Submitted in part fulfilment for the degree of MEng.

Optimus Prime:  
A new tool for interactive transformation and supercompilation of functional programs

Jason S. Reich

12th May 2009

Supervisor: Professor Colin Runciman

Number of words = 19569, as counted by `detex | wc -w`
Number of pages = 70.
This includes the body of the report, and Appendix A and Appendix E, but not other appendices, the bibliography and intentionally blank pages.
Abstract

This dissertation investigates the use of supercompilation to improve the performance of functional programs. We apply the transformation strategy to the f-lite subset of Haskell 98. Supercompilation is found to be an effective optimisation for f-lite programs and areas for further improvements are suggested.

An interactive transformation tool, Optimus Prime, is produced to aid the development of the supercompiler. The tool facilitates the observation of program transformation strategies in progress and we gain the ability to operate on programs directly by manually applying transformations.

A key technique, used in the implementation of Optimus Prime, is the Syntactic Zipper structure. We conclude that while the use of this data structure is nearly essential for interactive program transformation, it becomes a performance limitation for fully automatic transformation strategies.

Keywords: functional programming, program transformation, interactive transformation, supercompilation, zipper data structure, Haskell, Reduceron, f-lite
Acknowledgements

Many thanks go to Professor Colin Runciman for his support, guidance, good humour and RunciScribble™ through the course of this project.

My gratitude to Matthew Naylor for providing the project setting and his assistance with running Reduceron 2 f-lite code.

I am much obliged to Neil Mitchell his willingness to discuss his work on Supercompilation.

A solemn nod to the many programs who willingly dedicated their lives to science, only to be mangled into a non-executable form by early versions of Optimus Prime.

Finally, three cheers to the occupants of the CS/001 Masters’ Laboratory for their motivational techniques and other interesting activities.
# Contents

1 Introduction .................................................. 15  
1.1 Motivation .................................................. 15  
1.2 Aims of this Project ................................. 16  
1.3 Structure of this Report ............................ 16  
1.4 Statement of Ethics ............................... 17  

2 Background and Review .................................... 18  
2.1 Functional Languages and Haskell ......... 18  
2.2 Program Transformation ............................ 21  
   2.2.1 The Fold/Unfold Laws ....................... 22  
   2.2.2 Defunctionalisation ......................... 22  
   2.2.3 Deforestation and Fusion ................. 24  
   2.2.4 Supercompilation ......................... 24  
2.3 Program Transformation Tools .............. 25  
2.4 The Reduceron and f-Lite ...................... 28  
2.5 Summary .............................................. 30  

3 Problem Analysis ........................................ 31  
3.1 Specification of Transformations .......... 31  
3.2 The Core Language ................................. 31  
   3.2.1 Preliminary Definitions .................... 33  
3.3 Transformation Primitives for f-lite ...... 34  
   3.3.1 Inline / Unfold ............................... 34  
   3.3.2 Fold ............................................ 35  
   3.3.3 Residuate ..................................... 35  
3.4 The Fold/Unfold Laws ......................... 35  
3.5 Supercompilation Strategy ..................... 38  
   3.5.1 Termination Criteria ......................... 38  
   3.5.2 Unfolding Selection ......................... 39  
   3.5.3 Simplification Transformations for f-lite .. 39  
   3.5.4 Residuation, Folding and Generalisation .. 42  
   3.5.5 Inlining Unnecessary Residuals .......... 43  
3.6 Requirements A Transformation Framework .. 43  
3.7 Summary .............................................. 44  

4 Design and Implementation ......................... 46  
4.1 Representation through Syntactic Zippers .. 46  
4.2 Zipper Query and Modification .......... 50  
4.3 Transformation Specification ............... 55
## Contents

4.4 Transformation Specifics .................................................. 57
  4.4.1 Low-Level Definitions ................................................. 57
  4.4.2 Transformation Rules and Strategies ............................. 58

4.5 Output and Control ...................................................... 60
  4.5.1 Pretty printing ........................................................... 60
  4.5.2 Interactive Control .................................................... 61
  4.5.3 Autonomous Control ................................................... 62

4.6 Summary ................................................................. 62

5 Results and Evaluation .................................................. 63
  5.1 Performance of the Syntactic Zipper Library ...................... 63
  5.2 Performance of the Supercompiler .................................... 63
    5.2.1 Test Programs ...................................................... 64
    5.2.2 The Supercompilation Process .................................. 65
    5.2.3 The Supercompiled Programs ................................. 65
    5.2.4 Discussion ........................................................... 65
  5.3 Summary ................................................................. 67

6 Conclusions and Further Work .......................................... 68
  6.1 Satisfaction of Objectives ............................................. 68
  6.2 Taking Zippers out of Context ....................................... 69
  6.3 Optimisation of the Homeomorphic Embedding Relation ........ 69
  6.4 Supercompilation of Primitive Functions ......................... 69
  6.5 Closing Remarks ........................................................ 70

A Fold/Unfold Transformation Transcripts ............................. 71
  A.1 Fibonacci Transformation Transcript .................................. 71
  A.2 Append/Elem Proof Transcript ......................................... 72

B Tree Traversal Comparison ................................................ 74

C Test Programs ........................................................................... 77
  C.1 Perms — Permutation Counting ....................................... 77
  C.2 Fibonacci — Calculation of Fibonacci Numbers ................. 79
  C.3 Twizzler — The Twizzler Problem ................................. 80
  C.4 Word Count — Counting Words in a String ....................... 82
  C.5 N-Queens — $N = 10$ Instance .................................. 83

D Supercompilation Report for Word Count ............................... 85

E Supercompiled Word Count .................................................. 90

Bibliography ............................................................................. 92
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>An example of a tree transformation</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Compact notation for the transformation of Figure 3.1</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Syntax of our core language</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>The children of f-lite expressions</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Definition of <code>inline</code></td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of variable capture from a naive inlining</td>
<td>35</td>
</tr>
<tr>
<td>3.7</td>
<td>Definition of <code>fold</code></td>
<td>36</td>
</tr>
<tr>
<td>3.8</td>
<td>Definition of <code>residuate</code></td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td>Root expressions that will cause simple termination</td>
<td>38</td>
</tr>
<tr>
<td>3.10</td>
<td>The homeomorphic embedding relation, <code>homeo</code> or <code>&lt;</code></td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>The expression similarity relation, <code>∼</code></td>
<td>39</td>
</tr>
<tr>
<td>3.12</td>
<td>Simplification laws for f-lite</td>
<td>40</td>
</tr>
<tr>
<td>3.13</td>
<td>New and augmented simplification transformations for f-lite</td>
<td>41</td>
</tr>
<tr>
<td>3.14</td>
<td>Most specific generalisation rewrite rule</td>
<td>42</td>
</tr>
<tr>
<td>3.15</td>
<td>Requirements for Optimus Prime</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Optimus Prime architecture</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Example of a zipper structure</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of scoping issues</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>The supercompilation algorithm</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>The interface of Optimus Prime</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Figures
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Transformation variable conventions</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of tree query and modification techniques</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Effects of supercompilation on code</td>
<td>66</td>
</tr>
<tr>
<td>5.3</td>
<td>Supercompilation simplification statistics</td>
<td>66</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance of supercompiled programs</td>
<td>66</td>
</tr>
</tbody>
</table>
List of Tables
List of Listings

2.1 Examples of higher order functions ........................................ 19
2.2 Example of the effect of evaluation models ................................. 19
2.3 Infinite fibonacci sequence ................................................... 20
2.4 Example of overloading leaves using a typeclass .......................... 20
2.5 Monad typeclass ............................................................. 21
2.6 The Maybe monad ............................................................ 21
2.7 Using fold/unfold to compose two functions ............................... 23
2.8 Another example of a higher order function (Mitchell, 2007) .......... 23
2.9 Reynolds’ style defunctionalisation (Mitchell, 2007) ...................... 23
2.10 Mitchell’s style defunctionalisation (Mitchell, 2007) ..................... 23
2.11 Non-treeless and treeless definitions of flatten (Wadler, 1990) ....... 24
2.12 Supercompiling to perform fusion ......................................... 26
2.13 AST for the f-lite syntax ................................................... 29
2.14 Examples of Naylor’s transformations ..................................... 30

3.1 Examples of abstraction in f-lite ............................................ 37

4.1 Syntactic zipper types ....................................................... 47
4.2 Primitive syntactic zipper operations ...................................... 49
4.3 BindingMap data structure .................................................. 50
4.4 Zipper structure for f-lite ................................................... 51
4.5 Zipper query functions ........................................................ 51
4.6 Example of zipper query using list comprehensions ...................... 51
4.7 Zipper modification functions .............................................. 52
4.8 Example of boilerplate code ............................................... 53
4.9 Uniplate representation of Listing 4.8 ..................................... 53
4.10 Our zipper representation of [Listing 4.8] ................................ 54
4.11 Complex empty let removal for f-lite .................................... 55
4.12 Simpler empty let removal for f-lite ..................................... 56
4.13 Monadic empty let removal for f-lite .................................... 56
4.14 VariableMap state .......................................................... 57
4.15 Preliminary definitions ...................................................... 58

A.1 An example of optimising a function using fold/unfold, expressed in f-lite 71
A.2 Example of a fold/unfold proof, held in f-lite syntax ..................... 72

B.1 Code for tree traversal tests ................................................ 74

C.1 Code for the ‘perms’ test program ....................................... 77
C.2 Code for the ‘fibonacci’ test program ................................... 79
List of Listings

C.3 Code for the ‘twizzler’ test program ........................................... 80
C.4 Code for the ‘wc’ test program .................................................... 82
C.5 Code for the ‘nqueens’ test program ............................................ 83

E.1 Supercompiled code for the ‘word count’ test program ................. 90
1 Introduction

This project is concerned with interactive and automatic program transformation in a functional language.

Programs are descriptions of computations, represented in a language with formal syntactic and semantic models. A transformation is an alteration of a program’s syntax, such that the transformed program is semantically equivalent to the original. Transformations can introduce elementary changes, such as the renaming functions and variables, or more elaborate changes, such as optimisation for a platform’s characteristics.

A variety of programming language styles are available. The large majority of mainstream languages are described as imperative. These emphasise a “style in which programs execute commands sequentially, use variables to organise memory and update variables with assignment statements.” (Finkel, 1995) Another, less common, paradigm is functional programming. Computation is expressed as the evaluation of functions where the concepts of an implicit state and side effects are eliminated.

Proponents argue that, beyond the novelty of functional languages, “programs can be written quicker, are more concise, are higher level, are more amenable to formal reasoning and analysis and can be executed more easily on parallel architectures.” (Hudak, 1989) It is the compliance with formal reasoning that makes this project feasible.

A purely functional language provides referential transparency, “in which equals can be replaced by equals.” (Hudak, 1989) This is because immutable functions will always return the same result for a given input. Transformations can be safely performed as any expression can be replaced with its definition. The implications of referential transparency are discussed in Section 2.1.

We apply this principle to produce a tool, Optimus Prime, that allows the user to transform a program interactively. The transformations do not change the output of the program but do alter the execution characteristics.

To demonstrate the power of such a tool, we implement the supercompilation (Turchin, 1979) optimisation strategy and use it to improve the performance of programs automatically, without human guidance.

1.1 Motivation

Program transformations are often applied by a compiler to optimise readable code into a more efficient form. While reasoning about a transformation can prove its semantic equivalence, one will still need to experiment with the application of the transformation (and transformation strategy) to find the conditions under which an improvement is made.

An interactive transformation tool allows a user to manually apply transformations. The resulting code will always be executable as semantic equivalence is maintained. The user can then observe the efficiency of the executing program at any given stage.
Introduction

The Reduceron\textsuperscript{2} (Naylor and Runciman, 2008) is a FPGA graph reduction machine for executing a subset of Haskell (Simon Peyton Jones et al., 2003), f-lite. The machine is a quite sophisticated in its approach, but the current f-lite compiler is very simple and performs only basic optimisation.

In principle, program transformation strategies can be applied to optimise code to run more efficiently on this type of platform. Supercompilation is one strategy that could be applied. Questions include: (a) What extensions are required for it operate on the f-lite language and (b) Will lead to performance improvements on this architecture?

1.2 Aims of this Project

This project aims to do the following:

1. to develop an interactive transformation tool, Optimus Prime, that can operate on the f-lite subset of Haskell 98,

2. to apply the fold/unfold program transformations to the f-lite language, to be used for intuition-guided optimisation, theorem proving and as the basis of more complex program transformation strategies,

3. to investigate the complications of applying supercompilation to the f-lite language and where extensions may be required,

4. to implement a supercompiler for the f-lite language using the Optimus Prime framework and

5. to measure the costs of supercompilation and investigate whether it produces a distinct improvement to the performance of f-lite programs.

1.3 Structure of this Report

Chapter 2 — Background and Review A review of key concepts and existing literature is presented. Functional languages are introduced and particular features of interest are highlighted. Program transformations, specific examples and tools for performing transformations are discussed. Finally, the reader is acquainted with the Reduceron\textsuperscript{2} and the f-lite language.

Chapter 3 — Problem Analysis Investigates how to formally define a program transformation. A core language is defined and evaluated. Primitive transformations, the fold/unfold laws and the supercompilation strategy are applied to our core language. Finally, we discuss the requirements for our transformation tool.

Chapter 4 — Design and Implementation A data structure for representing a program inside an interactive transformation tool is presented. Operations upon this data structure are defined and evaluated. We discuss how to specify transformations and strategies within the tool. Finally, we investigate the formatting of programs to be returned by the tool and how the user may interact with the system.
1.4 Statement of Ethics

Chapter 5 — Results and Evaluation  We evaluate the performance of our Syntactic Zipper library and its suitability for our needs. The performance of the Supercompilation process is assessed in terms of speed of compilation and optimisation of programs.

Chapter 6 — Conclusions and Further Work  The project’s objectives are revisited and our success in satisfying them is assessed. We conclude that there is indeed scope for using supercompilation to optimise programs for the Reducer 2 platform. Areas for further research to improve the speed of supercompilation and the performance of optimised programs are presented.

1.4 Statement of Ethics

This project adheres to the ACM Code of Ethics (Anderson, 1992) and departmental guidelines. The project is concerned with the production of a software tool where no external human participants were required for testing. The tool is provided as-is and should be considered of experimental quality.
2 Background and Review

This chapter will introduce a number of the key concepts that are investigated and applied through the course of this project. Existing literature is examined to extract any beneficial experience and knowledge.

Section 2.1 demonstrates a selection of functional programming concepts and introduces the Haskell language. The functional language constructs described are either targets for optimisation or structures used to implement the system.

Transformations and their practical applications are discussed in Section 2.2 along with a few concrete examples of program transformations in functional languages. Existing tools for performing program in both autonomous and interactive fashions are investigated in Section 2.3.

Finally, in Section 2.4 the reader is introduced to Naylor’s (2008) Reducer 2 and f-lite. The f-lite subset of Haskell is the language of programs programs to be transformed and the Reducer 2 is the architecture upon which the code will be executed.

2.1 Functional Languages and Haskell

**Functional languages** Functional languages are characterised by de-emphasising or eradicating mutable data and ‘side effects’ by appealing to the mathematical definition of a function.

Proponents (Hudak, 1989) argue that programs in functional languages are more responsive to formal reasoning and encourage more concise program code. Constructs in functional languages also map well onto parallel architectures.

Several general purpose functional languages exist, including Miranda (Turner, 1990), ML (Milner et al., 1990), LISP (Steele, 1990), Erlang (Armstrong, 2007), Clean (van Eekelen and Plasmeijer, 2001) and Microsoft F# (Syme). MATLAB, XSLT and Microsoft Excel formulae are all examples of domain specific functional languages.

Some of these languages borrow concepts from Church (1936) lambda calculus, a formal logic describing functions, applications and recursion. In lambda calculus, expressions are reduced using three rules; alpha, beta and eta reduction. \( \alpha \)-conversion allows the names of any bound variables to be changed. \( \beta \)-reduction is function application by substituting a lambda’s variable with arguments. \( \eta \)-equivalence is where two expressions produce the results.

**Haskell** Haskell (Simon Peyton Jones et al., 2003) is a “general purpose, purely functional programming language”. It combines a number of emerging and interesting concepts in the field of functional languages such as; higher order functions, non-strict semantics and pattern matching. The subset, f-lite (see Section 2.4) retains these features. Monads and typeclasses are Haskell constructs that are not included in f-lite but are used in our implementation (see Chapter 4).
### 2.1 Functional Languages and Haskell

Higher order functions

A higher order function (or functional) is a function on a function. Examples include `map`, `foldr` and function composition. Example definitions of these higher order functions can be found in Listing 2.1.

Lazy evaluation

Non-strict evaluation is a reduction strategy where the outermost reducible expression is reduced first. Haskell, specifically, uses graph reduction to ensure that while expressions are only evaluated as required, they are also only evaluated once for any shared use. Other terms for this strategy include lazy, delayed and call-by-need evaluation.

In a call-by-value language, the function `f` in Listing 2.2 would not terminate because it needs to evaluate `loop` before it can evaluate `const`. A call-by-need language, such as Haskell, does not need the value of `loop` and terminates with the result 1.

Hudak et al. (1999) highlights that a significant advantage of Haskell’s non-strict nature is that data constructors are also non-strict. This allows the definition of infinite data structures. Despite the structure being infinite in size (taking an infinite amount of time and space to calculate in its entirety), the needed parts can be computed in finite time.

Listing 2.3 illustrates an example of an infinite data structure holding the Fibonacci sequence.

Typeclasses

Typeclasses are Haskell’s interpretation of function overloading. A typeclass defines a “set of functions that can have different implementations depending on the type of data they are given” (O’Sullivan et al. 2008). Listing 2.4 shows a typeclass, Tree, that defines

---

**Listing 2.1: Examples of higher order functions**

```haskell
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)

(.) f g x = f (g x)
```

---

**Listing 2.2: Example of the effect of evaluation models**

```haskell
loop = loop
const x y = x

f = const 1 loop

-- Under call-by-need semantics, f evaluates to 1.
-- Under call-by-value semantics, f does not terminate.
```

---

**Higher order functions**
A higher order function (or functional) is a function on a function. Examples include ‘map’, ‘foldr’ and function composition. Example definitions of these higher order functions can be found in Listing 2.1.

**Lazy evaluation**
Non-strict evaluation is a reduction strategy where the outermost reducible expression is reduced first. Haskell, specifically, uses graph reduction to ensure that while expressions are only evaluated as required, they are also only evaluated once for any shared use. Other terms for this strategy include lazy, delayed and call-by-need evaluation.

In a call-by-value language, the function `f` in Listing 2.2 would not terminate because it needs to evaluate `loop` before it can evaluate `const`. A call-by-need language, such as Haskell, does not need the value of `loop` and terminates with the result 1.

Hudak et al. (1999) highlights that a significant advantage of Haskell’s non-strict nature is that data constructors are also non-strict. This allows the definition of infinite data structures. Despite the structure being infinite in size (taking an infinite amount of time and space to calculate in its entirety), the needed parts can be computed in finite time.

Listing 2.3 illustrates an example of an infinite data structure holding the Fibonacci sequence.

**Typeclasses**
Typeclasses are Haskell’s interpretation of function overloading. A typeclass defines a “set of functions that can have different implementations depending on the type of data they are given” (O’Sullivan et al. 2008). Listing 2.4 shows a typeclass, Tree, that defines
Background and Review

\[ fib = 1 : 1 : \text{sumTwo } fib \]
\[ \text{sumTwo } (x:y:zs) = x + y : \text{sumTwo } (y:zs) \]

Listing 2.3: Infinite fibonacci sequence

```
data BinaryTree = Branch BinaryTree BinaryTree
| Leaf Int

data TernaryTree = Branch TernaryTree TernaryTree TernaryTree
| Leaf Int

class Tree a where
  leaves :: a \to [Int]

instance Tree BinaryTree where
  leaves (Leaf x) = [x]
  leaves (Branch x y) = leaves x ++ leaves y

instance Tree TernaryTree where
  leaves (Leaf x) = [x]
  leaves (Branch x y z) = leaves x ++ leaves y ++ leaves z

breadth :: Tree a \to a \to Int
breadth t = length (leaves t)
```

Listing 2.4: Example of overloading leaves using a typeclass

A function leaves that returns the list of integers for a Tree instance. Instances of Tree are provided for BinaryTree and TernaryTree. Another function, breadth is implemented using existing definitions of leaves. Notice, from its type signature, breadth accepts any instance of Tree as an input.

Monads Monads encapsulate computation in a data type. “Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations” (Newburn). Formally, a monad consists of a type constructor, a function to chain one monad onto a function producing the next (a binding operation) and a function to inject a normal value into a monad chain (a unit function).

In Haskell these notions are implemented through the Monad typeclass (Listing 2.5). An example is given in Listing 2.6 of the Maybe monad, which embodies partial functions. When the output of one function in a chain returns Nothing, this result is propagated through to the output. Otherwise, the value returned by a previous function will be used as the input to the next. The Maybe monad will be used in Section 4.3 to handle transformation applicability. The State monad will also be employed to provide a supply of fresh variable names.
2.2 Program Transformation

A program transformation is “the act of changing one program into another” (Visser et al., 2004). More specifically, there is a specific, formal, mapping of the program constructs in one form into another.

Transformation has long been used to increase the efficiency of programs during the compilation phase. The popular GNU Compiler Collection (Free Software Foundation, 2009) includes loop optimisation, common subexpression elimination and dead code elimination transformations.

The Glasgow Haskell Compiler (Haskell.org, 2008) includes inlining, specialisation and deforestation transformation strategies. GHC also uses normalisation transformations to ‘desugar’ high-level language constructs into a lower level ‘core’ Haskell syntax.

In addition to optimisation, transformations have also been used for: turning a programming written in one language into another (migration), moving from abstract specification to a concrete program (synthesis and refinement), changing the readability of the program code (refactoring and obfuscation) and reasoning about programs’ execution. (Visser et al., 2004)

Burstall and Darlington (1977) fold/unfold, introduced in Subsection 2.2.1 is a set of primitive program transformations. These can be applied to optimise a program or perform reasoning but there is no specific goal or strategy. A number of heuristics are suggested for optimising programs, but they are not deterministic.

In contrast, defunctionalisation (Subsection 2.2.2), deforestation (Subsection 2.2.3), fusion (Subsection 2.2.3) and supercompilation (Subsection 2.2.4) are all transformation strategies that use the primitive laws of fold/unfold to move code into specific forms.

Defunctionalisation eliminates higher order functions. Deforestation and Fusion remove intermediary data structures. Supercompilation attempts to execute the program as far as possible at compile time, without knowing the particular inputs. Through this process, it achieves some of the effects of defunctionalisation, deforestation and fusion.
Background and Review

2.2.1 The Fold/Unfold Laws

Burzall and Darlington (1977) introduced a series of inference rules under which recursive programs can be reasoned about and optimised. These laws have become collectively known as the ‘fold/unfold rules’. They are:

1. Definition — The introduction of a new equation.
2. Instantiation — The introduction of instances of an existing equation.
3. Unfolding — Replacing an instance of a ‘left hand side’ with its corresponding ‘right hand side’ definition.
4. Folding — Replacing an instance of a ‘right hand side’ with its corresponding ‘left hand side’ definition.
5. Abstraction — The extraction of expressions to introduce new local definitions.
6. Laws — Appealing to any known theorems about the functions, such as associativity, commutativity and distributively.

These laws were applied by Firth (1990) to a non-strict, polymorphically typed, pattern matching functional language, Glide. Firth highlighted and solved several issues that arose from using fold/unfold in this context.

For example, the non-strict semantics of Glide raises issues with the fold/unfold laws. The instantiation law replaces lazy variables with strict patterns. Therefore, regardless of the strictness of a function definition, an instantiated form (using Burzall and Darlington (1977) will be). Additional work is required to preserve the strictness of a function. Firth (1990) also introduces techniques for preserving the types and typability of functions and preserving the definedness of functions.

The application of fold/unfold laws is illustrated in Listing 2.7. In this example, two Haskell functions are composed into one. Burzall and Darlington (1977) do suggest a number of heuristics to select which rule to apply when optimising programs. However, no specific optimised strategy is presented for fold/unfold.

2.2.2 Defunctionalisation

Higher order functions were discussed in Section 2.1. Mitchell (2009) states that “having functions as first-class values leads to more concise code, but it often complicate analysis methods.” A technique that transforms programs written in a higher order style into first order would have obvious advantages.

Reynolds (1972) eliminated higher order functions to simplify the interpretation of functional code. His technique involved encoding functions as data which are then mapped onto the correct function definition at application. Listing 2.9 shows how Reynolds’ technique would transform the higher order function heads in Listing 2.8.

However, Mitchell (2009) highlights that while Reynolds’ technique is a reasonably simple and complete transformation, it actually makes the code more complex by adding indirection. Mitchell introduces another technique for defunctionalisation using simplification, inlining and specialisation program transformations. The effect of this style of defunctionalisation is shown in Listing 2.10.
2.2 Program Transformation

—— Original definitions

\[ \text{sum } [ ] = 0 \] —— (1)

\[ \text{sum } (x:xs) = x + \text{sum } xs \] —— (2)

\[ \text{squares } [ ] = [ ] \] —— (3)

\[ \text{squares } (x:xs) = x \times x : \text{squares } xs \] —— (4)

\[ \text{sumSquares } xs = \text{sum } (\text{squares } xs) \] —— (5)

—— Instantiate (5) with \( xs = [ ] \) and \( xs = (x:xs) \)

\[ \text{sumSquares } [ ] = \text{sum } (\text{squares } [ ]) \] —— (6)

\[ \text{sumSquares } (x:xs) = \text{sum } (\text{squares } (x:xs)) \] —— (7)

—— Unfold squares in (6) and (7)

\[ \text{sumSquares } [ ] = \text{sum } [ ] \] —— (8)

\[ \text{sumSquares } (x:xs) = \text{sum } (x \times x : \text{squares } xs) \] —— (9)

—— Unfold sum in (9)

\[ \text{sumSquares } (x:xs) = x \times x + \text{sum } (\text{squares } xs) \] —— (10)

—— Unfold sum in (8) / Fold back into sumSquares in (10) using (5).

\[ \text{sumSquares } [ ] = 0 \] —— (11)

\[ \text{sumSquares } (x:xs) = x \times x + \text{sumSquares } xs \] —— (12)

Listing 2.7: Using fold/unfold to compose two functions

map f [ ] = [ ]
map f (x:xs) = f x : map f xs
heads xs = map head xs

Listing 2.8: Another example of a higher order function [Mitchell, 2007]

data Func = Head
apply Head x = head x
map f [ ] = [ ]
map f (x:xs) = apply f x : map f xs
heads xs = map Head xs

Listing 2.9: Reynolds’ style defunctionalisation [Mitchell, 2007]

map_head [ ] = [ ]
map_head (x:xs) = head x : map_head xs
heads xs = map_head xs

Listing 2.10: Mitchell’s style defunctionalisation [Mitchell, 2007]


2.2.3 Deforestation and Fusion

Deforestation is a technique used to remove intermediate data structures, those that are constructed by one function only to be traversed by the next. The example used for composition in Listing 2.7 is actually a form of deforestation. The original definition of sumSquares would construct a list of the squares only to then take it apart again to sum the elements. The derived definition does not construct this intermediate list.

Originally proposed by Wadler (1990), the deforestation transformation strategy specifically removes intermediate trees by converting composed functions into a ‘treeless’ form. The difference between the non-treeless and treeless forms is demonstrated in Listing 2.11.

Wadler distinguishes between pure deforestation, under which no intermediate values are permitted, and blazed deforestation which allows some atomic intermediate values to remain. Under the blazed deforestation scheme, non-tree typed expressions are marked and prevented from composition.

Deforestation was refined into a technique called fusion by Chin (1992). He differentiates between the producer function that returns the data structure as output and the consumer function, which accepts the data structure as an input. In the expression c(p(x)), p is the producer and c is the consumer.

Chin (1992) states that a composition can be safely fused “if the transformation sequence which follows does not going into a loop and there is no loss of efficiency”. A composition can be fused effectively if the intermediate data structure is removed resulting in a performance improvement. Safe fusion is essential while effective fusion is merely desirable.

Similar to blazing, sub-expressions are tagged as safe or unsafe producers and consumers. Safe fusion can be ensured by only fusing if both the producer and the consumer are safe. Otherwise, the algorithm will fail.

2.2.4 Supercompilation

Supervised compilation, or supercompilation, (Turchin 1979) attempts to execute the program at compile time to produce a more efficient form. As the inputs are unknown at
2.3 Program Transformation Tools

Supercompilation achieves many of the effects of Wadler (1990) deforestation and Chin (1992) fusion. Listing 2.12 illustrates the deforestation/fusing abilities of supercompilation. Further examples will be presented as we investigate the application of this algorithm to our core language in Section 3.5.

The algorithm begins by driving through a function definition, inlining applications and simplifying the results. When it appears that the process is revisiting previously derived results (a condition known as ‘the whistle’), the children of the outermost expression are generalised where possible. The generalised forms are replaced by applications of existing equivalent functions or new functions, called residuals. The residuals are themselves driven through. The process continues until no new residuals are generated.

Sørensen et al. (2008) restricts the supercompiler to only propagate positive information and ignore negative information. Take the example of an equality test in a conditional: only the true branch has variables substituted as it is easier to pass the positive information, the known value of the variable, than it is to communicate negative information, the value that the variable is not.

The positive supercompiler is investigated again by Jonsson and Nordlander (2009) in the setting of a call-by-value language. They deal with the strictness issues discussed in Section 2.1 by performing strictness analysis on functions before inlining. Functions that have possibly non-strict arguments are prevented from being inlined.

Mitchell (2008, chapter 4) investigates the application of supercompilation to a core subset of Haskell. He introduces a new generalisation technique that appears to produce better performing residual programs than if the generalisation of Sørensen et al. (2008) is applied.

Mitchell shows optimisations across the ‘imaginary’ section of the NoFib benchmark suite (Partain, 1992). He presents the startling instance of a supercompiled Haskell program performing faster than its C equivalent. Mitchell’s work will be discussed further in Chapter 3, Problem Analysis.

2.3 Program Transformation Tools

Visser et al. (2004) state that “one of the aims of a general framework for program transformation is to define transformations that are reusable across as wide a range of languages as possible”.

Transformation metalanguages. One technique for specifying transformations is the use of a formal metalanguage to describe transformations and the languages they operate upon. Using these preliminary definitions, a transformation tool can parse a program to build a abstract syntax tree, perform transformation operations on this tree and output the newly derived program. These metalanguages are generally wide-spectrum to support the description of a multitude of programming language paradigms.

The TXL project (Cordy et al., 1988) is an example of such a tool. An arbitrary context free-grammar is provided for the language to be transformed and transformation rules are supplied in a functional form. A variety of language descriptions in the TXL grammar format are freely available for common (generally imperative) languages allowing the speedy implementation of new transformations.
map f xs = \texttt{case xs of}
\begin{align*}
[&] &\rightarrow &[&] \\
(y:ys) &\rightarrow f\ y : \text{map}\ f\ ys
\end{align*}

main xs = \text{map}\ ((1\ +)\ (\text{map}\ (2\ *)\ xs))
===> \{\text{Residuate}\}

main xs = ho\ xs

ho\ xs = \text{map}\ (1\ +)\ (\text{map}\ (1\ *)\ xs)
===> \{\text{Inline} \ 'map' \}

ho\ xs = \texttt{case}\ (\text{map}\ (2\ *)\ xs)\ of
\begin{align*}
[&] &\rightarrow &[&] \\
(y1:ys1) &\rightarrow (1\ +)\ y1 : \text{map}\ (1\ +)\ ys1
\end{align*}
===> \{\text{Inline} \ 'map' \}

ho\ xs = \texttt{case}\ (\texttt{case}\ xs\ of
\begin{align*}
[&] &\rightarrow &[&] \\
(y2:ys2) &\rightarrow (2\ *)\ y2 : \text{map}\ (2\ *)\ ys2)\ of
[&] &\rightarrow &[&] \\
(y1:ys1) &\rightarrow (1\ +)\ y1 : \text{map}\ (1\ +)\ ys1
\end{align*}
===> \{\text{Case of case}\}

ho\ xs = \texttt{case}\ xs\ of
\begin{align*}
[&] &\rightarrow &[&] \\
(y2:ys2) &\rightarrow \texttt{let}\ y1 = (2\ *)\ y2 \\
& &\quad ys1 = \text{map}\ (2\ *)\ ys2 \\
& &\quad \texttt{in}\ (1\ +)\ y1 : \text{map}\ (1\ +)\ ys1
\end{align*}
===> \{\text{Case of constructor, twice}\}

ho\ xs = \texttt{case}\ xs\ of
\begin{align*}
[&] &\rightarrow &[&] \\
(y2:ys2) &\rightarrow (1\ +\ (2\ +\ y2)) : \text{map}\ (1\ +)\ (\text{map}\ (2\ *)\ ys2)
\end{align*}
===> \{\text{Homeomorphic embedding termination, fold}\}

ho\ xs = \texttt{case}\ xs\ of
\begin{align*}
[&] &\rightarrow &[&] \\
(y2:ys2) &\rightarrow (1\ +\ (2\ +\ y2)) : \text{ho}\ ys2
\end{align*}

Listing 2.12: Supercompiling to perform fusion
2.3 Program Transformation Tools

Cordy (2006) highlights that transformation systems need to provide a method for controlling the direction of tree traversal. Some transformations, such as inlining a function body at an application, can lead to non-termination if executed in a top-down manner. Others require a bottom up movement.

FermatT (Ward and Zedan, 2005), Stratego/XT (Visser, 2004), DMS (Baxter et al., 2004) and ASF+SDT (van den Brand et al., 2001) are examples of other transformation systems that uses domain specific languages. Each has their own particular implementation specifics but overall they follow the same concept of describing the language and then transformations upon it.

An issue with such systems is due to their generic nature, error messages can be quite vague. “One problem with the general approach of language implementation by preprocessor is the difficulty in providing error diagnostics that are related to the original source when the base programming language processor detects errors in the transformed result.” (Cordy et al., 1988) This makes debugging parsing and transformation errors unnecessarily complicated.

Another approach is the Template Haskell (Sheard and Jones, 2002) meta-programming system. Haskell acts as both the language being operated upon and the language performing the operations. One gains the benefits of using a transformation system that is embedded in an existing, robust compiler with familiar behaviour and error responses. But one loses the ability to operate on other languages.

Pattern matching and boilerplate removal Another approach is to implement a transformation system in a host general purpose programming language. Parsing libraries for reading in programs in arbitrary ‘target’ languages already exist for most popular programming languages. Similarly, pretty printing libraries exist for outputting the results.

For the actual transformation, it is necessary to have a system that can traverse the abstract syntax tree and match patterns within the tree for replacement. This is where a host language that supports tree data type pattern matching has clear advantages. As we have already discussed, such pattern matching features are fully supported Haskell 98.

Custom functions could be written for performing tree traversal. This ‘boilerplate’ code would need to be rewritten for every variation of abstract syntax tree representation.

Several techniques are available to alleviate the burden of having to write the often tedious and repetitive definitions. ‘Scrap Your Boilerplate’ (Lämmel and Peyton Jones, 2003) introduces generic traversals where common traversal functions can be applied to any tree data type. This technique requires extensions to Haskell 98, such as rank-2 types, and utilises derivable type classes to relieve the user from having to write any additional code to use the generic operations.

Uniplate (Mitchell and Runciman, 2007; Mitchell, 2008, chap. 3) is another take on boilerplate removal. It abstracts all traversal operations to functions that utilise a single overloaded function uniplate, that is implemented as part of the Uniplate typeclass. Once an instance of the Uniplate class, any homogenous data type can be traversed using standard functions. Uniplate will be discussed further in Section 4.2 and compared with the solution for tree traversal in Optimus Prime.
Interactive Transformation  Major projects, such as CIP (Bauer 1979) and KIDS (Smith 1991), promoted the use of interactive program transformation systems to derive programs from formal specifications. The advantage of such a process would be that conformity to a specification can be ensured due to formal semantic equivalence.

Similar to the transformation metalanguages mentioned above, the Computer-aided, Intuition-guided Programming project (Bauer 1979) uses a wide-spectrum language, CIP-L, to define target languages and a related language, CIP-S, to define transformations upon it. While the transformation specifications include ‘applicability conditions’ the system does not automatically apply the transformations. Instead a human guides the process by entering transformation commands through a command-line interface. Despite the human direction, CIP ensures that transformations are performed correctly.

CIP was designed to be as generic and flexible as possible to cover all transformation needs in any given language. Feather (1982) introduced ZAP, an interactive system for improving program performance by transformation. ZAP was designed for optimising existing programs. Hierarchies of transformations are developed, from complex transformation strategies down to transformation primitives. A control language is used to apply these transformations and can be entered interactively at the command-line or batch scripted in advance and applied all in one go.

STARSHIP Firth (1990) is an interactive transformation for operating on the Glide non-strict, functional language. It is built upon the Burstall and Darlington (1977) fold/unfold laws with modifications for the new lazy, functional setting (see Subsection 2.2.1). Like ZAP, a control language is entered at a command like to control transformations. However, as STARSHIP is considered a ‘proof assistant’, advanced functionality is supported for “exploratory transformation.” One such feature is backtracking through previously introduced ‘laws’.

Programmer Assistant for Transforming Haskell (PATH) (Tullsen 2002) is another “user-directed program transformation system” for use with a non-strict, functional language. As the name suggests, it is specifically targeted at the Haskell language. Like STARSHIP, it builds upon the principles of Fold/Unfold transformation while dealing with some of the issues caused by Haskell’s advanced features.

The system is embedded in the Emacs text editor, providing a dialog-based interface. This is a break from the command-line interfaces we have seen previously. Tullsen claims that this is to improve its appeal to practitioners. “The goal is for PATH to be usable by any novice functional programmer.” (Tullsen 2002) This should be a goal for any interactive program transformation tool. An intuitive interface is essential.

2.4 The Reduceron and F-Lite

The Reduceron 2 (Naylor and Runciman 2008) is a graph reduction machine for executing a functional language, f-lite. It is implemented on a field programmable gate array (FPGA) and features parallel wide memories for program code, heap and stack.

Most mainstream computer processors are engineered to execute imperative programs. Functional programs are transformed by complex compilers from their functional forms to execute efficiently on these platforms. The Reduceron is being developed to research the advantages of utilising a special-purpose processor to execute functional code, rather than the other way around.
2.4 The Reduceron and F-Lite

---

Abstract syntax tree root. List of function declarations.

type Prog = [Decl]

---

A function declaration, including a name, arguments and expression.

data Decl = Func { funcName :: Id,
               , funcArgs :: [Pat]
               , funcRhs :: Exp }

---

A variable, function or constructor identifier

type Id = String

---

f-lite expressions

data Exp = App Exp [Exp]
| Case Exp [Alt]
| Let [Bind] Exp
| Var Id
| Con Id
| Fun Id
| Int Int
| Bottom

---

Patterns for case alternatives and arguments.

type Pat = Exp

---

Case alternative.

type Alt = (Pat, Exp)

---

Let binding

type Bind = (Id, Exp)

---

Listing 2.13: AST for the f-lite syntax

The f-lite language is a subset of Haskell 98 (Simon Peyton Jones et al., 2003) consisting of function definitions, case expressions, limited let expressions, function applications and constructor applications expressed in the explicit ‘braces’ layout-insensitive format. Notably, it excludes comments, type definitions, type checking and explicit lambda expressions.

The language is complete. Any Haskell constructs that are not included in the core language can be desugared into an f-lite form. The reduced language merely simplifies the development of the Reduceron. Many examples of f-lite source listings are provided throughout this project and the language is discussed in detail with reference to our needs in Section 3.2.

Naylor and Runciman have released the code for an early iteration of the Reduceron compiler, written in Haskell. It includes an f-lite parser, a few transformation-based optimisations, post-transformation pretty-printed output and a bytecode generator. The Reduceron compiler represents a program abstract syntax tree using the data structures outlined in Listing 2.13.

The parser is implemented using applicative parser combinators, similar to those described in O’Sullivan et al. (2008). This style benefits from a compact but readable
2 Background and Review

append Nil ys = let zs = ys in zs;
append (Cons x xs) ys = cons x (append xs ys);
cons x xs = Cons x xs;

— after equation pattern desugar and let inlining becomes —

append v1 ys = case v1 of {
    Nil → ys;
    Cons x xs → cons x (append xs ys);
};
cons x xs = Cons x xs;

Listing 2.14: Examples of Naylor’s transformations

representation. The compiler outputs the abstract syntax tree after the transformation phase for analysis and debugging. Our reuse of this parser is discussed in Section 4.1.

The output is produced through implementations of the ‘Show’ typeclass for the abstract syntax tree data structures. However, this approach was considered to cumbersome for our projects needs and an alternative will be discussed in Subsection 4.5.1.

A number of transformations are already implemented in the Reduceron compiler including; de-sugaring equation patterns to cases, de-sugaring multiple constructor patterns to nested cases and inlining some let bindings. The effects of these transformations are illustrated by Listing 2.14.

2.5 Summary

In this chapter, we have introduced the language that we shall transform, f-lite, and the language in which we shall implement our tool, Haskell. We have looked at relevant features of these languages in preparation for the Problem Analysis and Design and Implementation chapters which follow.

Program transformations and specific applications of program transformations have been discussed. Program transformation strategies range from the low-level Burstall and Darlington (1977) fold/unfold laws and to the powerful Turchin (1979) supercompilation strategy. We demonstrate how supercompilation can produce many of the desirable effects of other optimisation strategies.

We have classified existing program transformation tools into three groups, (i) transformation metalanguages which use domain specific programming languages to describe target languages and transformations, (ii) pattern matching which exploits native features in general purpose programming languages to perform transformations and (iii) interactive transformation which use human intuition to guide the transformation process.

Every transformation tool investigated has highlighted the need to be able to control the direction of traversal for transformation application. Another important point to take away from the literature on interactive transformation is that a tool should “be usable by any novice functional programmer.” (Tullsen, 2002)
3 Problem Analysis

This chapter considers the issues and prerequisites for developing an interactive transformation system and implementing a selection of transformations within such a system. Section 3.1 presents a formalisation for specifying transformations to be included in our system. The core language that our system will operate on is defined in Section 3.2 and the transformation primitives for that language are discussed in Section 3.3.

We shall investigate how the concepts of Burstall and Darlington (1977) fold/unfold transformation (Section 3.4) and Turchin (1979) supercompilation (Section 3.5) can be applied to our language.

Finally, in Section 3.6 the requirements for an interactive transformation system are derived from existing literature and our previous analysis.

3.1 Specification of Transformations

A program transformation can be viewed as a manipulation of the abstract syntax tree. Figure 3.1 illustrates an example of tree transformation. Circular objects represent known nodes. Single bordered triangular objects are placeholders for subtrees of any form. Double bordered triangular objects are placeholders for sequences of subtrees of any length.

A more compact notation, for the same transformation, is shown in Figure 3.2. Nodes are written in bold while transformation variables (placeholders) as written as \( x \) or \( \mathcal{X} \). Table 3.1 summarises the naming convention for transformation variables.

3.2 The Core Language

As this project is orientated towards the development of compiler transformations for the Reduceron, our core language will be a normalised form of Naylor’s f-lite language. A normalised, desugared format is used to aid the description of transformations. The syntax is illustrated in Figure 3.3.

As one of Naylor’s (Naylor and Runciman, 2008) own compiler transformations converts pattern matching for function applications into case expressions, patterns can be omitted from function definitions. Instead, a simpler flat variable argument list is used. This allows simpler transformation operations, as will be demonstrated later.

Similarly Naylor’s compiler desugars complex case patterns, containing more than one constructor or wild-card matches, into nested case expressions, each dealing with a single constructor component. Therefore, our core language can omit these ‘complex’ patterns, only permitting the simple type.

Comments are permitted at the declaration level, to allow syntax trees to be annotated with information about the transformations that have taken place.
3 Problem Analysis

Figure 3.1: An example of a tree transformation

\[ A (B E)(C x)(D \overline{ys}) \rightarrow E x (D \overline{ys}) \]

Figure 3.2: Compact notation for the transformation of Figure 3.1

Table 3.1: Transformation variable conventions

<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
<th>f-lite Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y, z )</td>
<td>Expression</td>
<td>‘let { x = inc 2 } in D x x’</td>
</tr>
<tr>
<td>( v )</td>
<td>Variable</td>
<td>‘v’, ‘var1’, ‘apples’</td>
</tr>
<tr>
<td>( v^l, w^l )</td>
<td>Local Variable</td>
<td>( v^l \subseteq v )</td>
</tr>
<tr>
<td>( v^f )</td>
<td>Fresh Variable</td>
<td>( v^f \subseteq v )</td>
</tr>
<tr>
<td>( c )</td>
<td>Constructor</td>
<td>‘Nil’, ‘Cons’, ‘Record2’</td>
</tr>
<tr>
<td>( f )</td>
<td>Function</td>
<td>‘f’, ‘(+), ‘incBy1’</td>
</tr>
<tr>
<td>( f^p )</td>
<td>Primitive</td>
<td>‘(+), ‘(==), ‘emit’</td>
</tr>
<tr>
<td>( f^F )</td>
<td>Fresh Function</td>
<td>( f^F \subseteq f )</td>
</tr>
<tr>
<td>( m, n )</td>
<td>Integer</td>
<td>‘0’, ‘5’, ‘111’</td>
</tr>
<tr>
<td>( p )</td>
<td>Pattern</td>
<td>‘v’, ‘Nil’, ‘(Cons x xs)’</td>
</tr>
<tr>
<td>( b )</td>
<td>Binding</td>
<td>‘x = 10’</td>
</tr>
<tr>
<td>( a )</td>
<td>Alternative</td>
<td>‘Nil → 10’</td>
</tr>
<tr>
<td>( d )</td>
<td>Declaration</td>
<td>‘inc x = (+) x 1’, ‘{-blah-}’</td>
</tr>
<tr>
<td>( \overline{s} )</td>
<td>Sequence</td>
<td>( \overline{s} ) is a sequence of variables</td>
</tr>
</tbody>
</table>
3.2 The Core Language

The core language includes seven primitive functions. Addition (+) and subtraction (−) operate on integers and return integers. Equality (==), inequality (/=) and less than or equal (<=) operate on integers and return one of the zero-arity constructors True and False. The functions emitInt and emit output their argument and display the value of that argument as a benign side-effect. Any application of a primitive function must be saturated.

This core language differs from that used by Mitchell (2008): (i) it does not include anonymous lambda abstractions, (ii) it has primitive functions (iii) and it allows multiple bindings within a let expressions. These differences must be taken into consideration when define simplification rules for the f-lite language, as we will discuss in Section 3.5.

Where more than one global declaration has been made for the same function name, only the first definition is considered ‘active’. This convention allows historical information about derivations to be stored without having to augment the abstract syntax tree structure.

The semantics of the f-lite language allow for variable names to be reused in different scopes. Therefore, any operation using variable names to identify specific variables must ensure that the variable name at this point refers to the intended variable. One possible fault that may occur is known as variable capture. An example is shown in Figure 3.6 and the concept explained in Subsection 3.3.1.

### 3.2.1 Preliminary Definitions

**Bound variables** The bound variables of an expression x are all those that are bound to a let expression binding or a case expression alternative pattern.
Problem Analysis

\[
\text{children}(x_1 x_2 ... x_n) \\
= \{x_1, x_2, ..., x_n\}
\]

\[
\text{children( let } \{ v_1 = x_1 ; ... ; v_n = x_n \} \text{ in } y) \\
= \{x_1, x_2, ..., x_n, y\}
\]

\[
\text{children( case } x \text{ of } \{ p_1 \rightarrow y_1 ; ... ; p_n \rightarrow y_n \}) \\
= \{x, y_1, y_2, ..., y_n\}
\]

\[
\text{children( } \_ \text{ )} = \{\}
\]

Figure 3.4: The children of f-lite expressions

Free variables Conversely, the free variables of an expression \( x \) are all those within \( x \) that are not bound to a let expression binding or a case expression alternative pattern. The function \( \text{free}(x) \) returns the free variables of expression \( x \).

Substitution A substitution of a variable for an expression is denoted by \( x [v / y] \). All free instances of \( v \) in \( x \) are replaced with expression \( y \).

Operations on sequences Collections of expressions can have a number of operations applied to them. The length of a sequence, \( \text{length}(\mathbb{E}) \), is the number of items contained. \( \text{append}(\mathbb{E}_1, \mathbb{E}_2) \) appends two sequences of expressions. The zip operation, \( \text{zip}(\{v_1, v_2, ..., v_m\}, \{x_1, x_2, ..., x_n\}) \) tuples together each element such that it returns \( \{v_1, x_1, v_2, x_2, ..., v_m, x_m\} \) when \( m \leq n \). The rest operation returns the elements from the \( \mathbb{E} \) sequence, not used in the zip operation. That is to say \( \text{rest}(\mathbb{E}, \mathbb{E}_1) = \{x_{m+1}, x_{m+2}, ..., x_n\} \).

Relationships between expressions The \( \text{children}(x) \) are all the expressions that it contains, as defined in [Figure 3.4]. Conversely, the \( \text{parent}(x) \) is such that \( \forall x : \text{children}(p) \cdot \text{parent}(x) = p \). The \( \text{descendants}(x) = \text{children}(x) \cup \text{descendants}(\text{children}(x)) \).

3.3 Transformation Primitives for f-lite

3.3.1 Inline / Unfold

An inline, or unfold, replaces function application with the corresponding instance of the function body. A partial application occurs where fewer arguments are supplied than the arity of the function. Inlining partial applications is not possible due to the lack of lambda abstractions in our core language. Full application occurs when the number of arguments and the arity are equal and over application occurs where the number of arguments exceeds the arity.

If inlining simply replaces variables by argument expression, sharing may be lost where a variable is used more than once. To preserve sharing, a let expression is used to
3.4 The Fold/Unfold Laws

The fold/unfold laws of Burstall and Darlington (1977) are useful for both the optimisation of code and the reasoning about programs. Originally conceived to operate on a first order recursion equation language, they can be extended to operate on a non-strict high
Figure 3.7: Definition of fold, where \( f \ vs = y \) and \( y \ [zip \ vs \ ys] \equiv_a x \)

\[
x \mapsto f \ vs
\]

and a new function definition is added for \( f^F \ free(x) = x \)

Figure 3.8: Definition of residuate, where \( f^F \) is a fresh function name

order functional language. This section will also consider how to express fold/unfold concepts in our core language.

Listing A.1 in Appendix A illustrates the application of number of the laws to produce an optimised f-lite definition of a Fibonacci series generator. This is similar to the example used by Burstall and Darlington (1977). A fold/unfold proof of the theorem ‘if an element occurs in the concatenation of two lists, if must exist in one of the lists independently’, is shown in Listing A.2 in Appendix A.

**Definition** The Definition rule (also known as Eureka) is used to introduce new function definitions into the derivation. One should be able to introduce new definitions at any time, not just the initial loading of the program. Lines 1 to 5 of Listing A.1 and 1 to 17 of Listing A.2 demonstrate the use of the definition law.

**Instantiation** Instantiation replaces a variable in a function definition by specific a constructor patterns. In Haskell98, instantiation can be expressed in the arguments of function definitions through pattern matching. As our normalised form suppresses this and our syntax does not support the construct, instantiation will instead be expressed in case alternatives.

In f-lite syntax, instantiation of an argument \( v_i \) with a constructor patterns \( c_0 \ vs_0, \ldots, c_m \ vs_m \) in function \( f \ vs = x \) can be done by inserting a **case** expression at the root of the definition:

\[
\text{case } v_i \text{ of } \{ \\
\quad c_0 \ vs_0 \to x \ [v_i / c_0 \ vs_0 ] ; \\
\quad \ldots ; \\
\quad c_m \ vs_m \to x \ [v_i / c_m \ vs_m ] \\
\}
\]

Then immediately perform the case of constructor transformation (**caseOfCons** in Figure 3.12) to propagate the pattern matching. Lines 6 to 13 of Listing A.1 illustrate the use of the instantiation law.
### 3.4 The Fold/Unfold Laws

\[
\text{power}4 \ x = \text{times} \left( \text{times} \ x \ x \right) \left( \text{times} \ x \ x \right);
\]
\[
\leftarrow \text{Abstract} \ '\text{times} \ x \ x' \ .
\]
\[
\text{power}4 \ x = \text{let} \ \{ \text{sqr} = \text{times} \ x \ x \} \ \text{in} \ \text{times} \ \text{sqr} \ \text{sqr};
\]
\[
\text{inc\_pair} \ x \ y = \text{Pair} \ ((+) \ 1 \ x) \ ((+) \ 1 \ y)
\]
\[
\leftarrow \text{Abstract} \ '(+) \ 1' \ .
\]
\[
\text{inc\_pair} \ x \ y = \text{let} \ \{ \text{inc} = (+) \ 1 \} \ \text{in} \ \text{Pair} \ (\text{inc} \ x) \ (\text{inc} \ y)
\]

Listing 3.1: Examples of abstraction in f-lite

Residuating the expressions in the alternatives can improve the readability of a derivation. \[\text{Listing A.2}\]\ illustrates this style.

In a non-strict language, \(\perp\) arguments should be considered, in addition to the range of constructed values. Failure to do so can lead to a derived function exhibiting different behaviour.

Reduction of expressions containing \(\perp\) proceeds as follows; case expressions with strict patterns and \(\perp\) as the subject simplify to \(\perp\) and applications of primitive functions with \(\perp\) as an argument simplify to \(\perp\). Instantiation with \(\perp\) is used in \(\text{Listing A.2}\), lines 19 to 29.

#### Unfolding

Unfolding (or Inlining) has already been defined in \[\text{Subsection 3.3.1}\]. We ensure that sharing is preserved by providing the arguments to the inlined expression through local variables, rather than directly substituting the expressions.

#### Folding

Folding has already been defined in \[\text{Subsection 3.3.2}\]. There are no practical differences between that definition and the one used in \[\text{Burstall and Darlington}\,(1977)\].

#### Abstraction

Abstraction introduces a local definition for similar subexpressions to promote sharing. In Haskell 98, local definitions can be functions of arbitrary arity. Subexpressions can be generalised to factor out similar computation and reduce redundancy.

All f-lite bindings must be unary variables. These variables may contain partial applications of functions but no new functions can be defined. This restriction means that only equal expressions can be abstracted, not similar computations. This type of transformation is known as \textit{common subexpression elimination}.

However, this restriction does make it much easier to determine when an abstraction should occur. \[\text{Listing 3.1}\] illustrates the application of these laws with examples.

#### Laws

One expression can be replaced by another known to be equivalent. Equivalents can be inferred from known theorems, such as associativity and commutativity, or may be determined by previous derivations.
3 Problem Analysis

\[ f \, \overline{v} = v \quad f \, \overline{v} = \text{case } v \text{ of } \overline{a} \overline{l}\overline{t}s \]
\[ f \, \overline{v} = c \, \overline{x} \overline{s} \quad f \, \overline{v} = \text{case } c \, \overline{x} \overline{s} \text{ of } \overline{a} \overline{l}\overline{t}s \]
\[ f \, \overline{v} = f^{p} \, \overline{x} \overline{s} \quad f \, \overline{v} = \text{case } f^{p} \, \overline{x} \overline{s} \text{ of } \overline{a} \overline{l}\overline{t}s \]

Figure 3.9: Root expressions that will cause simple termination

3.5 Supercompilation Strategy

Mitchell (2008, chap. 4) showed that supercompilation (Turchin, 1979) can lead to large performance boosts in a non-strict functional language. On a particular example, the word counting program, performance even bested the equivalent program written in C.

Supercompilation produces a chain of derivations by inlining function applications and simplifying the result. This process is known as driving. If this new function definition embeds a previously derived result, then driving terminates and the children of the outermost expression in the function body are residuated. The residuals are then supercompiled.

In a little more detail, the supercompilation strategy is as follows. For some function:

1. Check whether this function meets the termination criteria see Subsection 3.5.1
2. If it does not meet the termination criteria then drive;
   a) Inline a function application (see Subsection 3.5.2) and
   b) Simplify the function (Subsection 3.5.3).
   c) Continue supercompiling this new definition.
3. If it meets the termination criteria then; (Subsection 3.5.4)
   a) If the driving terminated because of an embedding, generalise the expression with respect to the function body it embeds.
   b) Tie expression. If any of the expression’s children are instances of a previously derived residual body, then fold back into this definition. Otherwise, residuate the child and supercompile the residual.
4. When every residual has been supercompiled, inline (Subsection 3.5.5) non-recursive residual definitions to alleviate unnecessary function application overheads.

3.5.1 Termination Criteria

Two types of termination criterion are used for Mitchell’s (2008) supercompiler. The simple termination criterion stops driving through a function if supercompiling the root of the function definition can not pass any more information to the others. The definition of the simple termination criterion for f-lite is shown in Figure 3.9.

A classic example of this is where the root of the definition is a case expression with an unbound variable as the subject. As this case expression cannot be resolved, one may as well supercompile its children expressions separately (see Subsection 3.5.4).
3.5 Supercompilation Strategy

\[
\begin{align*}
\text{dive}(x, y) \lor \text{couple}(x, y) & \quad \exists c : \text{children}(y) \cdot \text{homeo}(x, c) \\
\text{homeo}(x, y) & \quad \text{dive}(x, y)
\end{align*}
\]

\[
x \sim y \land \forall (x_c, y_c) : \text{zip(children}(x), \text{children}(y)) \cdot \text{homeo}(x_c, y_c)
\]

\[
\text{couple}(x, y)
\]

Figure 3.10: The homeomorphic embedding relation, \text{homeo} or \sqsubset

\[
\begin{align*}
f & \sim f & \text{same function name} \\
c & \sim c & \text{same constructor name} \\
v & \sim v & \text{same variable name} \\
v^L & \sim w^L & \text{both are local} \\
x \, \text{l} \sim y \, \text{l} & \quad \text{length} \, \text{l} = \text{length} \, \text{l} \\
\text{let } b_s \text{ in } x & \sim \text{let } b_s \text{ in } x & \text{both bind the same variables} \\
\text{case } x \text{ of } \text{alts}_x & \sim \text{case } y \text{ of } \text{alts}_y & \text{both contain the same patterns}
\end{align*}
\]

Figure 3.11: The expression similarity relation, \sim

Our definition of the simple termination criterion differs slightly from Mitchell (2008, chap. 4). Our language includes ‘primitive’ functions that cannot be inlined as they are handled at hardware level. To the supercompilation strategy, they behave very similar to a free variable. Therefore, a derivation containing a primitive function application at its root or a \text{case} expression on a primitive application should be residuated.

The other termination criterion is if a previously derived definition is \text{homeomorphically embedded} in the current expression. We say that \( x \) is a homeomorphic embedding of \( y \), if the relation \text{homeo}(x, y) holds as described in Figure 3.10.

3.5.2 Unfolding Selection

Functions are inlined using the ‘sharing preserving’ approach described in Subsection 3.3.1. In a given derivation, functions are inlined in order of evaluation and if they will not cause driving to terminate. If all inlinings will cause driving to terminate, then select the first in the order of evaluation. Otherwise, terminate driving.

3.5.3 Simplification Transformations for f-lite

Mitchell (2008) presents thirteen simplification laws, used as part of the supercompilation strategy. He notes that while “some of the rules duplicate code, ... none duplicate work. All the rules preserve the semantics and the sharing behaviour of an expression.”

The simplification rules presented in Figure 3.12 are defined similarly to those that
3 Problem Analysis

\[
( x \text{\,\,\,} \overline{xs} \text{\,\,\,} \overline{ys}) \quad (\text{appOfApp})
\]
\[
\Rightarrow
\]
\[
x \text{\,\,\,} \text{append}( \overline{xs} \text{\,\,\,} \overline{ys})
\]

\[
\text{case} \ c \ x_1 \ldots x_n \text{\,of\,\{\ldots ; v_1 \ldots v_n \rightarrow y \ldots \}} \quad (\text{caseOfCons})
\]
\[
\Rightarrow
\]
\[
\text{let} \ \{ v_1 = x_1 \ldots ; v_n = x_n \} \text{\,in\,} y
\]

\[
(\text{\,\,\,case} \ x \text{\,of\,\{p_1 \rightarrow y_1 \ldots ; p_n \rightarrow y_n \})} \overline{zs} \quad (\text{appToCase})
\]
\[
\Rightarrow
\]
\[
\text{case} \ x \text{\,of\,\{p_1 \rightarrow y_1 \overline{zs} \ldots ; p_n \rightarrow y_n \overline{zs} \}}
\]

\[
(\text{\,\,\,let} \ \{ v_1 = x_1 \ldots ; v_n = x_n \} \text{\,in\,} y) \overline{zs} \quad (\text{appToLet})
\]
\[
\Rightarrow
\]
\[
\text{let} \ \{ v_1 = x_1 \ldots ; v_n = x_n \} \text{\,in\,} y \overline{zs}
\]

\[
\text{case} \ (\text{\,\,\,let} \ \overline{vs} \text{\,in\,} y) \text{\,of\,\overline{zs}} \quad (\text{caseOfLet})
\]
\[
\Rightarrow
\]
\[
\text{let} \ \overline{vs} \text{\,in\,}(\text{case} \ y \text{\,of\,\overline{zs}})
\]

\[
\text{case} \ (\text{\,\,\,case} \ x \text{\,of\,\{p_1 \rightarrow y_1 \ldots ; p_n \rightarrow y_n \})} \text{\,of\,\overline{zs}} \quad (\text{caseOfCase})
\]
\[
\Rightarrow
\]
\[
\text{case} \ x \text{\,of\,\{p_1 \rightarrow \text{case} \ y_1 \text{\,of\,\overline{zs}} \ldots ; p_n \rightarrow \text{case} \ y_n \text{\,of\,\overline{zs}} \}}
\]

\[
\text{case} \ v \text{\,of\,\{\ldots ; c \overline{vs} \rightarrow y \ldots \}} \quad (\text{substituteVar})
\]
\[
\Rightarrow
\]
\[
\text{case} \ v \text{\,of\,\{\ldots ; c \overline{vs} \rightarrow y \{v / c \overline{vs}\} \ldots \}}
\]

Figure 3.12: Simplification laws for f-lite

40
3.5 Supercompilation Strategy

\[ \text{let} \{ v_1 = x_1, \ldots, v_i = x_i, \ldots, v_n = x_n \} \] (letInCase)

\[ \text{in} \ (\text{case} \ y \ of \ \{ \ldots; p_j \rightarrow y_j; \ldots \}) \]

\[ \implies \]

\[ \text{let} \{ v_1 = x_1, \ldots, v_n = x_n \} \]
\[ \text{in} \ (\text{case} \ y \ of \ \{ \ldots; p_j \rightarrow \text{let} \{ v_i = v_j \} \text{in} \ y_j; \ldots \}) \]

\[ \text{where} \ v_i \not\in \text{free}(y) \cup \bigcup_j \text{free}(x_j). \]

\[ \text{let} \{ v_1 = x_1, \ldots, v_i = x_i, \ldots, v_n = x_n \} \text{in} \ y \] (inlineLet)

\[ \implies \]

\[ (\text{let} \{ v_1 = x_1, \ldots, v_n = x_n \} \text{in} \ y) \ [v_i/x_i] \]

\[ \text{where} \ v_i \not\in \bigcup_j \text{free}(x_j) \] and only occurs once in free(y) or \( x_i \) is a variable.

\[ \text{let} \{ \ldots; v_i = c \ x_{i1} \ldots x_{im}; \ldots \} \text{in} \ y \] (splitLet)

\[ \implies \]

\[ (\text{let} \{; \ldots; \} \text{in} y) [v_i/c \ v_{i1}^F \ldots v_{im}^F] \]

\[ \text{where} \ v_{i1}^F \ldots v_{im}^F \text{ are fresh variables.} \]

\[ \text{let} \{ \} \text{in} \ x \implies x \] (removeLet)

\[ \text{let} \overline{b_{s1}} \text{ in} \ (\text{let} \overline{b_{s2}} \text{ in} \ x) \] (letInLet)

\[ \implies \]

\[ \text{let} \ (\overline{b_{s1}} ++ \overline{b_{s2}}) \text{ in} \ x \]

\[ \text{where} \ \text{no variables names bound in} \ \overline{b_{s2}} \text{ occur in} \ \overline{b_{s1}}. \]

---

Figure 3.13: New and augmented simplification transformations for f-lite
Mitchell uses for his core language. However, our core language has differs from that used in Mitchell (2008), as explained in Section 3.2. For example, we allow multiple bindings within a single let expression. Laws that required major alterations are defined in Figure 3.13.

For letInCase, only bindings that are not used in the either the subject of the case expression, nor any other binding that does not meet this criteria. Any non-inlinable bindings must be retained in the inlineLet law. Furthermore, the law stating that bindings where the expression is itself is a variable can be inlined is merged in. Finally, the law splitLet needs to ensure that the variable is substituted into any referencing bindings.

To tidy up any ‘needless’ let expressions, two new rules are introduced. The remove-Let rule merely replaces a let containing no bindings with the subject expression. The letInLet rule coalesces lets in much the same was as the appOfApp rule does with applications.

The f-lite language does not include anonymous lambda abstractions. Lambda are not required for any of the simplification rules. The lam-app law converts an applied lambda into a let expression, so can be safely omitted from our collection.

These simplification laws can be non-deterministically applied throughout the abstract syntax tree until it reaches a normalised, simplified form.

3.5.4 Residuation, Folding and Generalisation

When driving terminates, it is necessary to produce residual definitions so that supercompilation can continue. The production of these residual expressions depends on a variety of conditions.

**Most specific generalisation** If compilation terminates due to a homeomorphic embedding, one may be able to generalise the current definition with respect to the similar function. We utilise the most specific generalisation techniques described by Sørensen et al. (2008).

If \( y \) is the candidate function body and \( x \) is the expression previous residual that was found to be homeomorphically embedded in \( y \), \( \text{msg}(x, y) \) is the result of applying the rewrite rule in Figure 3.14 to the triple \( (v^f, \{v^f_1 = x\}, \{v^f_2 = y\}) \). A fresh variable is represented by \( v^f \). The rewrite rule is applied until no further rewrites are possible.

\[
\begin{align*}
  t^g & \\
  \{v^f = \sigma(x_1, ..., x_n) \cup \theta_x\} & \rightarrow \\
  t^g[v^f_1, ..., v^f_n] & \\
  \{v^f = \sigma(y_1, ..., y_n) \cup \theta_y\}
\end{align*}
\]

Figure 3.14: Most specific generalisation rewrite rule, where \( v_1, ..., v_n \) are fresh variables and the \( \sigma(\overline{v^f}) \) detonates an expression spine and its children.

If most specific generalisation returns \( (x', \overline{bs_x}, \overline{bs_y}) = \text{msg}(x, y) \), the generalised expression is \( \text{let } \overline{bs_x} \text{ in } x' \). This is only valid if \( bs_x \) contains no free variables, as any
subsequently bound variables will be references before their binding.

Mitchell’s generalisation [Mitchell (2008), chap. 4] extends this method to deal with this limitation. His generalisation, denoted by $x \triangleright y$, utilises lambda expressions to lift free variables to the root level. As $\triangleright$ is able to factor out more expressions than most specific generalisation, subsequent derivations may be able to tie back sooner. Our core language does not include anonymous lambda abstractions so we cannot immediately make use of this technique.

Residuation and Folding Whether or not generalisation is feasible, the children of the root expression are residuated, under certain conditions.

If the child contains no inlinable function applications, there is no reason to produce a residual, so no residuation occurs.

If the child is an instance of to a previously derived function body, it can be folded back into an application of that function. This process is discussed in Subsection 3.3.2. Otherwise, a residual expression is produced and supercompilation continues.

3.5.5 Inlining Unnecessary Residuals

Supercompilation terminates when every residual has been processed. Residuation creates many function applications that would add overhead if they were to be executed in this form.

Many of these residuals can be inlined back, after supercompilation, into their originating expression. We inline application to a residual that is not recursive across our supercompiled functions. As our inline transformation uses $\textit{let}$ expressions to preserve sharing, we then apply the $\text{inlineLet}$ and $\text{removeLet}$ simplifications (see Figure 3.12) to remove any redundant abstraction.

3.6 Requirements A Transformation Framework

Our analysis so far has indicated several requirements to be able to perform transformations such as fold/unfold and supercompilation. One requires a method for specifying transformations in a manner similar to that defined in Section 3.1. Basic transformation functions (Subsection 3.2.1) and primitive transformations (Section 3.3) need to be provided, so that transformations and transformation strategies can be constructed using them. Throughout the analysis there have been warnings about variable capture and the use of fresh variables has been highlighted. The transformation framework must take these into account.

Existing systems Existing program transformation systems were investigated in Chapter 2. Common themes included; (a) the ability to control the tree traversal strategy, (b) the potential to transform arbitrary languages, (c) a facility by which one could introduce custom transformations to the system. Interactive program transformation tools, in particular, needed to provide an intuitive user interface suitable for “any novice functional programmer”.

43
3 Problem Analysis

Arbitrary languages Supercompilation appears to be particularly well suited to call-by-need, functional languages. This project is orientated towards work on a particular platform so it seems unnecessary to provide full flexibility in the types of languages that can be operated upon.

However, it is convenient to abstract transformation operations away from the actual target language, as far as is practical to so. It would be unfortunate to have to completely re-implement the system if changes to the f-lite specification occur.

History and annotation Whether transformations are being applied interactively or automatically, it is useful to maintain a history of previous derivations for later analysis. For a strategy like supercompilation, it is essential for the control of residuation.

The annotation of these derivations, with information about what transformations were performed to produce the new result, allows the debugging of transformations and gives greater insights into the path of derivation in proof and optimisation.

To this end, two additional low-level operations should be supplied. One is to introduce an annotation attached to the current instance. In the f-lite language, annotations can be introduced as comments at declaration level. Another operation is to clone a current instance and its annotations as a historical definition. As only the first function definition is consider the ‘active’ definition, historical definitions can be stored directly below it in the abstract syntax tree.

Automatic vs. Interactivity A full automatic program transformer merely requires an original program and a strategy to perform over the abstract syntax tree. For human-directed program transformation, the tool needs to expose information about which transformations are applicable given certain states and should give the user the ability to target specific expressions.

Systems found in literature, such as CIP (Bauer, 1979) and ZAP Feather (1982) used control languages entered at a command-line to instruct the tool. These are very much a product of their age. Given today’s profusion of graphical user interfaces for software development, a minimum requirement would be a cursor and dialogue based interface.

The output from an interactive transformer should be ‘pleasant’ for the human to read. The rules for generating fresh variable names should be derived with care and program code should be formatted according to language style guidelines. When displayed for manipulation, syntax highlighting would ease the users’ interactions.

3.7 Summary

We have presented a notation for describing transformations (Section 3.1) and defined the preliminary ‘building blocks’ of transformations (Subsection 3.2.1). A core language for transformed programs has been defined (Section 3.2), a normalised form of the Reduceron 2’s f-lite, taking advantage of already implemented transformations.

Using these as a foundation, we have described a variety of transformations, from the primitives in Section 3.3 to the complex supercompilation strategy in Section 3.5. The issues involved in applying supercompilation to the f-lite language have been emphasised and potential solutions have been identified.
Finally, drawing on the background provided by the literature and the prerequisites
of transformation that have become apparent through the course of the Problem Ana-
ysis, requirements have been extracted for the transformation tool. These are listed in
Figure 3.15

The interactive transformation system requires;

1. to accept any program written in our core language, normalised f-lite, as input but
   be flexible enough to handle changes in language specification without requiring
   significant revising,

2. a format for describing transformations and strategies, like those in Section 3.3,
   Section 3.4 and Section 3.5 preferably in a form similar to that defined in Section 3.1

3. to display the current state of the programs abstract syntax tree, in a human-friendly
   format following standard style guidelines,

4. to permit specific subexpression to be targeted for transformation and to inform
   the user which transformations are applicable,

5. to maintain a historical trace of previously derived results and to document how
   results were derived, both for the user’s information and to support complex
   strategies decision making processes,

6. to output to file the final state of the transformed program in human-readable and
   readily executable forms.

Figure 3.15: Requirements for Optimus Prime
4 Design and Implementation

We develop the requirements and concepts described in the previous chapter into a working system, dubbed Optimus Prime. The implementation can be broken down into a number of discrete ‘tasks’, illustrated by Figure 4.1.

The parser and zipper conversion (discussed in Section 4.1) read an f-lite source file into Optimus Prime’s internal syntax representation. Transformations are implemented using the techniques described in Section 4.2 and Section 4.3 and strategy specifics are highlighted in Section 4.4. Interfaces for controlling the transformation process and producing pretty printed output are discussed in Section 4.5.

The implementation specifics for implementing Fold/Unfold and Supercompilation are presented in Section 4.4.

4.1 Representation through Syntactic Zippers

This project is essentially concerned with the representation and manipulation of program syntax. We need a suitable data structure. It must be easily manipulated and it must hold the information required to perform transformations.

Simple trees Initially, a typed tree structure, similar to that described in Listing 2.13 was considered to represent the abstract syntax tree. It would be advantageous to implement new transformations in a similar manner to Naylor’s existing optimisations. Furthermore, as Naylor’s technique is inspired by Mitchell and Runciman (2007) Uniplate, the Supercompilation transformations described by Mitchell (2008) could be easily mapped onto that model.
4.1 Representation through Syntactic Zippers

---

**Zipper position**

```hs
type ZipPath = [Int]
```

---

**Syntactic zipper node**

```hs
data ZipParent a = NA (ZipParent a) -- Invertable Tree Node
| H -- Empty Tree Node
```

---

**Syntactic zipper structure**

```hs
type Zip a = (ZipPath, ZipParent a, BindingMap (ZipParent a))
```

---

**Type class of a zippable syntax structure**

```hs
class Zippable a where
  -- Swap a zipper node with a tree nodes child
  holeify' :: Int -> ZipParent a -> a -> (a, ZipParent a)

  -- Number of children a tree node has
  expWidth' :: a -> Int

  -- Add an expressions bindings to the scope
  addBindings :: Int -> ZipParent a ->
    BindingMap (ZipParent a) ->
    BindingMap (ZipParent a)

  -- Remove an expressions bindings from the scope
  removeBindings :: ZipParent a ->
    BindingMap (ZipParent a) ->
    BindingMap (ZipParent a)

  -- Initialise the variable map
  initFresh :: Zip a -> VarMap
```

Listing 4.1: Syntactic zipper types

---

**Huet's Zipper** The issues is that in a purely functional language, data structures are not mutable. “For trees, this amounts to modifying an occurrence in a tree non-destructively by copying its path from the root of the tree.” (Huet, 1997) Non-mutable tree structures are not well suitable for editor buffers, states Huet (1997). Any modification requires traversal down the tree to the appropriate point and the reconstruction of the tree back to the root.

Instead, he suggests a zipper, a tree “turned inside-out like a returned glove”. The structure holds a description of the current position, the tree from the current position down and an inverted tree back to the root.

**Syntactic Zipper** Our interpretation of a zipper differs from Huet’s. An attempt was made to generalise the concept to any abstract syntax tree definition. Our zipper (defined in Listing 4.1 and visualised in Figure 4.2) contains an integer list representation of the path to the current position in the tree and a ZipParent structure. The ZipParent structure contains a syntax node and either a terminating symbol or another ZipParent structure describing the path back to the tree root.

Any traversal or update operation (see Listing 4.2) on a Zipper structure should
Figure 4.2: Example of a zipper structure. The left shows a zipper at the root of the tree. The right illustrates the zipper after being navigated to the first child of the second child of the root.

\[
\begin{align*}
  f &= 1 \\
  g \ f &= \text{let} \{ h = f ; \} \text{ in let} \{ f = 2 ; \} \text{ in } f
\end{align*}
\]

Figure 4.3: Example of scoping issues

maintain the following invariants (in terms of the data types in Listing 4.1):

- For any ZipParent \( N \ e (N \ e' \ p') \) (a context), exactly one child of \( e' \) is \( H \). This child is the placeholder for the subject, \( e \).
- For any other, non-context ZipParent, no children may be \( H \). No placeholders are required.
- In a Zipper \( (zp, N \ e \ p, bm) \), \( zp = [ ] \) if and only if \( p = H \). Both of these conditions imply that the zipper is positioned at the root of the tree.
- Each element of \( zp \) relates to the position of the placeholder child in each context. Therefore, the length of \( zp \) is equal to the distance moved down the tree.

**Scoping for Zippers** Section 3.2 highlighted that f-lite allows variable names to be reused in separate scopes. For example, in Figure 4.3 the variable \( f \) is defined three times. There is a top-level definition of \( f \), the function \( g \) uses \( f \) as an argument and \( f \) is used as a local variable.

One must ensure that a variable name is referring to the appropriate definition. The first use of \( f \) (in the first let) refers to the argument of \( g \), but the second refers to the \( f \) defined in the second let.
4.1 Representation through Syntactic Zippers

--- Build a zipper out of a zippable tree
initZip :: Zippable a ⇒ a → Zip a

--- Move to parent of the current node
moveUp :: Zippable a ⇒ Zip a → Zip a

--- Move to a children of the current node
moveDown :: Zippable a ⇒ Int → Zip a → Zip a

--- Move to the root
moveTop :: Zippable a ⇒ Zip a → Zip a

--- Move to the preceding sibling
moveLeft :: Zippable a ⇒ Zip a → Zip a

--- Move to the succeeding sibling
moveRight :: Zippable a ⇒ Zip a → Zip a

--- Is this the rightmost sibling?
isEdge :: Zippable a ⇒ Zip a → Bool

--- Does this node have any children?
isTerminal :: Zippable a ⇒ Zip a → Bool

--- Replace the expression in the current node
update :: Zippable a ⇒ a → Zip a → Zip a

--- Apply a function to the current node
updateWith :: Zippable a ⇒ (Zip a → a) → Zip a → Zip a

--- Apply a monadic function to the current node
updateWithM :: (Zippable a, Monad m) ⇒ (Zip a → m a) → Zip a → m (Zip a)

Listing 4.2: Primitive syntactic zipper operations
Design and Implementation

---

**Representation of variable bindings**

```haskell
import qualified Data.Map as Map

-- Representation of variable bindings

**Type** BindingMap a = Map.Map String [VariableType a]

---

**Variable binding**

data VariableType a = Global Int [String] a — Global function
| Known Int a — Local definition
| Unknown Int — Function argument
| Substitution a — Substitute for expression

Listing 4.3: BindingMap data structure
```

This ability to reuse variable names in separate scopes is a common feature of programming languages. Several options for handling scoping issues were considered. These included: (a) only allowing variable names to be used once for the entire program, not just within scopes, (b) traversing back up the tree to find the nearest appropriate binding, (c) storing the depth of the appropriate binding with the variable reference.

Option (a) was discounted for making code too awkward to write and, as a result, less readable on output. Option (b) was considered too expensive an operation at the point of use. Option (c) seemed to require far too much computation on a manipulation of the tree.

Instead, we abstract the concept of scopes to a structure known as a *binding map* (Listing 4.3). The binding map maintains a stack of definitions for each variable name at a given point in the zipper. Any traversal or update operation on a zipper must ensure that the binding map is maintained.

**Zipper f-lite and Parsing**

The syntax described in Section 3.2 is represented by the zipper structure defined in Listing 4.4. Naylor’s own parser is used to read in f-lite source files. The parser uses the data type defined in Listing 2.13. This representation is then converted into the zipper type for f-lite. This approach reduces development time and ensures that no major deviations from Naylor’s f-lite are made.

4.2 Zipper Query and Modification

Primitive operations for moving through a zipper structure were shown in Listing 4.2. However, more abstract query and modification functions are required for the specification and application of transformation strategies.

Similar to uniplate, our query operations (Listing 4.5) produce lists of zipper nodes fitting a relationship with the input node. These lists can be queried further through list comprehensions to select nodes meeting specific requirements. For example, the function `appPaths` (as defined in Listing 4.6) returns the list of `ZipPaths` where applications occur in the descendants of a.

Modifications can be propagated through a zipper using the modification operations presented in Listing 4.7. These apply a partial transformation function to appropriate nodes in the tree. Where a transformation is not applicable to a certain structure, no
4.2 Zipper Query and Modification

```haskell
type ZipFLite = Zip FLite

type Id = String

type Binding = (Id, ZipParent FLite)

data FLite = Prog [ZipParent FLite]
  | Decl Id [Id] (ZipParent FLite)
  | App (ZipParent FLite) [ZipParent FLite]
  | Let [Binding] (ZipParent FLite)
  | Case (ZipParent FLite) [ZipParent FLite]
  | Alt (ZipParent FLite) (ZipParent FLite)
  | Var Id
  | Con Id
  | Int Int
  | Annotations [String]
  | Selected (ZipParent FLite)  -- Used for display

instance Zippable FLite where
...
```

Listing 4.4: Zipper structure for f-lite

---

*A list of a nodes children*

children :: Zippable a ⇒ Zip a → [Zip a]
children x@(_, e, _) = map (flip moveDown x) [0 .. (expWidth e - 1)]

---

*A list of a nodes decendants, pre-order traversal*

decendants :: Zippable a ⇒ Zip a → [Zip a]
decendants x = concat [c : decendants c | c <- children x]

---

*A list of all nodes in a subtree, pre-order traversal*

subtree :: Zippable a ⇒ Zip a → [Zip a]
subtree x = x : decendants x

Listing 4.5: Zipper query functions

```haskell
appPaths :: Zip FLite → [ZipPath]
apppaths z = [zp | (zp, N (App _) _, _) <- decendants z ]
```

Listing 4.6: Example of zipper query using list comprehensions
4 Design and Implementation

--- Apply a transformation to each child
updateChildren :: Zippable a ⇒ (Zip a → Fresh a) →
  Zip a → Fresh (Zip a)

--- Apply a transformation to a subtree, pre-order traversal
updateSubtree :: Zippable a ⇒ (Zip a → Fresh a) →
  Zip a → Fresh (Zip a)

--- Apply a transformation to a subtree, post-order traversal
updateSubtreeB :: Zippable a ⇒ (Zip a → Fresh a) →
  Zip a → Fresh (Zip a)

--- Apply, pre-order but fail if no modification is made
updateSubtreeF :: Zippable a ⇒ (Zip a → Fresh a) →
  Zip a → Fresh (Zip a)

--- updateSubtreeF until no more modifications can be made
normalise :: Zippable a ⇒ (Zip a → Fresh a) →
  Zip a → Fresh (Zip a)

--- Normalise, keeping note of what transformations are applied
normalises :: Zippable a ⇒ [(String , Zip a → Fresh a)] →
  Zip a → Fresh ([String], Zip a)

--- Listing 4.7: Zipper modification functions

change is made. However, updateSubtreeF is undefined when no modifications are
made to any node in the subtree.

The normalise function applies a transformation throughout a tree, repeatedly, until
no more applications are possible. In the variant normalises, a list of transformations
and names are supplied. An accumulator holds a list of all the transformations that have
been applied during the normalisation process.

Discussion Several traversal techniques and libraries exist for querying and updating
immutable trees. Mitchell and Runciman (2007) present a library for "generic traversals
over recursive data structures" by alleviating the need for so called boilerplate code.

Listing 4.8 shows boilerplate for performing query and modification operations on
a simple tree structure. Much of the code is repeated for different pattern expressions.
Mitchell’s Uniplate library removes the mundane code by abstracting all query and
modification operations to use a single uniplate function.

An instance of Uniplate typeclass is written for each structure by implementing the
uniplate function. An example Uniplate instance is shown in Listing 4.9 along with
Uniplate implementations of the consts and vars functions.

Our Syntactic Zipper data structure replicates this functionality. Similar to Uniplate,
an instance is written for the Zippable typeclass. However, an intermediary type,
ZipParent, must be inserted into the tree structure. This is to handle the ‘holes’ left
when child is moved to be the subject of the zipper. Listing 4.10 illustrates the adjusted
form of Logic, named LogicZ and its corresponding Zippable instance.
4.2 Zipper Query and Modification

```
data Logic = And Logic Logic
| Or Logic Logic
| Not Logic
| Variable String
| Constant Bool

consts :: Logic → [Bool]
consts (And x y) = consts x ++ consts y
consts (Or x y) = consts x ++ consts y
consts (Not x) = consts x
consts (Variable _) = []
consts (Constant x) = [x]

vars :: [(String, Bool)] → Logic → Logic
vars bs (And x y) = And (vars bs x) (vars bs y)
vars bs (Or x y) = Or (vars bs x) (vars bs y)
vars bs (Not x) = Not (vars bs x)
vars bs (Variable x) | isJ ust b = Constant (fromJust b)
| otherwise = Variable x

where
  b = lookup x bs
vars bs (Constant c) = Constant c

Listing 4.8: Example of boilerplate code

instance Uniplate Logic where
  uniplate (And x y) = ( Two (One x) (One y), \ (Two (One x’) (One y’)) → And x’ y’)
  uniplate (Or x y) = ( Two (One x) (One y), \ (Two (One x’) (One y’)) → Or x’ y’)
  uniplate (Not x) = ( One x, \ One x’ → Not x’)
  uniplate x = ( Zero, \ Zero → x)

consts :: Logic → [Bool]
consts l = [ x | Constant x ← universe l ]

vars :: [(String, Bool)] → Logic → Logic
vars bs l = transform vars’ l
where
  vars’ (Variable x) | isJ ust b = Constant (fromJust b)
  where
    b = lookup x bs
  vars’ x = x

Listing 4.9: Uniplate representation of [Listing 4.8]
data LogicZ = And (ZipParent LogicZ) (ZipParent LogicZ)  
| Or (ZipParent LogicZ) (ZipParent LogicZ)  
| Not (ZipParent LogicZ)  
| Variable String  
| Constant Bool

instance Zippable LogicZ where
  holeify' 0 h (And x y) = (And h y, x)
  holeify' 1 h (And x y) = (And x h, y)
  holeify' 0 h (Or x y) = (Or h y, x)
  holeify' 1 h (Or x y) = (Or x h, y)
  holeify' 0 h (Not x) = (Not h, x)

  expWidth' (Variable _) = 0
  expWidth' (Constant _) = 0
  expWidth' (Not _) = 1
  expWidth' _ = 2

  addBinding _ _ = id
  removeBinding _ _ = id

cons :: Zip LogicZ → [Bool]
cons l = [ x | (_, N (Constant x) _, _) <- subtree l ]

vars :: [(String, Bool)] → Zip LogicZ → Zip LogicZ
vars bs l = updateSubtree vars' l
  where
    vars' (Variable x) | isJust b = Constant (fromJust b)
      where
        b = lookup x bs
    vars' x = die

Listing 4.10: Our zipper representation of Listing 4.8
4.3 Transformation Specification

Transformations were specified in Section 3.1. In terms of the structures that have been introduced so far, a transformation maps one syntactic zipper to another.

**Zipper unsafe transformation**   A Haskell function can be defined such it only pattern matches on applicable expressions. Where a transformation is not appropriate, the function returns the zipper, unmodified. [Listing 4.11] illustrates a transformation written in this style.

**Zipper safe transformation**   A transformation should only alter the structure of the current zipper expression, refraining from modifying the ancestry and leaving the zipper in the same position. The binding map should be updated to reflect any effects the modification of the expression has on the current scope.

A transformation may still require the use of this contextual information while not modifying it directly. The `updateWith` function (type signature defined in [Listing 4.2]) modifies a zipper using a function that takes the full zipper as its input, but only returns...
4 Design and Implementation

emptyLet :: Zip FLite → FLite
emptyLet (_, N (Let [] (N e H)) _, _) = e
emptyLet (_, N e _, _) = e

Listing 4.12: Simpler empty let removal for f-lite

emptyLet :: Zip FLite → Maybe FLite
emptyLet (_, N (Let [] (N e H)) _, _) = Just e
emptyLet _ = Nothing

Listing 4.13: Monadic empty let removal for f-lite

the new sub-expression structure. This ensures that the criteria above are met. Listing 4.12 represents the same transformation written for the updateWith function.

Applicable transformation  So far, when a transformation is not appropriate for a particular expression, the expression is returned unmodified. However, a strategy may require the knowledge of whether a transformation has been applied or not. A transformation could be defined as a partial function of zipippers to expressions.

Haskell allows the encoding a partial function using the Maybe monad. Where the output of the function is undefined, it returns Nothing. Otherwise, the function returns Just the value. The updateWithM operation lifts the partial computation results to the whole zipper. Listing 4.13 provides an example of our example transformation rewritten for this format.

Fresh variable supply  Several operations and transformations described in Chapter 3 require the production of fresh variable names.

Our solution is to use a variable map as defined by in Listing 4.14. A map is maintained that holds indexes for variable prefixes. When a new fresh variable name is requested using the getFresh function the index for that variable prefix will be incremented.

The advantage of using a map of variable prefixes is that fresh variable names can be requested that are similar to an existing variable name. For example, when ‘freshening’ the variable names in a function body before inlining, fresh variables can be selected so that the function still reads similarly to the original definition.

This map could have been held inside the Syntactic Zipper as it already holds other information about the abstract syntax tree. However, so far, only information that is updated on tree traversal (the zipper position, subject, context and scope) have been included and it would seem out of improper to place a structure that is constant regardless of tree position.

Instead, a State monad is used to encapsulate the current state of the variable map. The State monad can be used to alleviate the need to pass around the state explicitly, while still fitting the functional model.

The State monad is coupled with the Maybe represented by using a monadic transformer to form the Fresh monad. Both the state and result can be backtracked when a transformation is not appropriate. As this composite Monad is an instance of the Monad typeclass, it can still operate with the updateWithM function, as above.

56
4.4 Transformation Specifics

4.4.1 Low-Level Definitions

In [Subsection 3.2.1] several preliminary meta-syntactic functions were abstractly defined. Some, such as relationships between expressions, are implemented directly by the traversal methods described in [Section 4.2].

Others require concrete Haskell definitions so that the transformations defined in [Section 3.3] [Section 3.4] and [Section 3.5] can be implemented in similar fashions. [Listing 4.15] outlines the type signatures for these functions.

Two versions of the ‘free variables of an expression’ function are defined. One that only lists each variable name once for any number of free occurrences in an expression (freeSet) and one that lists a variable name each time there is a free reference (freeMulti).

The substitution function, `subs`, is target language specific. The usage, shown in [Listing 4.15], is not far removed from the notation used throughout the previous chapter. The tree is traversed in a bottom-up manner and any number of substitutions can be made in one application of the substitution function.

The functions `length` and `zip` are already implemented by the Haskell 98 prelude. The `(+++)` combinator performs the same general function as append, but we use the opportunity to define one that produces required by transformation.

The transformation documentation functions `clone` and `annotate` (as required by [Section 3.6]) are f-lite specific, like `subs`.

4.4 Transformation Specifics

import qualified Data.Map as Map
import Data.Char

type VarMap = Map.Map String Int

type Fresh a = StateT VarMap Maybe a

splitName :: String → (String, Int)
splitName xs = (reverse s_, if null n_ then o else (read . reverse) n_) where
    (n_, s_) = (span isDigit . reverse) xs

getFresh :: String → Fresh String
getFresh s = let (s’, _) = splitName s in do
    vs ← get
    n ← return (Map.findWithDefault o s’ vs + 1)
    put (Map.insert s’ n vs)
    return (s’ ++ show n)

Listing 4.14: VariableMap state

57
### 4 Design and Implementation

--- Lists the names of any free variable in the expression, no duplicates
```haskell
freeSet :: Zip a → [String]
```

--- Lists the names of any free variable in the expression, includes multiple references
```haskell
freeBag :: Zip a → [String]
```

--- Substitution, where "x 'subs' [(v, y)]" = "x [ v / y ]"
```haskell
subs :: Zip FLite → [Binding] → Zip FLite
```

--- Included in Haskell 98 prelude
```haskell
length :: [a] → Int
```

--- Combines two lists of zippers into one list of expressions
```haskell
append :: [Zip a] → [Zip a] → [ZipParent a]
```

--- Included in Haskell 98 prelude
```haskell
zip :: [a] → [b] → [(a, b)]
```

--- Clone the current function definition into the history
```haskell
clone :: Zip FLite → Zip FLite
```

--- Add a list annotations to the current list attach to a function definition
```haskell
annotate :: [String] → Zip FLite → Zip FLite
```

Listing 4.15: Preliminary definitions

#### 4.4.2 Transformation Rules and Strategies

The Fold/Unfold laws are implemented, as described in Section 3.4, using the techniques manner described in Section 4.3 and the low-level functions described above. The ‘eureka’, ‘instantiation’ and ‘laws’ rules require an command-line interface for the user to enter definitions and patterns. The ‘abstraction’ rule searches for common subexpression within the currently selected expression. Only the fold and unfold laws can operate autonomously.

The supercompilation strategy (of Section 3.5) is similarly defined out of the transformation ‘building blocks’. The simplification transformations are defined using the techniques presented in Section 4.3 and the normalises modification function is used to apply and annotate these transformations across a definition.

An scUnfold transformation is defined over function declarations. This transformation inlines the first function application that, upon inlining and simplification, does not cause the termination criteria to be triggered. Failing this, the first function application is inlined. The transformation is undefined when no function can be unfolded.

The selective unfolding transformation and the simplifying normalisation are combined to form the driving function. When a termination criterion is triggered, generalisation occurs where appropriate, and functions are residuated as described in Section 3.5.

The driving process is then triggered on the residual functions until all residuals have
Figure 4.4: The supercompilation algorithm
been supercompiled. Any non-recursive functions are inlined to reduce unnecessary function application overhead. Figure 4.4 depicts the algorithm as a flowchart.

### 4.5 Output and Control

We have previously discussed how programs are parsed into the Syntactic Zipper representation (Section 4.1). However, the state of the internal representation needs to be returned to the user in an appropriate form. Furthermore, either the user or some other decision making mechanism need to guide the transformation process through appropriate interfaces.

#### 4.5.1 Pretty printing

The abstract syntax tree needs to be returned to the user in the original programming code format. We use the `wl-pprint` combinator library to produce output in the literate and core f-lite forms. An extension to the `wl-pprint` library, `ansi-wl-pprint`, is used to produce syntax coloured output for display during interactive transformation. Both of these libraries are based on the paper ‘A Prettier Printer’ by Wadler (1998) and are freely available in the Hackage repository.

This library provide a large variety of combinators for concatenating ‘documents’ together. To name a few possible operations, documents can be combined with a space between, with a line-break between or, more interestingly, with a line-break only where a space would cause the line to be too long.

A maximum line width and a relative ribbon width are supplied to the algorithm. These determine how long a line can be and introduce line-breaks where appropriate to enforce these restrictions wherever possible.

The appropriate use of these combinators allow us to produce program outputs that conform to standard Haskell ’98 style guidelines, making it easier for humans to read the resulting code.

Additional functions in the `ansi-wl-pprint` library utilise ANSI escape sequences to produce colours on a terminal supporting their display. The functions allows the colour coding of syntax when displaying programs in an interactive fashion. Syntax highlighting makes interpreting programs a far more enjoyable experience for the user.

Using these two, closely related, libraries, three output formats can be generated for a given f-lite abstract syntax tree.

**Display** The display format is shown, on-screen, during interactive transformation using the `ansi-wl-pprint`. Code is formatted in layout-sensitive Haskell style, where braces and semicolons are omitted. Colour-coded syntax highlight is used to improve the users’ experience. A cursor is displayed for the currently selected expression. Figure 4.5 shows the display output.

**Literate** The literate format uses the Literate Haskell style to permit rich contextual information about transformation derivations. The definitions are surround by \LaTeX comments describing the transformation steps. Current definitions are enclosed in code environments while previous derivations are in spec environments. This allows a compiler to distinguish between them. The lhs2TeX tool allows these
4.5 Output and Control

Literate Haskell files to be compiled into \LaTeX\ documents. Appendix D shows an example of the literate output.

**Tidy** The tidy format is the most readily executable form. Only the current definitions are used and then only the definitions that are reachable from the main function. Code is formatted in the layout-insensitive explicit style, as required by the f-lite specification. The tidy format can be immediately executed by Naylor’s compiler and interpreter. Listing E.1 shows an example of tidy output.

### 4.5.2 Interactive Control

Using the display output to view the current state of the Syntactic Zipper, the user traverses the abstract syntax tree uses in the arrow keys. Right moves to a child expression and left to the parent. The down and up keys move through siblings. The currently select expression is given a background.

To perform a transformation, the user presses the ‘T’ key and is presented with a menu of permissible transformations for this expression. An example of the interface in this state is shown in Figure 4.5.

The current state of the Syntactic Zipper can be saved to a file using the ‘L’ key for literate output and ‘S’ for tidy output.
4.5.3 Autonomous Control

Autonomous control is created quite simply by applying encapsulating interactively activated transformation strategy in a command-line tool. Programs are passed in, along with any strategy parameters, through command-line arguments. The program is then returned to stdout in either the tidy or literate formats described above.

4.6 Summary

Optimus Prime is a tool that permits both interactive and automatic transformations on f-lite programs. Transformations are specified as operations on the tool’s internal representation of the abstract data structure, the Syntactic Zipper. All the requirements noted at the end of Chapter 3 have been fulfilled as follows.

The structure, operations on the structure and transformation types have been detailed in Section 4.1, Section 4.2 and Section 4.3. These fulfil to Requirements 1 and 2 in Figure 3.15.

The generic capabilities of the Syntactic Zipper library permit the f-lite language to be extended without having to change specific traversal functions and most existing transformations. While it is slower at full tree traversal then other techniques, it is able to support the concept of an interactive cursor through expressions. So it fulfils Requirements 1 and 4 in Figure 3.15.

A Wadler (1998) style pretty printer is used to produce a variety of outputs for varying needs (Subsection 4.5.1). One of these outputs is used to display the abstract syntax tree during interactive transformation and allows the user to move through the tree in an intuitive manner (Subsection 4.5.2). In this way we fulfil to Requirements 3, 4 and 6 in Figure 3.15.

Finally, functions have been made available to store histories and annotate transformation traces, in accordance with to Requirement 5 of Figure 3.15.
5 Results and Evaluation

This project has investigated techniques for performing interactive transformation and applying the Supercompilation strategy to the f-lite language. In this chapter, we discuss the results obtained.

The performance of the core component of Optimus Prime, the Syntactic Zipper library described in Section 4.1 and Section 4.2 is assessed in Section 5.1.

The capabilities of the Optimus Prime Supercompiler for the f-lite language are evaluated in Section 5.2. Supercompilation times and other statistics are collected for the supercompilation of a selection of programs. The performance of these supercompiled programs is then compared to the performance of the original programs.

5.1 Performance of the Syntactic Zipper Library

In Section 4.2, the performance of the Syntactic Zipper library was discussed and briefly compared with other tree traversal techniques. Table 5.1 gives a more comprehensive comparison between the boilerplate, Uniplate and Syntactic Zipper approaches to tree traversal. Code for the experiments is provided in Listing B.1.

Four operations are considered. The \texttt{consts} functions are the same as those described in Section 4.2, returning a list of all the constants in a logic tree. The \texttt{searchOr} functions return a list of all the \texttt{OrZ} terms in a tree. The \texttt{eval} functions use tree transformation to reduce a logic tree containing only constants to a boolean value. The \texttt{subs} functions inverts the value of every constant in the tree.

In every case, the Syntactic Zipper takes the longest to perform both the query and modification operations. However, its awareness of context allows immediate traversals to sibling and parent nodes. This type of operation, in addition to being able to hold a zipper at a particular location, facilitates interactive targeted transformation.

We therefore argue that where interactive transformation is a requirement, despite the performance overheads, the Syntactic Zipper is a more acceptable solution than using the boilerplate or Uniplate techniques.

5.2 Performance of the Supercompiler

The supercompiler is an optimising transformation. To gauge its success, we need to determine the execution time of supercompiled programs relative to their original forms.

However, this is not the only measure of a compiler optimisation. We investigate the time that the supercompilation process takes, and provide other statistics to give better insight into the supercompilation process.

\footnote{Values were obtained using the UNIX time command on a 2.16 GHz Intel Core Duo, Apple MacBook Pro, with 2GB of RAM running Mac OS X 10.5.6.}
5. Results and Evaluation

Table 5.1: Comparison of tree query and modification techniques (lower is better and worst scores are bold)

<table>
<thead>
<tr>
<th>Test</th>
<th>Library</th>
<th>2^8</th>
<th>2^10</th>
<th>2^12</th>
<th>2^14</th>
<th>2^16</th>
</tr>
</thead>
<tbody>
<tr>
<td>consts</td>
<td>boilerplate</td>
<td>0.004s</td>
<td>0.005s</td>
<td>0.008s</td>
<td>0.018s</td>
<td>0.053s</td>
</tr>
<tr>
<td></td>
<td>uniplate</td>
<td>0.004s</td>
<td>0.006s</td>
<td>0.006s</td>
<td>0.016s</td>
<td>0.053s</td>
</tr>
<tr>
<td></td>
<td>zipper</td>
<td>0.005s</td>
<td>0.006s</td>
<td>0.015s</td>
<td>0.056s</td>
<td>0.241s</td>
</tr>
<tr>
<td>searchOr</td>
<td>boilerplate</td>
<td>0.008s</td>
<td>0.023s</td>
<td>0.101s</td>
<td>0.316s</td>
<td>1.832s</td>
</tr>
<tr>
<td></td>
<td>uniplate</td>
<td>0.007s</td>
<td>0.020s</td>
<td>0.072s</td>
<td>0.319s</td>
<td>1.821s</td>
</tr>
<tr>
<td></td>
<td>zipper</td>
<td>0.048s</td>
<td>1.046s</td>
<td>15.949s</td>
<td>&gt; 1m</td>
<td>&gt; 1m</td>
</tr>
<tr>
<td>eval</td>
<td>boilerplate</td>
<td>0.004s</td>
<td>0.005s</td>
<td>0.005s</td>
<td>0.004s</td>
<td>0.006s</td>
</tr>
<tr>
<td></td>
<td>uniplate</td>
<td>0.004s</td>
<td>0.005s</td>
<td>0.010s</td>
<td>0.021s</td>
<td>0.058s</td>
</tr>
<tr>
<td></td>
<td>zipper</td>
<td>0.005s</td>
<td>0.008s</td>
<td>0.017s</td>
<td>0.042s</td>
<td>0.153s</td>
</tr>
<tr>
<td>subs</td>
<td>boilerplate</td>
<td>0.006s</td>
<td>0.007s</td>
<td>0.023s</td>
<td>0.069s</td>
<td>0.232s</td>
</tr>
<tr>
<td></td>
<td>uniplate</td>
<td>0.005s</td>
<td>0.007s</td>
<td>0.021s</td>
<td>0.061s</td>
<td>0.219s</td>
</tr>
<tr>
<td></td>
<td>zipper</td>
<td>0.006s</td>
<td>0.012s</td>
<td>0.029s</td>
<td>0.128s</td>
<td>0.514s</td>
</tr>
</tbody>
</table>

5.2.1 Test Programs

Five test programs are used to assess the performance of the Optimus Prime supercompiler. They represent a variety of problem classes that should give an indication as to how the supercompiler would operate on a wider set of programs.

**Perms** The listing for the Perms test program can be found in Section C.1. The program calculates returns the size of the set of permutations for the list of the first four non-negative integers.

**Fibonacci** The listing for the Fibonacci test program can be found in Section C.2. The program returns the first eighteen elements of the Fibonacci series. The Fibonacci series is represented as an lazy infinite data structure and the integers themselves are represented by a data structure for Peano numerals.

**Twizzler** The listing for the Twizzler test program can be found in Section C.3. The `twiz` function takes the first integer of a list, `n`, and reverses the top `n` elements of the list. The `twizzle` function repeatedly applies `twiz` until the number 1 is at the top of the list. The Twizzler problem asks from what starting states does the `twizzle` function terminate with all the elements in order. This program counts the number of ordered results where the inputs are all the permutations \{1, 2, 3, 4\}.

**Word Count** The listing for the Word Count test program can be found in Section C.4. The program finds the number of words, separated by the space character, in the string "quick brown fox jumps over the lazy dog". Appendix D shows the supercompilation trace for the word counting program and Listing E.1 presents the final, supercompiled code. Appendix D shows the effect of supercompilation on the word counting program.
5.2 Performance of the Supercompiler

N-Queens  The listing for the N-Queens test program can be found in Section C.5. The problem asks in for an nxn size chessboard, how many unique solutions are there to placing n queens such that no queen is attacking another. This program calculates the result for the n = 10 instance.

5.2.2 The Supercompilation Process

Each program is passed through the supercompiler. Table 5.2 lists statistics concerning the time the supercompilation process takes\(^1\), the number of lines of program code before and after supercompilation, the number of definitions in the program before and after supercompilation and how many residuals were produced, prior to final inlining. Ratios are after/before, so that the supercompilation process producing less lines of code than the original results in a ratio that is less than one.

Further statistics regarding the simplification process are listed in Table 5.3. Occurrences of annotations relating to each simplification transformation are, were counted using the UNIX grep command.

5.2.3 The Supercompiled Programs

Performance of the programs is measured in two ways. Reductions are counted using the HUGS :s +s, statistics printing command. Reducerson clock-ticks are simulated using a compiler and emulator provided by Matthew Naylor.

For f-lite programs to be executable in HUGS, several alterations need to be made. Any function that is defined in the program that also exists in the Haskell 98 prelude needs to be prevented from being imported. This is done by adding a line like import Prelude hiding (map,concat,concatMap). In addition, any data constructor used must be defined explicitly. For example, data List a = Nil | Cons a (List a) is required for programs that use list structures.

Table 5.4 lists the execution performance of the test programs in both environments. Ratios are, once again, after/before, so that the supercompiled programs requiring less Reducerson ticks then the original program have ratios that are less than one.

The N-Queens program appeared to be broken by the supercompilation. An error was introduced by the supercompilation process such that the program no longer terminates. In all other cases, executed supercompiled code returned the same results as their original forms.

5.2.4 Discussion

The supercompilation process terminated for all five programs. The time taken to supercompile seems to be loosely related to the number of lines in the original source files but this is not a strict rule. It is more closely aligned with the count of the number of residuals produced, as one would expect.

Every supercompiled test program resulted in more lines of code than the original definition. This is probably due the “duplicating code but not work” (Mitchell, 2008, chap. 4) behaviour of some simplification rules.

\(^1\)Supercompilation time is calculated using the UNIX time command on a 2.16 GHz Intel Core Duo, Apple MacBook Pro, with 2GB of RAM running Mac OS X 10.5.6.
Results and Evaluation

Table 5.2: Effects of supercompilation on code

<table>
<thead>
<tr>
<th>Program</th>
<th>Time</th>
<th>Before</th>
<th>After</th>
<th>Ratio</th>
<th>Before</th>
<th>After</th>
<th>Ratio</th>
<th>Residuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perms</td>
<td>6.943s</td>
<td>47</td>
<td>4811</td>
<td>102.36</td>
<td>12</td>
<td>16</td>
<td>1.33</td>
<td>27</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>0.597s</td>
<td>33</td>
<td>228</td>
<td>6.91</td>
<td>8</td>
<td>12</td>
<td>1.50</td>
<td>16</td>
</tr>
<tr>
<td>Twizzler</td>
<td>1.134s</td>
<td>89</td>
<td>161</td>
<td>1.81</td>
<td>22</td>
<td>15</td>
<td>0.68</td>
<td>18</td>
</tr>
<tr>
<td>Word Count</td>
<td>0.137s</td>
<td>39</td>
<td>83</td>
<td>2.13</td>
<td>8</td>
<td>5</td>
<td>0.63</td>
<td>11</td>
</tr>
<tr>
<td>N-Queens</td>
<td>26.011s</td>
<td>77</td>
<td>400</td>
<td>5.20</td>
<td>21</td>
<td>16</td>
<td>0.76</td>
<td>58</td>
</tr>
</tbody>
</table>

Table 5.3: Supercompilation simplification statistics

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of simplifications performed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>appOfApp</td>
</tr>
<tr>
<td>Perms</td>
<td>3</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>0</td>
</tr>
<tr>
<td>Twizzler</td>
<td>0</td>
</tr>
<tr>
<td>Word Count</td>
<td>0</td>
</tr>
<tr>
<td>N-Queens</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of simplifications performed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>substiuteVar</td>
</tr>
<tr>
<td>Perms</td>
<td>0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>0</td>
</tr>
<tr>
<td>Twizzler</td>
<td>1</td>
</tr>
<tr>
<td>Word Count</td>
<td>6</td>
</tr>
<tr>
<td>N-Queens</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.4: Comparison of the performance of supercompiled programs (lower is better)

<table>
<thead>
<tr>
<th>Program</th>
<th>Hugs Reductions</th>
<th>Reduceron Clock-ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>Perms</td>
<td>594</td>
<td>455</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>236486</td>
<td>210307</td>
</tr>
<tr>
<td>Twizzler</td>
<td>5732</td>
<td>5196</td>
</tr>
<tr>
<td>Word Count</td>
<td>505</td>
<td>354</td>
</tr>
<tr>
<td>N-Queens</td>
<td>15494927</td>
<td>N/A</td>
</tr>
</tbody>
</table>
5.3 Summary

The ‘perms’ example produced a significant increase in the number of lines. This appears to have been caused by the inlining of many application of a non-recursive function, once supercompilation terminated. It would be interesting to observe the penalty caused if this inlining had not occurred.

No discernible pattern can be seen in the relationship between the number of definitions produced and any other statistic. The final number of definitions is determined by how many recursive residuals are left by the final inlining process and is likely program specific.

It is notable that in none of the five test programs were the (a) application to case expression, (b) application to a let expression, (c) moving a let expression within a case expression (d) or nested let concatenation transformations ever used. The first two are probably due to partial evaluation not being necessary or properly exploited in the original program code. The latter two require further investigation to explain.

The inlineLet transformation is heavily used due to the unfolding operation making use of lets to preserving sharing. Similarly, the removeLet transformation is heavily used to remove excess let expressions left over from inlineLets.

The caseOfCase and caseOfCons transformations are the driving force behind the supercompilation strategy, producing the evaluation effects that allow expressions to be reduced.

The supercompiled forms of all except the N-Queens program, indicated better performance than the original code in terms of Reduceron ticks and HUGS reductions. Some transformation appears to have changed the semantic meaning of the N-Queens program and this warrants further investigation.

Performance gains appear to come from programs where the intermediate data structures have been collapsed, as would have been achieved by fusion, and where separate functions have been composed. The word counting program builds up a number of intermediate data structures in its original form but the supercompiled version shows that these have been removed and the higher order functions have been specialised.

5.3 Summary

The Syntactic Zipper library, that lies at the heart of Optimus Prime, does take longer to perform query and modification than some other generic tree traversal techniques. Yet, the ability to traverse not just to a node’s children but to parents and siblings is essential functionality for an interactive transformation tool.

The Optimus Prime supercompiler has produced performance gains in all but one test so far. Further work is required to determine where semantic equivalence is lost in the N-Queens test. Similarly, our analysis of the results has shown that some simplification transformation are never triggered. It is of interest to discover what constructs precipitate their use and whether they occur in general use.
6 Conclusions and Further Work

This chapter first revisits the project’s five core aims, as laid out in Section 1.2. With reference to these and other experience from the rest of this dissertation, we shall propose topics for further research.

6.1 Satisfaction of Objectives

Interactive transformation  The goal was “to develop an interactive transformation tool, Optimus Prime, that can operate on the f-lite subset of Haskell 98.” An interactive tool has been developed, meeting all the requirements listed in Figure 3.15. The tool’s interactive transformation interface is described in Section 4.5.

Fold/Unfold for f-lite  In Section 3.4, we presented the Burstall and Darlington (1977) fold/unfold transformation rules, adapted for the particular constructs of f-lite. Section 4.4 discussed how these rules have been implemented on the Optimus Prime platform.

Supercompilation for f-lite  Section 3.5 discussed the particulars of applying Turchin (1979) supercompilation to the f-lite subset of Haskell 98. A set of twelve transformations were presented, for the simplification of f-lite expressions. The termination criteria and residuation functions of (Mitchell, 2008, chap 4.) were changed to take into account f-lite’s primitive functions.

Optimus Prime supercompiler  Using the blueprint described in Section 3.5, the supercompilation strategy was implemented in Optimus Prime for the f-lite language. It can be executed using Optimus Prime’s interactive interface, or autonomously through a command-line tool.

Effectiveness of the supercompiler  In Chapter 5, we assessed the speed of the supercompilation process and its core components. A selection of test programs were supercompiled, using the tool. Four out of five of the test programs showed a performance gain for the supercompiled version compared with the original form. One test program was rendered non-executable by the supercompilation process and possible reasons for its failure were discussed.

Overall, it would appear that the objectives outlined in Section 1.2 have been largely achieved. However, further work is required to reduce the time that supercompilation takes and to find the reason for its incompleteness.
6.2 Taking Zippers out of Context

During traversal, at every point in the tree, the Syntactic Zipper (Section 4.1) maintains a full record of its context. That is, a description of its position in the tree, a reversed pointer to its parent and the scope stacks for each variable.

For interactive transformation, this information is essential for a user to move from node to related node, without having to traverse down the complete tree. During automatic, strategy-driven transformation, these types of movements are not performed, so the contextual data being carried is a hindrance to performance.

The Syntactic Zipper representation could be replaced with a standard tree structure when performing autonomous transformations. However, transformations would need to be reimplemented for this new structure, eliminating some of the benefits of using interactive transformation to develop the techniques.

A compromise between the two positions would be to produce an alternative Syntactic Zipper library which could be substituted in for the existing one during unaided transformation. This new Nimble Syntactic Zipper library would only permit the decent through the zipper. This restriction means that the contextual information is no longer required. The reversed pointer to the parent can be omitted and only the top element of the scope stacks need be retained.

This would likely result in similar performance to that of the Uniplate library, without having to rewrite the transformations for a new structure and API.

6.3 Optimisation of the Homeomorphic Embedding Relation

Another bottleneck in the supercompilation process is the use of the homeomorphic embedding relation as a termination criterion. Each previous residual body is compared to the expression currently being supercompiled. The comparison traverses the full expression looking for sections that are similar the the derivation in question.

Mitchell (2008) suggests the use of Sillman’s (1989) algorithm to improve performance through the use of a memoization table. Mitchell reckons that there are performance gains to made from using this technique and it would be interesting to see if it would work in the context of our supercompiler variant for f-lite.

6.4 Supercompilation of Primitive Functions

Our interpretation of the supercompilation strategy treats primitive function applications as constructs that cannot be simplified. This is may be a premature assumption.

An equality primitive, \((==)\), being applied to the same constructor, integer or the same strict variable could be evaluated to \(True\). Similarly, \((+)\), the addition of two integers can be resolved to the result of the addition. The emit primitive produces side effects by emitting results from the Reduceron, but to the program, it merely acts as the
identity function. At supercompilation time, any input to the function could be passed
outward to a case expression for matching.

There is plenty of scope for performing further supercompilation through primitive
functions. However, termination criteria and generalisation functions will have to be
adjusted for the loss structural information.

It could also be argued that special rules for numeric primitives are necessary in
this context. The Reduceron 2 is orientated towards symbolic computations, those that
manipulate structures rather than perform arithmetic. As such, programs written for this
platform are likely to involve little arithmetic computation.

However, the primitive comparison of atomic values, such as integers and constructors,
occur in many programs. The ideas of the Sørensen et al. (2008) positive supercompiler or
the more general case, passing on both positive and negative information, could be used
to pass on inferred information about variable values.

6.5 Closing Remarks

The use of an interactive program transformation tool to prototype transformations
allows both user and developer to step through strategies and directly observe the effect
on code. We found that this substantially aided the debugging of the Optimus Prime
supercompiler.

The internal representation of abstract syntax trees in Optimus Prime, the Syntactic
Zipper, is a very appropriate data structure for interactive transformation. With further
refinement (as discussed in Section 6.2), it can be better utilised in autonomous, unaided
transformation systems.

The results presented in Section 5.2 show that there are significant performance gains
to be made from the use of the supercompilation strategy on Reduceron 2 programs. The
current form of the Optimus Prime supercompiler is only reasonable to use on small,
experimental programs. However, the Reduceron 2 is, similarly, not yet ready for use
with large scale applications. Both of these platforms can grow and stabilise, in parallel,
to handle large applications with time.
A Fold/Unfold Transformation Transcripts

A.1 Fibonacci Transformation Transcript

Based on the example in Burstall and Darlington (1977). Discussed in Section 3.4.

Listing A.1: An example of optimising a function using fold/unfold, expressed in f-lite

```f-lite
1 | ← Eureka → |
2 | fib x = case (<=) x 1 of |
3 | True → 1;
4 | False → (+) (fib ((-) x 2)) (fib ((-) x 1)) |
5 | };
6 | ← Instantiate ‘fib’ with ‘x <= 2’ and ‘x > 2’ → |
7 | fib x = case (<=) x 1 of |
8 | True → 1;
9 | False → case (==) x 2 of |
10 | True → (+) (fib ((-) 2 2)) (fib ((-) 2 1));
11 | False → (+) (fib ((-) x 2)) (fib ((-) x 1)) |
12 | }
13 | ← Primitive laws on ‘(−)’ and unfold ‘fib’ → |
14 | fib x = case (<=) x 1 of |
15 | True → 1;
16 | False → case (==) x 2 of |
17 | True → (+) (fib 0) (fib 1);
18 | False → (+) (fib ((-) x 2)) |
19 | ((+) (fib ((-) x 3)) (fib ((-) x 2))) |
20 | }
21 | }
22 | ← Unfold ‘fib’s, primitive laws on ‘(+’ and abstract ‘fib2’ → |
23 | fib x = case (<=) x 1 of |
24 | True → 1;
25 | False → case (==) x 2 of |
26 | True → 2;
27 | False → let { fib2 = fib ((-) x 2) } in |
28 | (+) (fib ((-) x 3)) ((+) fib2 fib2) |
29 | }
30 | }
```

Listing A.1: An example of optimising a function using fold/unfold, expressed in f-lite
A Fold/Unfold Transformation Transcripts

A.2 Append/Elem Proof Transcript

Discussed in [Section 3.4]

```haskell
(- Preliminary Definitions -)

elem x ys = case ys of 
  Nil → False;
  Cons y1 ys1 → or ((==) x y1) (elem x ys1)
);
or x y = case x of 
  True → True;
  False → y;
);
append xs ys = case xs of 
  Nil → ys;
  Cons x1 xs1 → Cons x1 (append xs1 ys)

(- Hypothesis -);

hLhs v xs ys = elem v (append xs ys);

hRhs v xs ys = or (elem v xs) (elem v ys);

(- Instantiate xs in hypothesis with _|_, Nil and (Cons x xs) -);

hLhs v xs ys = case xs of 
  _|_ → hLhs1 v vs;
  Nil → hLhs2 v vs;
  Cons x xs → hLhs3 v vs x xs;

);

hRhs v xs ys = case xs of 
  _|_ → hRhs1 v vs;
  Nil → hRhs2 v vs;
  Cons x xs → hRhs3 v vs x xs;

);

(- _|_ case -)

hLhs1 v ys = elem v (append _|_ ys);

hRhs1 v ys = or (elem v _|_) (elem v ys);

(- Strict argument in 'append' and 'elem' -);

hLhs1 v ys = elem v _|_;

hRhs1 v ys = or _|_ (elem v ys);

(- Strict argument in 'elem' and 'or' -);

hLhs1 v ys = _|_;

hRhs1 v ys = _|_;

(- QED -);

hLhs1 v ys = hRhs1 v ys;

(- Nil case -)

hLhs2 v ys = elem v (append Nil ys);

hRhs2 v ys = or (elem v Nil) (elem v ys);

(- Unfold 'append' and 'elem' -);

hLhs2 v ys = elem v ys;

hRhs2 v ys = or False (elem v ys);

(- Unfold 'or' -);
```

A.2 Append/Elem Proof Transcript

```
50  hRhs2 v ys = elem v ys;
51  {− QED −};
52  hLhs2 v ys = hRhs2 v ys;
53
54  {− (Cons x xs) case −}
55  hLhs3 v ys x xs = elem v (append (Cons x xs) ys);
56  hRhs3 v ys x xs = or (elem v (Cons x xs)) (elem v ys);
57  {− Unfold ‘append’ and ‘elem’ −};
58  hLhs3 v ys x xs = elem v (Cons x (append xs ys));
59  hRhs3 v ys x xs = or (or ((==) v x) (elem v xs)) (elem v ys);
60  {− Unfold ‘elem’ and apply law of associativity on ‘or’ −};
61  hLhs3 v ys x xs = or ((==) v x) (elem v (append xs ys));
62  hRhs3 v ys x xs = or ((==) v x) (or (elem v xs) (elem v ys));
63  {− Inductive hypothesis so fold/unfold hRhs/hLhs −};
64  hRhs3 v ys x xs = or ((==) v x) (elem v ( append xs ys ));
65  {− QED −};
66  hLhs3 v ys x xs = lRhs v ys x xs;
```

Listing A.2: Example of a fold/unfold proof, held in f-lite syntax
B Tree Traversal Comparison

This code is used to compare the performance three tree traversal techniques in Section 5.1.
(B) Boilerplate, (U) Uniplate and (Z) Syntactic Zipper.

```haskell
import Zipper
import Data.Generics.UniplateStr
import Data.Generics.Str
import FreshState
import Data.Maybe

-- Definition of the LogicZ data type
data LogicZ = AndZ (ZipParent LogicZ) (ZipParent LogicZ)
  | OrZ (ZipParent LogicZ) (ZipParent LogicZ)
  | NotZ (ZipParent LogicZ)
  | VariableZ String
  | ConstantZ Bool

-- Instance of Uniplate library for LogicZ
instance Uniplate LogicZ where
  unipl ate (AndZ x y) = (Two (One x) (One y)
                     , \ (Two (One x') (One y')) \rightarrow
                     AndZ (N x' H) (N y' H))
  unipl ate (OrZ x y) = (Two (One x) (One y)
                      , \ (Two (One x') (One y')) \rightarrow
                      OrZ (N x' H) (N y' H))
  unipl ate (NotZ x) = (One x , \ (One x') \rightarrow
                        NotZ (N x' H))
  unipl ate x = (Zero , \ Zero \rightarrow)

-- Instance of Syntactic Zipper library for LogicZ
instance Zippable LogicZ where
  holeify' 0 h (AndZ x y) = (AndZ h y, x)
  holeify' 1 h (AndZ x y) = (AndZ x h, y)
  holeify' 0 h (OrZ x y) = (OrZ h y, x)
  holeify' 1 h (OrZ x y) = (OrZ x h, y)
  holeify' 0 h (NotZ x) = (NotZ h, x)

  expWidth' (VariableZ _) = 0
  expWidth' (ConstantZ _) = 0
  expWidth' (NotZ _) = 1
  expWidth' _ = 2
  addBindings _ _ = id
  removeBindings _ = id

  initFresh = undefined
```
— Tree generation where \( n \) is tree depth. Number of nodes = \( 2^n \)

testZ :: Int \( \rightarrow \) LogicZ

testZ o = ConstantZ True

testZ n \mid odd n = AndZ (N (testZ (n - 1)) H) (N (testZ (n - 1)) H)
\mid otherwise = OrZ (N (testZ (n - 1)) H) (N (testZ (n - 1)) H)

— Consts function

costsB :: LogicZ \( \rightarrow \) [Bool]
costsB (AndZ (N x H) (N y H)) = costsB x ++ costsB y
costsB (OrZ (N x H) (N y H)) = costsB x ++ costsB y
costsB (NotZ (N x H)) = costsB x
costsB (VariableZ _) = []
costsB (ConstantZ x) = [x]

costsU :: LogicZ \( \rightarrow \) [Bool]
costsU l = [ x | ConstantZ x \( \leftarrow \) universe l ]

costsZ :: Zip LogicZ \( \rightarrow \) [Bool]
costsZ l = [ x | ( _ , N (ConstantZ x) _ , _ ) \( \leftarrow \) decendants l ]

— Search function

searchOrsB :: LogicZ \( \rightarrow \) [LogicZ]

searchOrsB (AndZ (N x H) (N y H)) = searchOrsB x ++ searchOrsB y

searchOrsB (OrZ (N x H) (N y H)) = (1:searchOrsB x) ++ searchOrsB y

searchOrsB (NotZ (N x H)) = searchOrsB x

searchOrsB (VariableZ _) = []

searchOrsB (ConstantZ x) = []

searchOrsU :: LogicZ \( \rightarrow \) [LogicZ]

searchOrsU l = [ l' | (l'@(OrZ _ _)) \( \leftarrow \) universe l ]

searchOrsZ :: Zip LogicZ \( \rightarrow \) [LogicZ]

searchOrsZ l = [ l' | (l'@(OrZ _ _)) \( \leftarrow \) decendants l ]

— Eval function

evalB :: LogicZ \( \rightarrow \) Bool

evalB (AndZ (N x H) (N y H)) = evalB x \&\& evalB y

evalB (OrZ (N x H) (N y H)) = evalB x || evalB y

evalB (NotZ (N x H)) = \neg (evalB x)

evalB (VariableZ _) = error "Can't evaluate variable"

evalB (ConstantZ x) = x

evalU :: LogicZ \( \rightarrow \) Bool

evalU l = case rewrite eval l of { ConstantZ x \( \rightarrow \) x }

where

eval (AndZ (N (ConstantZ x) H) (N (ConstantZ y) H)) = Just $ ConstantZ (x \&\& y)
eval (OrZ (N (ConstantZ x) H) (N (ConstantZ y) H)) = Just $ ConstantZ (x || y)
Listing B.1: Code for tree traversal tests
C Test Programs

The programs described in these listings are explained in Subsection 5.2.1.

C.1 Perms — Permutation Counting

```haskell
main = count (take 4 inf);
count xs = length (perms xs);
append xs ys = case xs of {
    Nil → ys;
    Cons z zs → Cons z (append zs ys)
};
concat xs = case xs of {
    Nil → Nil;
    Cons y ys → append y (concat ys)
};
concatMap f xs = concat (map f xs);
length xs = case xs of {
    Nil → 0;
    Cons y ys → (+) 1 (length ys)
};
insertions x ys = case ys of {
    Nil → Cons (Cons x Nil) Nil;
    Cons z zs → Cons (Cons x (Cons z zs)) (map (Cons z) (insertions x zs))
};
perms xs = case xs of {
    Nil → Cons Nil Nil;
    Cons y ys → concatMap (insertions y) (perms ys)
};
map f xs = case xs of {
    Nil → Nil;
    Cons y ys → Cons (f y) (map f ys)
};
inc x = (+) 1 x;
```
C Test Programs

39 inf = Cons 1 (map inc inf);
40 take n xs = case (==) n o of {
41     True → Nil;
42     False → case xs of { Cons y ys → Cons y (take ((-) n 1) ys) }
43     };
46 }

Listing C.1: Code for the ‘perms’ test program
C.2 Fibonacci — Calculation of Fibonacci Numbers

Listing C.2: Code for the ‘fibonacci’ test program

```haskell
{ values = I (I (I (I (I (I (I (I (I (I (I (I (I (I (I (I (I (I Z))))))))))))))) ))) ))) ))) ))) ))) 

main = takeF values fibs;

fibs = mapF fib infF;

mapF f ds = case ds of {
    Nil    → Nil;
    Cons e es → Cons (f e) (mapF f es)
};

infF = Cons Z (mapF I infF);

takeF gs hs = case gs of {
    Z    → Nil;
    I i  → case hs of {
        Cons j js → Cons j (takeF i js);
    }
};

plus a c = case a of {
    Z    → c;
    I b  → I (plus b c);
};

fib x = case x of {
    Z    → I Z;
    I y  → case y of {
        Z    → I Z;
        I z  → plus (fib z) (fib (I z));
    }
};
```
C Test Programs

C.3 Twizzler — The Twizzler Problem

```haskell
main = count 4;

map f xs = case xs of {
    Nil → Nil;
    Cons y ys → Cons (f y) (map f ys)
};

append xs ys = case xs of {
    Nil → ys;
    Cons z zs → Cons z (append zs ys)
};

concat xs = case xs of {
    Nil → Nil;
    Cons y ys → append y (concat ys)
};

concatMap f xs = concat (map f xs);

length xs = case xs of {
    Nil → 0;
    Cons y ys → (+) 1 (length ys)
};

insertions x ys = case ys of {
    Nil → Cons (Cons x Nil) Nil;
    Cons z zs → Cons (Cons x (Cons z zs)) (map (Cons z) (insertions x zs))
};

perms xs = case xs of {
    Nil → Cons Nil Nil;
    Cons y ys → concatMap (insertions y) (perms ys)
};

take n xs = case (==) n o of {
    True → Nil;
    False → case xs of {Cons y ys → Cons y (take ((-) n 1) ys)}
};

drop n xs = case (==) n o of {
    True → xs;
    False → case xs of {Cons y ys → drop ((-) n 1) ys}
};

head xs = case xs of { Cons y ys → y }

reverse xs = reverseInto xs Nil;
```
reverseInto $xs\ ys = \text{case } xs\ \text{of} \{
    \text{Nil } \rightarrow \ ys;
    \text{Cons } z\ zs \rightarrow \text{reverseInto } zs\ (\text{Cons } z\ ys)\}
\};

twiz $xs = \text{let } \{ n = \text{head } xs \} \ \text{in} \ \text{append} (\text{reverse} (\text{take} n\ xs)) \ (\text{drop}\ n\ xs);

twizzle $xs = \text{case } (==) \ (\text{head } xs)\ 1\ \text{of} \{
    \text{True } \rightarrow \ xs;
    \text{False } \rightarrow \text{twizzle } (\text{twiz } xs)\}\;

filter\ f\ xs = \text{case } xs\ \text{of} \{
    \text{Nil } \rightarrow \text{Nil};
    \text{Cons } y\ ys \rightarrow \text{case } f\ y\ \text{of} \{
        \text{True } \rightarrow \text{Cons } y\ (\text{filter } f\ ys);\n        \text{False } \rightarrow \text{filter } f\ ys;\n    \}\};

unsorted\ xs = \text{case } xs\ \text{of} \{
    \text{Nil } \rightarrow \text{False};
    \text{Cons } y\ ys \rightarrow \text{case } xs\ \text{of} \{
        \text{Nil } \rightarrow \text{False};
        \text{Cons } z\ zs \rightarrow \text{case } (<=)\ y\ z\ \text{of} \{
            \text{True } \rightarrow \text{unsorted } ys;\n            \text{False } \rightarrow \text{True}\n        \}\}\};

inc\ x = (+)\ 1\ x;

inf = \text{Cons } 1\ (\text{map}\ inc\ inf);\n
input\ n = ((\text{take}\ n\ inf))\;

operation\ xs = (\text{map}\ twizzle\ (\text{perms}\ xs));

count\ n = \text{length} (\text{filter}\ \text{unsorted}\ (\text{operation}\ (\text{input}\ n)));
C.4 Word Count — Counting Words in a String

```haskell
{ 
  values = "quick brown fox jumps over the lazy dog";
  main = wc values;
  wc s = length (words s);
  length xs = case xs of {
    Nil → 0;
    Cons y ys → (+) 1 (length ys)
  };
  isSpace x = (==) ' ' x;
  words s = let { x = dropWhile isSpace s } in case x of {
    Nil → Nil;
    Cons t ts → case break isSpace x of {
      Pair w y → Cons w (words y)
    }
  };
  dropWhile f xs = case xs of {
    Nil → Nil;
    Cons y ys → case f y of {
      True → dropWhile f ys;
      False → Cons y ys
    }
  };
  break f xs = case xs of {
    Nil → Pair Nil Nil;
    Cons y ys → case f y of {
      True → Pair Nil xs;
      False → case break f ys of {
        Pair ls rs → Pair (Cons y ls) rs
      }
    }
  }
}
```

Listing C.4: Code for the ‘wc’ test program

C Test Programs
C.5 N-Queens — $N = 10$ Instance

```
C.5 N-Queens — $N = 10$ Instance

```

```haskell

```
C Test Programs

```c
};
next mask = merge (merge (down mask) (left mask)) (right mask);
fill ins = case ins of {
    Nil → Nil;
    Cons x xs → append (lrd x xs) (map (Cons x) (fill xs));
};
lrd ins ys = case ins of {
    Nil → Cons (Cons (Cons l (Cons r (Cons d Nil))) ys) Nil;
    Cons x xs → Nil;
};
solve n mask =
    case (==) n o of {
        True → Cons Nil Nil;
        False → concatMap (sol ((−) n 1)) (fill mask);
    };
sol n row = map (Cons row) (solve n (next row));
nqueens n = length (solve n (replicate n Nil));
main = emitInt (nqueens 10) 0;
```

Listing C.5: Code for the ‘nqueens’ test program
This is the literate output of the Optimus Prime supercompiler for the Word Count test program.
Reachable functions

main

main = wc values

wc

wc s = length (words s)
≡ { Inline 'length', Inline Lets, Remove Lets }
w s = case words s of
  Nil → 0
  Cons y1 ys1 → (+) 1 (length ys1)
≡ { Inline 'words', Substitute Variables, Inline Lets, RemoveLets, RemoveLets, Case of Case, Case of Constructor, RemoveLets, Case of Case, Case of Constructor, Inline Lets, Inline Lets, Remove Lets, Homeomorphically embeds 'wc.0' [2,0,1,0,2], Generalised }
w s = case dropWhile isSpace s of
  Nil → 0
  Cons t1 ts1 → case break isSpace (Cons t1 ts1) of
    Pair y2 y3 → (+) 1 (length (words y2))
≡ { Residuated, ["wc1","wc2"] }
w s = case wc1 s of
  Nil → 0
  Cons t1 ts1 → wc2 t1 ts1
≡ { Final inlining, Inline Lets, Case of a Let, Inline Lets, Remove Lets, Inline Lets, Case of Case, Inline Lets, Remove Lets, Case of Constructor, Inline Lets, Split Lets, Case of Case, Inline Lets, Case of Constructor, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Case of Constructor, Inline Lets, Split Lets, Case of Case, Inline Lets, Case of Constructor, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets }
w s = case wc9 s of
  Nil → 0
  Cons t1 ts1 → case (≡) 32 t1 of
    True → (+) 1 (uc (Cons t1 ts1))
    False → case t1 of
      Nil → (+) 1 (uc Nil)
      Cons y1 y2 y3 y4 y5 → case (≡) 32 y3 of
        True → (+) 1 (uc (Cons y1 y2 y3 y4 y5))
        False → case y3 of
          Nil → (+) 1 (uc Nil)
          Cons y6 y7 → case (≡) 32 y7 of
            True → (+) 1 (uc (Cons y6 y7))
            False → case y7 of
              Nil → (+) 1 (uc Nil)
              Cons y8 y9 y10 y11 y12 → case (≡) 32 y10 of
                True → (+) 1 (uc (Cons y8 y9 y10 y11 y12))
                False → case y10 of
                  Pair ls10 rs12 → (+) 1
values

values = Cons 113 (Cons 117 (Cons 105 (Cons 99 (Cons 107 (Cons 32 (Cons 98 (Cons 114 (Cons 119 (Cons 110 (Cons 32 (Cons 102 (Cons 111 (Cons 120 (Cons 32 (Cons 106 (Cons 117 (Cons 109 (Cons 112 (Cons 115 (Cons 32 (Cons 111 (Cons 118 (Cons 101 (Cons 114 (Cons 32 (Cons 108 (Cons 97 (Cons 122 (Cons 121 (Cons 32 (Cons 100 (Cons 111 (Cons 103 Nil)))))))))))))))))))))))))))))))

wc1

- Residual of wc
- \( wc1 \ s = \text{dropWhile isSpace } s \equiv \{ \text{Inline 'dropWhile', Inline Lets, Inline Lets, Remove Lets, Homeomorphically embeds 'wc1.0' } [0,1,0,2], \text{Generalised} \} \\
- wc1 s = \text{case } s \text{ of} \\
  \text{Nil} \rightarrow \text{Nil} \\
  \text{Cons } y3 \ ys2 \rightarrow \text{case isSpace } y3 \text{ of} \\
  \text{True} \rightarrow \text{dropWhile isSpace } ys2 \\
  \text{False} \rightarrow \text{Cons } y3 \ ys2 \\
  \equiv \{ \text{Residuated ["wc3"] } \} \\
- wc1 s = \text{case } s \text{ of} \\
  \text{Nil} \rightarrow \text{Nil} \\
  \text{Cons } y3 \ ys2 \rightarrow \text{wc3 } y3 \ ys2 \\
  \equiv \{ \text{Final inlining, Inline Lets, Remove Lets, Inline Lets, Inline Lets, Remove Lets} \} \\
- wc1 s = \text{case } s \text{ of} \\
  \text{Nil} \rightarrow \text{Nil} \\
  \text{Cons } y3 \ ys2 \rightarrow \text{case } (\equiv) 32 y3 \text{ of} \\
  \text{True} \rightarrow wc1 y3 \\
  \text{False} \rightarrow \text{Cons } y3 \ ys2 \\

wc9

- Residual of wc8
- \( wc9 \ ys4 = \text{break isSpace } y4 \equiv \{ \text{Inline 'break', Substitute Variables, InlineLets, Inline Lets, Remove Lets, Homeomorphically embeds 'wc9.0' } [0,0,2,0,2], \text{Generalised} \} \\
- wc9 y4 = \text{case } y4 \text{ of} \\
  \text{Nil} \rightarrow \text{Pair Nil Nil} \\
  \text{Cons } y6 \ ys5 \rightarrow \text{case isSpace } y6 \text{ of} \\
  \text{True} \rightarrow \text{Pair Nil (Cons y6 ys5)} \\
  \text{False} \rightarrow \text{case break isSpace y5 \text{ of} } \\
  \text{Pair ls3 rs5} \rightarrow \text{Pair (Cons y6 ls3) rs5} \\
  \equiv \{ \text{Residuated ["wc4"] } \} \\
- wc9 y4 = \text{case } y4 \text{ of} \\
  \text{Nil} \rightarrow \text{Pair Nil Nil} \\
  \text{Cons } y6 \ ys5 \rightarrow wc4 y6 ys5 \\
  \equiv \{ \text{Final inlining, Inline Lets, Remove Lets, InlineLets, Inline Lets, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets} \} \\
- wc9 y4 = \text{case } y4 \text{ of} \\
  \text{Nil} \rightarrow \text{Pair Nil Nil} \\
  \text{Cons } y6 \ ys5 \rightarrow \text{case } (\equiv) 32 y6 \text{ of} \\
  \text{True} \rightarrow \text{Pair Nil (Cons y6 ys5)} \\
  \text{False} \rightarrow \text{case } y6 \text{ of} \\
  \text{Nil} \rightarrow \text{Pair (Cons y6 Nil) Nil} \\
  \text{Cons } y17 \ ys12 \rightarrow \text{case } (\equiv) 32 y17 \text{ of} \\
  \text{True} \rightarrow \text{Pair (Cons y6 Nil) (Cons y17 ys12)} \\
  \text{False} \rightarrow wc9 y12 \text{ of} \\
  \text{Pair ls5 rs7} \rightarrow \text{Pair (Cons y6 (Cons y17 ls5)) rs7}
D Supercompilation Report for Word Count

Obsolete functions

\textbf{wc3}

- Residual of wc1, Homeomorphically embeds \textquote{wc1.0} \([0,1]\), Generalised

\[ wc3 \ y3 \ ys2 = \begin{cases} \text{case } \text{isSpace } y3 \ \text{of} \\ \text{True} & \rightarrow \text{dropWhile isSpace } ys2 \\ \text{False} & \rightarrow \text{Cons } y3 \ ys2 \\ \end{cases} \]

\{ Residuated \[\text{"isSpace"}, \text{\"wc1\"}] \} \]

\[ wc3 \ y3 \ ys2 = \begin{cases} \text{case } \text{isSpace } y3 \ \text{of} \\ \text{True} & \rightarrow \text{wc1 } ys2 \\ \text{False} & \rightarrow \text{Cons } y3 \ ys2 \\ \end{cases} \]

\{ Final inlining, Inline Lets, Remove Lets \}

\[ wc3 \ y3 \ ys2 = \begin{cases} \text{case } (=) \ 32 \ y3 \ \text{of} \\ \text{True} & \rightarrow \text{wc1 } ys2 \\ \text{False} & \rightarrow \text{Cons } y3 \ ys2 \\ \end{cases} \]

\textbf{wc8}

- Residual of wc7, Homeomorphically embeds \textquote{wc6.0} \([0,2]\)

\[ wc8 \ ys4 \ t1 \ y5 = \begin{cases} \text{break isSpace } ys4 \ \text{of} \\ \text{Pair } ls2 \ rs2 & \rightarrow \text{Pair } (\text{Cons } t1 \ (\text{Cons } y5 \ ls2)) \ rs2 \\ \end{cases} \]

\{ Residuated \[\text{\"wc9\"}] \} \]

\[ wc8 \ ys4 \ t1 \ y5 = \begin{cases} \text{wc9 } ys4 \ \text{of} \\ \text{Pair } ls2 \ rs2 & \rightarrow \text{Pair } (\text{Cons } t1 \ (\text{Cons } y5 \ ls2)) \ rs2 \\ \end{cases} \]

\textbf{wc7}

- Residual of wc6, Homeomorphically embeds \textquote{wc6.0} \([0,2]\)

\[ wc7 \ y5 \ t1 \ ys4 = \begin{cases} \text{isSpace } y5 \ \text{of} \\ \text{True} & \rightarrow \text{Pair } (\text{Cons } t1 \ Nil) \ (\text{Cons } y5 \ ys4) \\ \text{False} & \rightarrow \text{wc8 } ys4 \ t1 \ y5 \\ \end{cases} \]

\{ Final inlining, Inline Lets, Remove Lets, Inline Lets, Inline Lets, Inline Lets, Remove Lets \}

\[ wc7 \ y5 \ t1 \ ys4 = \begin{cases} \text{case } (=) \ 32 \ y3 \ \text{of} \\ \text{True} & \rightarrow \text{wc9 } ys4 \ \text{of} \\ \text{Pair } ls6 \ rs8 & \rightarrow \text{Pair } (\text{Cons } t1 \ (\text{Cons } y5 \ ls6)) \ rs8 \\ \end{cases} \]

\textbf{wc6}

- Residual of wc4

\[ wc6 \ ts1 \ t1 = \begin{cases} \text{case } \text{break } \text{isSpace } ts1 \ \text{of} \\ \text{Pair } ls1 \ rs1 & \rightarrow \text{Pair } (\text{Cons } t1 \ ls1) \ rs1 \\ \end{cases} \]

\{ Inline \textquote{\textquoteright break}, Substitute Variables, Inline Lets, Case of a Let, Inline Lets, Remove Lets, Case of Case, Case of Constructor, Inline Lets, Split Lets, Case of Case, Inline Lets, Case of Constructor, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Remove Lets, Homeomorphically embeds \textquote{wc6.0} \([0,2,0,2]\) \}

\[ wc6 \ ts1 \ t1 = \begin{cases} \text{case } ls1 \ \text{of} \\ \text{Nil} & \rightarrow \text{Pair } (\text{Cons } t1 \ Nil) \ Nil \\ \text{Cons } y5 \ ys4 & \rightarrow \text{case } \text{isSpace } ys4 \ \text{of} \\ \text{True} & \rightarrow \text{Pair } (\text{Cons } t1 \ Nil) \ (\text{Cons } y5 \ ys4) \\ \text{False} & \rightarrow \text{case } \text{break } \text{isSpace } ys4 \ \text{of} \\ \text{Pair } ls2 \ rs2 & \rightarrow \text{Pair } (\text{Cons } t1 \ (\text{Cons } y5 \ ls2)) \ rs2 \\ \end{cases} \]

\{ Residuated \[\text{\"wc7\"}] \} \]

\[ wc6 \ ts1 \ t1 = \begin{cases} \text{case } ls1 \ \text{of} \\ \text{Nil} & \rightarrow \text{Pair } (\text{Cons } t1 \ Nil) \ Nil \\ \text{Cons } y5 \ ys4 & \rightarrow \text{case } wc6 \ t1 \ t1 \ ys4 \ \text{of} \\ \text{Pair } ls7 \ rs9 & \rightarrow \text{Pair } (\text{Cons } t1 \ (\text{Cons } y5 \ ls7)) \ rs9 \\ \end{cases} \]
\[ \text{wc4} \]
- Residual of \( \text{wc2} \)

\[ \text{wc4} t_1 t_3 t_1 = \text{break isSpace (Cons t_1 t_3 t_1)} \]

\[ \equiv \{ \text{Inline 'break', Substitute Variables, Inline Lets, Split Lets, Case of Constructor, Inline Lets, Inline Lets, Remove Lets, Inline Lets, Inline Lets, Remove Lets} \} \]

\[ \text{wc4} t_1 t_3 t_1 = \text{case isSpace t_1 t_2 t_3 t_1 of} \]

\[ \text{True} \rightarrow \text{Pair Nil (Cons t_1 t_3 t_1)} \]

\[ \text{False} \rightarrow \text{case break isSpace t_1 t_2 t_3 t_1 of} \]

\[ \text{Pair t_1 t_2 t_3 t_1} \rightarrow \text{Pair (Cons t_1 t_3 t_1) t_1 t_2 t_3} \]

\[ \equiv \{ \text{Inline 'isSpace', Inline Lets, Remove Lets, Homeomorphically embeds 'isSpace.1' [0], Generalised} \} \]

\[ \text{wc4} t_1 t_3 t_1 = \text{case (≡)} t_1 t_2 t_3 t_1 of \]

\[ \text{True} \rightarrow \text{Pair Nil (Cons t_1 t_3 t_1)} \]

\[ \text{False} \rightarrow \text{case break isSpace t_1 t_2 t_3 t_1 of} \]

\[ \text{Pair t_1 t_2 t_3 t_1} \rightarrow \text{Pair (Cons t_1 t_3 t_1) t_1 t_2 t_3} \]

\[ \equiv \{ \text{Residuated ["wc6"]} \} \]

\[ \text{wc4} t_1 t_3 t_1 = \text{case (≡)} t_1 t_2 t_3 t_1 of \]

\[ \text{True} \rightarrow \text{Pair Nil (Cons t_1 t_3 t_1)} \]

\[ \text{False} \rightarrow \text{wc6 t_1 t_3 t_1 of} \]

\[ \equiv \{ \text{Final inlining, Inline Lets, Inline Lets, Remove Lets} \} \]

\[ \text{wc4} t_1 t_3 t_1 = \text{case (≡)} t_1 t_2 t_3 t_1 of \]

\[ \text{True} \rightarrow \text{Pair Nil (Cons t_1 t_3 t_1)} \]

\[ \text{False} \rightarrow \text{wc6 t_1 t_3 t_1 of} \]

\[ \equiv \{ \text{Residuated ["wc6"]} \} \]

\[ \text{wc4} t_1 t_3 t_1 = \text{case (≡)} t_1 t_2 t_3 t_1 of \]

\[ \text{True} \rightarrow \text{Pair Nil (Cons t_1 t_3 t_1)} \]

\[ \text{False} \rightarrow \text{case ts_1 of} \]

\[ \text{Nil} \rightarrow \text{Pair (Nil Cons t_1 Nil Nil)} \]

\[ \text{Cons y22 ys17} \rightarrow \text{case (≡) Cons y22 ys17 of} \]

\[ \text{True} \rightarrow \text{Pair (Cons t_1 Nil Nil) (Cons y22 ys17)} \]

\[ \text{False} \rightarrow \text{case Cons y22 ys17 of} \]

\[ \text{Pair l8 r8 t10} \rightarrow \text{Pair (Cons t_1 (Cons y22 l8 r8)) r10} \]

\[ \text{wc5} \]
- Residual of \( \text{wc2} \), Homeomorphically embeds 'wc.0' [2], Generalised

\[ \text{wc5} y_2 = (+) 1 \text{(length words y2)} \]

\[ \equiv \{ \text{Residuated ["wc"]} \} \]

\[ \text{wc5} y_2 = (+) 1 \text{(wc y2)} \]

\[ \text{wc2} \]
- Residual of \( \text{wc} \), Homeomorphically embeds 'wc.0' [2,0,1], Generalised

\[ \text{wc2} t_1 t_3 t_1 = \text{case break isSpace (Cons t_1 t_3 t_1) of} \]

\[ \text{Pair w1 y2 \rightarrow (+) 1 \text{(length words y2)}} \]

\[ \equiv \{ \text{Residuated ["wc4","wc5"]} \} \]

\[ \text{wc2} t_1 t_3 t_1 = \text{case (≡) t_1 t_2 t_3 t_1 of} \]

\[ \text{True} \rightarrow \text{Pair Nil (Cons t_1 t_3 t_1)} \]

\[ \text{False} \rightarrow \text{wc4 y2 y11 ys10 of} \]

\[ \text{Pair l8 r8 t10} \rightarrow \text{Pair (Cons t_1 (Cons y22 l8 r8)) r10} \]

\[ \text{length} \]

\[ \text{length xs} = \text{case xs of} \]

\[ \text{Nil} \rightarrow 0 \]

\[ \text{Cons y ys \rightarrow (+) 1 \text{(length ys)}} \]
E Supercompiled Word Count

This is the tidy output of the Optimus Prime supercompiler for the Word Count test program.

```latex
\begin{verbatim}
1 \{ 
2   wc1 s = case s of \{ 
3       Nil \rightarrow Nil;
4       Cons y3 ys2 \rightarrow case (==) 32 
5           y3 of \{ 
6           True \rightarrow wc1 ys2;
7           False \rightarrow Cons y3 ys2 
8           \};
9 \};
10 wc9 ys4 = case ys4 of \{ 
11       Nil \rightarrow Pair Nil Nil;
12       Cons y6 ys5 \rightarrow case (==) 32 
13           y6 of \{ 
14           True \rightarrow Pair Nil (Cons y6 ys5);
15           False \rightarrow case ys5 of \{ 
16               Nil \rightarrow Pair (Cons y6 Nil) Nil;
17               Cons y17 ys12 \rightarrow case (==) 32 
18                   y17 of \{ 
19                   True \rightarrow Pair Nil (Cons y17 ys12);
20                   False \rightarrow case wc9 ys12 of \{ 
21                       Pair ls5 rs7 \rightarrow Pair (Cons y6 
22                           (Cons y17 ls5)) rs7 
23                       \};
24 \};
25 \};
29 \};
31 wc s = case wc1 s of \{ 
32       Nil \rightarrow 0;
33       Cons t1 ts1 \rightarrow case (==) 32 
34           t1 of \{ 
35           True \rightarrow (+) 1 (wc (Cons t1 
36               ts1));
37           False \rightarrow case ts1 of \{ 
38               Nil \rightarrow (+) 1 (wc Nil);
39               Cons y23 ys18 \rightarrow case (==) 32 
40                   y23 of \{ 
41                   True \rightarrow (+) 1 (wc (Cons y23 
42                       ys18));
43 \}
\}
\end{verbatim}
```
False → case ys18 of {
    Nil → (+) 1 (wc Nil);
    Cons y24 ys19 → case (==) 32
        y24 of {
            True → (+) 1 (wc (Cons y24
                ys19));
        }
    False → case ys19 of {
        Nil → (+) 1 (wc Nil);
        Cons y25 ys20 → case (==) 32
            y25 of {
                True → (+) 1 (wc (Cons y25
                    ys20));
            }
        False → case wc9 ys20 of {
            Pair ls10 rs12 → (+) 1 (wc
                rs12)
        }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
    }
Bibliography


Bibliography


Bibliography


