Herb Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

Guy Tremblay and Guang R Gao. The Impact of Laziness on Parallelism and the Limits of Strictness Analysis. In *Proceedings High Performance Functional Computing*, pages 119–133, 1995.

Philip W Trinder, Kevin Hammond, James S Mattson Jr, Andrew S Partridge, and Simon Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *PLDI '96: Proceedings of the 1996 Conference on Programming Language Design and Implementation*, pages 79–88. ACM, 1996.

Philip W. Trinder, Kevin Hammond, Hans Wolfgang Loidl, and Simon Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

David Turner. Some History of Functional Programming Languages. In *Symposium on the trends in functional programming 2012*, 2012.

Philip Wadler. Strictness Analysis on Non-Flat Domains. In *Abstract Interpretation of Declarative Languages*, pages 266–275. Ellis Horwood, 1987.

Philip Wadler and John Hughes. Projections for Strictness Analysis. In *Functional Programming Languages and Computer Architecture*, pages 385–407. Springer, 1987.

C. P. Wadsworth. *Semantics and Pragmatics of the λ-Calculus*. PhD thesis, Programming Research Group, Oxford University, 1971.