Catamorphism-Based Program Transformations for Non-Strict Functional Languages

László Németh

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computing Science at the University of Glasgow

November 2000

© László Németh 2000
Abstract

In functional languages intermediate data structures are often used as glue to connect separate parts of a program together. These intermediate data structures are useful because they allow modularity, but they are also a cause of inefficiency: each element need to be allocated, to be examined, and to be deallocated.

Warm fusion is a program transformation technique which aims to eliminate intermediate data structures. Functions in a program are first transformed into the so called build-cata form, then fused via a one-step rewrite rule, the cata-build rule. In the process of the transformation to build-cata form we attempt to replace explicit recursion with a fixed pattern of recursion (catamorphism).

We analyse in detail the problem of removing — possibly mutually recursive sets of — polynomial datatypes.

We have implemented the warm fusion method in the Glasgow Haskell Compiler, which has allowed practical feedback. One important conclusion is that catamorphisms and fusion in general deserve a more prominent role in the compilation process. We give a detailed measurement of our implementation on a suite of real application programs.
Contents

Abstract ii

1 Introduction 1
  1.1 Contributions ................................................. 2
  1.2 Structure of the thesis ...................................... 3

2 State of the Art in Intermediate Data Structure Removal 5
  2.1 Wadler’s schema deforestation .............................. 7
  2.2 The rules and strategies approach ......................... 9
  2.3 Listlessness and deforestation ............................ 10
  2.4 Cheap deforestation ........................................ 12
  2.5 Supercompilation ............................................ 13
  2.6 Warm Fusion ................................................. 13
  2.7 Warm Fusion (almost) without inlining .................... 14
  2.8 Hylo Fusion .................................................. 15
  2.9 Deforestation for free ..................................... 16
  2.10 Generic program transformation ......................... 16

3 The Theory of Warm Fusion 18
  3.1 Preliminaries ............................................... 19
  3.2 Catamorphisms .............................................. 20
  3.3 Build ......................................................... 25
  3.4 The correctness of buildify ............................... 25
## The Practice of Warm Fusion I: The Basics

4.1 Informal introduction to warm fusion
  4.1.1 Buildify informally
  4.1.2 Catify informally

4.2 Definitions

4.3 Overview of the method
  4.3.1 The preprocessing stage
  4.3.2 First stage of fusion
  4.3.3 Buildify detailed
  4.3.4 Catify detailed
  4.3.5 The second stage
  4.3.6 Cleaning up

4.4 Discussion
  4.4.1 Do catas deserve a special treatment or should they be ordinary Core bindings?
  4.4.2 When should catas, maps and builds be derived?
  4.4.3 When to transform functions to `build-cata` form
  4.4.4 Buildify-catify vs catify-buildify

4.5 First-order fusion
  4.5.1 Deriving maps
  4.5.2 Deriving catas: implementing the cata evaluation rule
  4.5.3 Cata-Core rules
  4.5.4 Buildify
  4.5.5 Catify

## The Practice of Warm Fusion II: Extensions

5.1 Functions with more than one argument
  5.1.1 Avoiding more than one argument
  5.1.2 Higher-order catas
  5.1.3 Buildify
5.1.4 Catify ......................................................... 70
5.1.5 Higher-order fusion ............................................ 73
5.1.6 Static argument transformation ................................. 74
5.2 Mutually recursive datatypes ................................. 74
  5.2.1 Deriving maps .................................................... 76
  5.2.2 Deriving catas .................................................... 78
  5.2.3 New Cata-Core rules ............................................ 79
  5.2.4 Buildify ......................................................... 80
  5.2.5 Catify ......................................................... 81
5.3 The dynamic rewrite system .................................. 84
  5.3.1 The details ....................................................... 84
5.4 Standardising argument ordering .............................. 86
5.5 Two practical issues ........................................... 88
  5.5.1 Separate compilation ........................................... 88
  5.5.2 List comprehensions .......................................... 91

6 Measuring Warm Fusion ....................................... 99
  6.1 Measuring warm fusion ....................................... 99
  6.2 What we want to measure .................................... 100
  6.3 How to measure it? .......................................... 101
  6.4 A detailed example ......................................... 102
  6.5 The benchmarks ............................................. 117
  6.6 A short analysis of the benchmarks ......................... 117
  6.7 Summary ...................................................... 120
    6.7.1 The control run ........................................ 120
    6.7.2 Normalised run ........................................ 121
    6.7.3 Buildify only .......................................... 123
    6.7.4 Catify only ........................................ 124
  6.8 Conclusions ................................................. 129

7 Conclusions and Further Work ................................ 136
7.1 Further Work ......................................................... 137
7.1.1 Automatically deriving code from types ...................... 137
7.1.2 Special abstract machine for fused programs ............... 138
7.1.3 Transparency of transformations ............................. 138
7.1.4 More aggressive inlining .......................................... 139
7.1.5 Fusion for datatypes with embedded functions ............. 139
7.1.6 Fegaras style folds ............................................. 140
7.1.7 Monadic maps, folds and fusion ............................... 140
7.1.8 Warmer fusion .................................................... 141

A The Framework ......................................................... 142
A.1 The compiler (pre-warm fusion) ................................. 142
A.2 The simplifier ...................................................... 144
A.3 The compiler (post-warm fusion) ................................. 146
List of figures

2.1 The idea of program transformation ...................................... 6
2.2 Wadler’s algebraic deforestation rules ............................... 8

4.1 Overview of the fusion transformation ............................... 39
4.2 Rules for the interaction of catamorphisms and Core ........... 55

5.1 Cata-Core rules in the presence of mutually recursive datatypes .... 79
5.2 Semantics of Haskell list comprehensions ........................... 92
5.3 Traditional list comprehension desugaring scheme ............... 93
5.4 Optimal list comprehension desugaring scheme ................... 98

A.1 Glasgow Haskell Compiler passes .................................... 149
A.2 Syntax of the Core language .......................................... 150
List of tables

2.1 Summary of deforestation efforts ........................................... 7

6.1 Programs of the imaginary subset ........................................... 117
6.2 Programs of the spectral subset ............................................ 118
6.3 Programs of the spectral subset: the Hartel Benchmarks ............... 119
6.4 Programs of the real subset .................................................. 119
6.5 Control run: imaginary subset ............................................... 121
6.6 Control run: the Hartel Benchmarks ....................................... 121
6.7 Control run: spectral subset .................................................. 122
6.8 Control run: the real subset .................................................. 123
6.9 Normalise: imaginary subset .................................................. 123
6.10 Normalise: the Hartel Benchmarks ........................................ 124
6.11 Normalise: spectral subset ................................................... 125
6.12 Normalise: the real subset ................................................... 126
6.13 A datatype making buildify too successful .............................. 126
6.14 Buildify only: imaginary subset .......................................... 127
6.15 Buildify only: the Hartel Benchmarks ................................... 127
6.16 Buildify only: spectral subset .............................................. 128
6.17 Buildify only: the real subset .............................................. 129
6.18 Catify only: imaginary subset .............................................. 130
6.19 Catify only: spectral subset ................................................ 131
6.20 Catify only: the Hartel Benchmarks ...................................... 132
6.21 Catify only: the real subset ................................................ 132
List of tables ix

6.22 Buildify, catify and the cata-build rule: imaginary subset .......................... 133
6.23 Buildify, catify and the cata-build rule: spectral subset .............................. 134
6.24 Buildify, catify and the cata-build rule: the Hartel Benchmarks .................. 135
6.25 Buildify, catify and the cata-build rule: the real subset ............................ 135

A.1 Local transformations .................................................................................. 145
Acknowledgements

I would like to thank my supervisor, Simon Peyton Jones for his support, encouragement, and for answering my endless stream of questions about GHC. I would also like to thank my previous supervisor, Phil Wadler who encouraged me to apply to Glasgow.

I spent my second year at the Oregon Graduate Institute (Portland, Oregon). It was a terrible year — both personally and professionally — but this was not entirely their fault. The country, the climate, Rock Creek, and myself also had prominent roles in it. The person I need to thank from this period is John Launchbury who said ‘yes’ to a question which, at that time, seemed very important to me.

As a consequence of my time in the US (and probably the winter months in Glasgow) I developed a taste for living in sunny places. Thanks to Limsoon Wong of Kent Ridge Digital Laboratories, Singapore for giving me the opportunity to work in Singapore for three months and to discover the pleasures of eating a real mango and laksa. For similar reasons (a fine mixture of sunshine and wild night life), I would like to thank Wai Wong of the Hong Kong Baptist University for the six months I worked there. It was a refreshing break from hacking GHC: I was hacking HOL instead.

I would also like to thank my long-suffering officemates, flatmates and all sorts of other people I met for bearing with me during long, dark periods of depression. I know I hurt a lot of them so besides my thanks I would also like to apologise. Special thanks go to Mark Shields, office and flatmate, who actually saved my life. Joy Goodman read the first draft of this thesis and corrected many of my mistakes. Manuel Chakravarty released his haskell.sty for \LaTeX{} just in time to let me typeset the code appearing in this thesis.

The friendship of István Fekete helped me many times. For help of a very different kind, I would like to thank Anikó Szemethy, Ildikó Balogh, Susan (sorry, I forgot your surname), Lori Weedon, Julie Wilson, Tanya Widen, Christina Endemann, Miriam Elze, Kata Tiry, Swee Chua, Laura Thompson, Chen Nee See, Julie Mullaly and a few others whose name escapes me.

I thank my mother and my family, who never fail to love, support or believe in me although I never managed to explain what I was doing during all these years.

This work was supported by the Overseas Students Award Scheme (no.: ORS/96017017), the University of Glasgow, Department of Computing Science grant and several travel grants from OTKA.
Chapter 1

Introduction

When writing a program, especially in a functional language like Haskell, the programmer is faced with the tension between abstraction and efficiency. A program that is easy to understand and maintain often fails to be efficient, while a more efficient solution often compromises clarity. To allow programmers write readable code, while getting reasonable performance, the transformational approach to program development was advocated by Burstall and Darlington [BD77] as early as in 1977, although the basic ideas had been presented in previous papers by the same authors [DB76].

The transformational approach is performed in phases: first an initial, maybe inefficient, but clear and easy to understand program is written. The second phase, possibly divided into subphases, consist of transforming the initial program into a more efficient one. The approach is often adopted in compilers for various programming languages: the first phase is the program supplied by the user — which is considered the specification — the second phase is performed automatically by the compiler.

Program transformations come in two flavours:

- **Non-automatic** transformations, which are either performed on paper, often referred as program derivation, or assisted by the computer, but requires the intervention of the user to select which transformation to perform or to provide new transformations when needed.

- **Automatic** transformations, that can be entirely automated and are suitable to be incorporated into a compiler.

In this thesis we study a completely automatic program transformation method.

One particular cause of inefficiency in functional programs is the presence of intermediate
data structures. Consider the following Haskell program\(^1\) (with suitable definitions for \textit{sum}, \textit{map}, and \textit{upto}) to compute the sum of the squares from 1 to \(n\):

\[
\text{foo} = \text{\textit{sum . map square . upto 1}}
\]

In this case the intermediate list \([1, 2, \ldots, n]\) connects the functions \textit{upto} and \textit{map square}. Another intermediate list \([1, 4, \ldots, n^2]\) connects \textit{map square} with \textit{sum}. If strict evaluation is used, the program requires space proportional to \(n\), since the intermediate lists needs to be completely built. Under lazy evaluation space is not a problem: each list is generated as it is needed and consumers and producers behave as coroutines, but even under lazy evaluation each element requires time to be allocated, to be examined, and to be deallocated.

A somewhat more complex and error prone, but more efficient definition is:

\[
\begin{align*}
\text{foo} &= \text{\textit{bar 1}} \\
\text{\textit{bar l u}} &= \text{\textit{case l \leq u of}} \\
& \quad \text{True} \rightarrow l^2 + \text{\textit{bar}} (l + 1) \ u \\
& \quad \text{False} \rightarrow 0
\end{align*}
\]

Intermediate data structure removal algorithms attempt to automatically turn the former definition to the later one. In general, intermediate data structures can be of any type: trees, booleans and so on. In this thesis, we describe a transformation technique to remove these intermediate data structures from functional programs.

### 1.1 Contributions

This dissertation explores in detail a non-trivial intermediate data structure removal algorithm that allows programmers to use a compositional style of programming in non-strict functional languages without paying a substantial performance penalty. We make the following contributions:

- We present and analyse (Section 4.5) a non-trivial intermediate data structure removal algorithm to be included as part of a production quality functional language compiler.

- We formulate the algorithms in the properly typed framework of the second-order polymorphic lambda calculus and relate the implementation to theory via parametricity proofs.

\(^1\)In the literature, usefulness of deforestation like program transformations are always demonstrated with this simple example.
1.2 STRUCTURE OF THE THESIS

- We extend earlier work by allowing the algorithms to work on non-recursive and mutually recursive data structures (Section 5.2).

- We present a previously unpublished, simple transformation, normalise in Section 5.4, which simplifies several stages of our transformations.

- We apply the same technique of intermediate data structure removal to both recursive and non-recursive types. The techniques are the same in theory, but in the implementation such uniformity is rare.

- We demonstrate how the technique of warm fusion simplifies the compilation process, namely desugaring, by eliminating the need for special, optimal translations for list comprehensions.

- We prove two important properties, confluence and termination of the rewrite system (Section 5.3), of the transformations.

- We demonstrate the usefulness of the transformations by providing detailed, quantitative measurements of improvements on a large set of programs including hand crafted benchmarks and real programs (Chapter 6).

1.2 Structure of the thesis

This thesis is the culmination of work done in two very different communities in computer science: on one hand compiler writers and on the other hand theorists. The language of both of these communities is well established, but unsurprisingly quite different. In a way of helping the theorist or the fellow compiler writer who does not have much knowledge of the Glasgow Haskell Compiler (GHC 3.03), Appendix A provides a brief introduction to the compiler, its passes and defines the internal language of the compiler, the so called Core Language. This contains everything required to read the rest of the thesis and it is assumed to be known to the reader. For those wishing to pursue further study extensive references are provided.

Chapter 2 reviews earlier work on program transformation and puts this thesis into the entire picture. It also introduces some more terminology, which will be trivial to anyone with a reasonable knowledge of compiler technology but may be new to a theorist.

Chapter 3 is for the compiler writer and serves as background material to the theory of intermediate data structure removal. It contains the essential definitions and theorems on which this thesis is built and references to the proofs. It is not an introduction however to category theory or the categorical treatment of datatypes in general.
Chapter 4 is the core of the thesis. It starts off with an informal introduction to the ideas of the two transformations. Then it formally presents — as rewrite rules in the second-order lambda calculus — several transformations and includes many examples to help understanding. It also contains a discussion of fundamental design decisions regarding the implementation.

Chapter 5 is devoted to two important extensions: fusion for higher-order catamorphisms, which extends the techniques in Chapter 4 to functions with more than one, non-static argument, and fusion in the presence of mutually recursive datatypes. Section 5.3 presents the ‘dynamic’ rewrite system which is used in Chapter 4 and the first two sections of the current chapter. Section 5.4 details a surprisingly simple transformation, standardisation of function arguments, which simplifies most of the material presented in this thesis. The rest of the chapter discusses two related issues: separate compilation and optimal compilation of list comprehensions.

In Chapter 6 we provide detailed quantitative measurements of the gains of the implementation of the intermediate data structure removal algorithm.

Finally, Chapter 7 concludes and suggests further work.

Parts of this work have been previously presented in Németh and Peyton Jones [NPJ98].
Chapter 2

State of the Art in Intermediate Data Structure Removal

Program transformation is a technique for program development that can be used both to generate programs from formal specifications and to generate new programs from old ones. In this thesis, we will exclusively be concerned with the later meaning. Generation of new programs from old programs can be completely manual, often referred to as program derivation, or fully automatic which requires no intervention from the user of the program transformation system. In the context of this thesis we will use the term program transformation to denote fully automatic instances only.

Program transformation has been based either on the schemata approach [Coo66, WS72, MFP91, MH95, Jeu95] whereby new programs are derived by instantiating a fixed set of equivalences between program schemata, or the rules and strategies approach [BD77] whereby new programs are derived by sequences of applications of rules that replace program fragments by new, equivalent program fragments. The applications of the rules (such as the unfold, fold, instantiate etc.) are controlled by strategies (such as tupling, loop fusion, abstraction, accumulation etc.) A third largely unexplored approach is program transformation by proof transformation [BC85]. The advantage of the schemata based approach is that once the dictionary of program equivalences is produced, the transformation only requires matching the program, or its fragments, against fixed set of rules in the dictionary. Its disadvantage is that if there is no rule in the dictionary no transformation at all is possible. Producing a good dictionary however is quite hard. As for the rules and strategies approach, its flexibility is a major attraction, but complex strategies may require application of a long sequence of rules, so transformation can be computationally expensive. The transformations discussed in this thesis is a mixture of the two approaches since application of the cata-build rule resembles the schemata approach, while the two transformations
buildify and catify is closer in spirit to the rules and strategies approach.

The motivation for deriving a new program, $P_2$, from an old one, $P_1$, is typically that we want to improve some aspect of $P_1$ while preserving its semantics: we want $\text{Sem}(P_1) = \text{Sem}(P_2)$ for some given semantic function. More precisely, we want the equivalence induced by $\text{Sem}$ to be a congruence with respect to the operations used for building programs.

The fundamental idea of program transformation is depicted in Figure 2.1. Given an initial program $P_0$, which we consider as the specification, we want to derive a program $P_n$, with the same semantic value $V$, that is, $\text{Sem}(P_0) = \text{Sem}(P_n)$ for some given semantic function $\text{Sem}$. This is often done in more than one step, by constructing a sequence $<P_0, \ldots, P_n>$ of programs such that $\text{Sem}(P_i) = \text{Sem}(P_{i+1})$ for $0 \leq i < n$.

A given a cost function $C$ which indicates the space or time requirements of the execution of a program should satisfy the inequation: $C(P_0) \geq C(P_n)$. However, in the course of program transformation we may allow ourselves to perform a transformation step which results in a program $P_i$, for some $i > 0$, such that $C(P_{i-1}) \leq C(P_i)$, that is to locally increase the associated cost, or we may even allow a $P_i$, such that $C(P_0) \leq C(P_i)$, that is $P_i$ is worse than what we started with, because subsequent transformations may lead to a program version whose cost is smaller than the one of $P_0$. We shall see in Chapter 6 that some of our transformations do have this property. Unfortunately, no general theory of program transformations that deals with this situation in a satisfactory way exists.

While preserving semantics and improvement with respect to some cost function are essential requirements of any optimisation, there is often a third one: it must be worth the effort for both the compiler writer and the user [ASU86], meaning that, it must not require excessive amounts of time to implement it and it must be sufficiently efficient to not unduly affect compilation times. This second aspect is quantified in Chapter 6.

A summary of the discussed methods is shown in Table 2.1.
2.1. WADLER’S SCHEMA DEFORESTATION

<table>
<thead>
<tr>
<th>Method</th>
<th>Data types</th>
<th>Language</th>
<th>Condition</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema deforestation [Wad81]</td>
<td>list</td>
<td>higher</td>
<td>syntactic condition: only for expression written in terms of the basic combinators: <em>map, foldl, generate</em></td>
<td>NA</td>
</tr>
<tr>
<td>Listless transformer [Wad86]</td>
<td>list</td>
<td>first</td>
<td>preorder traversal</td>
<td>no</td>
</tr>
<tr>
<td>Deforestation [Wad90]</td>
<td>polynomial</td>
<td>first</td>
<td>functions in treeless form</td>
<td>prototype</td>
</tr>
<tr>
<td>Chin’s extensions [Chi92b]</td>
<td>polynomial</td>
<td>higher</td>
<td>accepts all functions but deforestation does not take place for non-treeless terms. Higher-order functions are removed</td>
<td>??</td>
</tr>
<tr>
<td>Higher order deforestation</td>
<td>polynomial</td>
<td>higher</td>
<td>??</td>
<td>yes, GHC</td>
</tr>
<tr>
<td>Supercompilation [Tur86]</td>
<td>polynomial</td>
<td>higher</td>
<td>??</td>
<td>NA</td>
</tr>
<tr>
<td>Cheap deforestation [GLPJ93]</td>
<td>list</td>
<td>higher</td>
<td>fixed set of functions (from the Standard Prelude)</td>
<td>yes, GHC</td>
</tr>
<tr>
<td>Warm Fusion [LS95] and this</td>
<td>polynomial</td>
<td>higher</td>
<td>syntactic condition</td>
<td>yes, GHC</td>
</tr>
<tr>
<td>thesis</td>
<td></td>
<td></td>
<td></td>
<td>in progress, GHC</td>
</tr>
<tr>
<td>Hylo Fusion [Hu96, OHIT97]</td>
<td>polynomial</td>
<td>higher</td>
<td>structural hylomorphisms</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.1** Summary of deforestation efforts

### 2.1 Wadler’s schema deforestation

Wadler [Wad81] proposed a simple deforestation algorithm by using a small set of combinators. His combinators — transliterated into the more convenient Haskell notation — are known as *map, foldl, and generate*. 
2.1. WADLER’S SCHEMA DEFORESTATION

\[
\begin{align*}
\text{map } f \, (\text{map } g \, xs) &= \text{map } (f \cdot g) \, xs \\
\text{foldl } f \, a \, (\text{map } g \, xs) &= \text{foldl } h \, a \, xs \\
\text{where} \\
  h \, a' \, x &= f \, a' \, (g \, x) \\
\text{map } f \, (\text{generate } p \, g1 \, g2 \, x) &= \text{generate } p \, h \, g2 \, x \\
\text{where} \\
  h \, b' &= f \, (g1 \, b') \\
\text{foldl } f \, a \, (\text{generate } p \, g1 \, g2 \, x) &= h \, a \, x \\
\text{where} \\
  h \, a' \, b' \mid p \, b' &= a' \\
  h \, a' \, b' \mid \text{otherwise} &= h \, (f \, a' \, (g1 \, b')) \, (g2 \, b') 
\end{align*}
\]

Figure 2.2 Wadler’s algebraic deforestation rules

map, or as we will call this function in later chapters, the type functor for the datatype of lists is defined by

\[
\begin{align*}
\text{map} &:: \forall \alpha \beta. (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\text{map } f \, [] &= [] \\
\text{map } f \, (x : xs) &= f \, x : \text{map } f \, xs 
\end{align*}
\]

List consumption is expressed via foldl, which is the catamorphism for the so called snoc lists:

\[
\begin{align*}
\text{foldl} &:: \forall \alpha \beta. (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
\text{foldl } f \, z \, [] &= z \\
\text{foldl } f \, z \, (x : xs) &= \text{foldl } f \, (f \, z \, x) \, xs 
\end{align*}
\]

Finally, generate is used to express list production:

\[
\begin{align*}
\text{generate} &:: \forall \alpha \beta. (\alpha \to \text{Bool}) \to (\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha \to [\beta] \\
\text{generate } p \, f \, g \, n \mid p \, n &= [] \\
\text{generate } p \, f \, g \, n &= f \, n : \text{generate } p \, f \, g \, (g \, n) 
\end{align*}
\]

The corresponding set of algebraic rules is given in Figure 2.2. This deforestation scheme, as given, is limited to lists, but in theory it is relatively easy to extend to any other algebraic datatype. In practice, however, the number of rules will soon become too large to be implementable.

Wadler’s method is clearly subsumed by the topic of this thesis the warm fusion method.
2.2 The rules and strategies approach

The basic idea is very simple [BD77]. Given the program as a set of recursive equations, the transformational rules are as follows:

1. **Definition Rule.** It consists of adding to the current program a set of mutually exclusive, that is two different left-hand sides do no have common instances, and exhaustive, that is for any element in the domain of any function $f$, there is at least one left-hand side that matches, recursive equations. Left-hand sides of the newly introduced equations are assumed to be unique, that is none of the equations is an instance of the left-hand side of any previously defined equation.

2. **Unfolding Rule (aka inlining).** It consists of replacing the occurrence of the left-hand side of an equation with its right-hand side.

3. **Folding Rule.** The inverse of the Unfolding Rule: it consists of replacing an instance of the right-hand side of an equation by the corresponding instance of its left-hand side.

4. **Instantiation Rule.** It consists of the introduction of an instance of an already existing equation.

5. **Where-abstraction Rule.** We replace the equation $f(\ldots) = \ldots e\ldots$ by $f(\ldots) = \ldots z\ldots$ **where** $z = e$, provided that $z$ does not occur in the equation. Under call-by-value this has the advantage that the evaluation of $e$ is performed only once.

6. **Algebraic Replacement Rule.** We derive a new equation by using algebraic properties, for instance to get $f = a + b$ from $f = b + a$ by appealing to the associativity property of $+$. 

It was shown in [Cou90] that any sequence of the application of these six rules preserves partial correctness of the original program, that is if the transformed program terminates it computes the same value. In a non-automatic transformation system a separate proof of termination must be provided, while an automated system should ensure that total correctness is preserved for example by restricting the sequence of the rules. As we noted, the Unfolding Rule is the inverse of the Folding Rule, therefore an infinite sequence of the application of the transformation rules is possible unless we keep track of the entire transformation history and use strategies to control the application of the rules.

Amongst the many strategies proposed, the best known are the Composition Strategy [BD77, PK82, Par90] and its variant Deforestation [Wad90], the Tupling Strategy [BD77, PK82,
2.3. LISTLESSNESS AND DEFORESTATION

Par90, Chi93, CK93, HIT96b, HIT97], and the Generalisation Strategy [BM75, BD77, PS87, Par90].

The Tupling Strategy aims to avoid multiple access to data structures by tupling functions together which traverse the same data structure.

The Generalisation Strategy, as its name suggest can generalise:

- expressions to variables
- functions to functions by implicit definition

The aim of the Composition Strategy is to eliminate intermediate data structures that arise in expressions like \( f (g(x)) \), or equivalently in the compositional style \( f \circ g \), because the value produced by \( g \) is immediately consumed by \( f \). A variant on the composition strategy is Wadler’s deforestation technique [Wad90]. The transformation studied in this thesis shares the goal of the Composition Strategy, but it also uses techniques previously only used in the schemata based approach (Bird-Meertens formalism [Bir86, Bir87, Bir89, Mee86] or Squiggol).

In the following, we will discuss those techniques which had been influential in the development of the method which forms the core of this thesis. A thorough exposition to the rules and strategies approach to the transformation of functional and logic programs can be found in the paper [PP96b] by Pettorossi. An even higher level discussion of future directions in program transformation is given in [PP96a].

2.3 Listlessness and deforestation

Wadler’s early work on listlessness [Wad84, Wad86] is a refinement of the Composition Strategy. His listless transformer converts programs written in a functional language into imperative ‘listless programs’. By defining a listless form in a very restrictive way (it requires a semantic condition, preorder traversal, to be verified) he proves that listless functions must evaluate in constant bounded space.

The definitive work [Wad90], which coined the term deforestation, improves on listlessness in many ways. Firstly, the definition of a treeless form is much simpler, purely syntactical. This eases the work of both the compiler writer and perhaps more importantly the user, because it makes it easy to characterise what sort of expressions will be optimised. Secondly, deforestation applies to all terms composed solely of treeless functions, whereas
the corresponding listless algorithm applies only when the semantic condition can be verified. Thirdly, the treeless transformer is entirely source-to-source, therefore it is easier to make it part of the compilation by transformation [KH89] process. Fourthly, the concept of blazing is introduced to mark terms of certain types (integers, booleans) which need not be removed.

While working in a first-order language Wadler recognised that his transformation need to be extended to accommodate higher-order functions. He proposed non-recursive, higher-order macros which allowed him for example to deforest the function \( \text{map } f \cdot \text{upto} \).

The comparison of listless and treeless forms is somewhat difficult. In some ways, the treeless form is more general (it allows the definition of functions like reflect), but in other ways it is less general (it does not apply to terms which traverse the data structure twice). While the listless transformer guarantees evaluation of the resulting functions in constant bounded space, the treeless transformer may use space bounded by the depth of the tree.

Wadler’s carefully worded deforestation theorems (both pure and blazed) guarantee the transformation can be without loss of efficiency.

There have been various attempts to extend his method, but still major drawbacks remain. Termination of the transformer is proved in [FW89]. All these algorithms need to keep track of all function calls occurred previously, and introduce a definition for a recursive function on detecting a repetition, which corresponds to the Folding Rule of Burstall and Darlington [BD77]. The process of keeping track of function calls and the clever control to avoid infinite unfolding introduces substantial cost and complexity into algorithms which hinders deforestation to be adopted as part of any serious compiler.

Chin [Chi92b, Chi90, Chi94] extended Wadler’s work in many ways. He devised a double blazing technique, which allowed him to use the treeless transformer for all functions, not only the ones in treeless form. Combined with higher-order removal technique [Chi90, Chi92a] his transformer could process a complete higher-order functional language, although the remaining higher-order functions and functions not in treeless form are not deforested.

Marlow [Mar95] extended Wadler’s work into a yet another direction. Instead of using a blazing technique to avoid higher-order functions which can not be deforested by the original treeless transformer he developed a transformer which works in the presence of higher-order functions. He also produced an implementation of his algorithm for the Glasgow Haskell Compiler. Marlow [Mar95] appears to be the first who elaborated on the connection between deforestation and cut elimination contribution is the notion of transparency, i.e. the property of a transformation, which helps the programmer to decide if the transformation applies.
2.4 Cheap deforestation

In order to avoid the problems associated with infinite unfolding (non-termination of the compiler), the paper by Gill, Launchbury and Peyton Jones [GLPJ93] took a less purist approach. The idea came from the Squiggol community, where it had long been recognised that program transformation is hard in the presence of general recursion. Instead, they advocate the use of higher-order functions which follow a fixed pattern of recursion. One of these fixed patterns on lists is known in the Haskell community as the \textit{foldr} list data type

\begin{verbatim}
data [α] = [] | α : [α]
\end{verbatim}

The \textit{foldr} function standardises the consumption of its argument by traversing the argument in a predefined order and replacing the list constructors (:) by \textit{c} and [] by \textit{n}. Many Standard Prelude functions can be written using \textit{foldr}.

To standardise the production of lists, they introduced a function \textit{build} with type and definition:

\begin{verbatim}
build :: (∀β. (α → β → β) → β → [α]) → [α]
build g = g (:) []
\end{verbatim}

and the following property:

\[ \text{foldr} \ c \ n \ (\text{build} \ g) = g \ c \ n \]  

which appears in the literature under the names \textit{foldr/build} rule, \textit{cata-build} rule, and instance of the Acid Rain theorem for catamorphisms at the datatype of lists.

Equation 2.1 is the essence of cheap deforestation: if a list is produced a certain way, by using \textit{build}, and the result is consumed by \textit{foldr} then the intermediate list need not be built, the result can be constructed by passing \textit{foldr} first two argument directly to \textit{g}. Each application of the \textit{foldr}-\textit{build} rule can be seen as a canned application of unfold/simplify/fold in the traditional deforestation framework. Unfortunately, at that time \textit{build} could not be safely exposed to the Haskell programmer since it does not have a Hindley-Milner [Mil78] type. To circumvent this, most functions of the Standard Prelude were redefined using \textit{foldr}.
2.5. SUPERCOMPILATION

and build. The cheap deforestation algorithm then looks for applications of the above form and rewrites them in one-step. In effect, only certain Standard Prelude functions can be deforested.

While the approach seems to be practical, the measurements of Gill’s implementation [Gil96] did not show any performance improvement on real programs.

2.5 Supercompilation

Turchin’s supercompiler [Tur86] can be seen as another automatic instance of the unfold/fold framework. It is a powerful technique, and it can achieve effects of both deforestation – removing intermediate data structures – and partial evaluation. This makes it strictly more powerful than the approach advocated in this thesis. For example, the supercompiler can derive a Knuth-Morris-Pratt style matcher from the naive definition (see [SGJ94]). The drawback is that it is a much more expensive technique and has never been implemented in a practical way.

The supercompiler does not transform programs. Instead, it traces all possible generalised histories of the computation by the original program, and compiles an equivalent program. Reexpressed in the Darlington and Burstall terminology, supercompilation performs driving: unfolding and propagation of information, and generalisation: a form of abstraction which enables folding. Pettorossi [PP96b] calls this form of generalisation Lambda Abstraction.

Supercompilation is compared with deforestation, and two other techniques partial evaluation and generalised partial computation, in [SGJ94] using a simple test program the Knuth-Morris-Pratt matching algorithm.

2.6 Warm Fusion

Warm fusion [LS95] is the starting point of this thesis. It is the culmination of the work done by Fegaras and Sheard [FSS92, SF93, SF94], which is in turn based on the work of Hagino [Hag88] and Malcolm [Mal89] and related to Kieburstz [KL95]. In this school, intermediate structure removal is often called fusion [MFP91, Fok92b] or promotion [Mal89]. The technique also incorporates ideas from cheap deforestation, namely that removing intermediate data structures is implemented by a one-step rewrite rule.

The fusion rule applies to programs in the so called build-cata\(^1\) form, that is data struc-

\(^1\)We say build-cata form because when functions in such form are read from left to right build appears
ture consumption is expressed using an explicit \textit{fold} (or catamorphism), production is expressed using an explicit \textit{build}. It generalises the cheap deforestation work by extending the \texttt{cata-build} rule to arbitrary polynomial datatypes. Later work by Meijer and Hutton [MH95] and Fegaras and Sheard [FS96] extend the basic method to apply to datatypes with embedded functions, i.e. to include the function space constructor. The major contribution of the Launchbury and Sheard paper is, that in order to make the one-step fusion rule apply more often they suggested a completely automated, conservative, decision procedure which became known as the warm fusion method. This attempts to turn functions in general recursive form into \texttt{build-cata} form with an explicit \textit{fold} and \textit{build}, which, when successful, allows the application of the one-step fusion rule, the \texttt{cata-build} rule. When the warm fusion method is not successful fusion is not attempted.

The original work has been extended in various ways. Theory is extended to \textit{monadic folds} by Fokkinga [Fok94] and Meijer and Jeuring [MJ95]. The implementation of the extension to the monadic case is hindered by the fact that monadic folds are sometimes too specific to be useful and there is a side condition on the monad which is known not to hold for several often used monads.

Fegaras continued to develop fusion techniques: first, in [FSZ94], he proposed a new binary promotion theorem, which can successfully fuse on both arguments of \texttt{zip}. Later he abandoned the fixed pattern of recursion idea, \textit{(fold)}, returned to the basics and suggested the direct use of the \textit{parametricity theorem} [Rey83, Wad89] to fusion [Feg96].

\section{Warm Fusion (almost) without inlining}

In an attempt to overcome the difficulties of the warm fusion method which were first reported in [NPJ98] and discussed at length in this thesis, Chtitil [Chi99] uses the type system to predict when the transformation to explicit \texttt{build} form can be successful. More recently [Chi00], he managed to dispense \texttt{build} completely. Unfortunately, as we shall demonstrate in Section 3.3 this does not simplify the implementation of warm fusion, only decreases the penalty we are paying for the transformation to explicit \texttt{build-cata} form (see Page 99 for further details). No complete design, suitable for incorporation into a production quality compiler like GHC, has been put forward so far.

\begin{table}
\end{table}

\begin{itemize}
\item \texttt{cata-build} rule because that describes the application of a \texttt{cata} to a \texttt{build} (also read from left to right).
\end{itemize}
2.8 Hylo Fusion

Warm Fusion is a program transformation based on catamorphisms, which is just one fixed pattern of recursion. Other well-known patterns are anamorphisms [MFP91], paramorphisms [Mee90], mutumorphisms [Fok92b] and hylomorphisms [MFP91, TM95] just to name a few. A method for intermediate data structure removal was suggested by Hu, Iwasaki and Takeichi in [HIT96d] and discussed in detail in Hu’s thesis [Hu96], is based on hylomorphisms (what you get by composing a catamorphism with an anamorphism: \( \langle \varphi, \psi \rangle = (\varphi) \cdot (\psi) \)) and its associated fusion laws. An extension of the basic system is suggested in [HIT96d], progress report on the ongoing effort to implement the system can be found in [OHIT97].

The idea resembles to the idea of the warm fusion method. Functions in general recursive form are first transformed into hylomorphisms in triplet form [TM95], which is more convenient for program transformation. This step is similar to the two transformations, catify and buildify, studied in this thesis.

\[
f :: B \to A
f \Rightarrow \langle \varphi, \eta, \psi \rangle_{G,F}, \text{where } \varphi :: (GA \to A), \eta :: (F \to G), \psi :: (B \to FB) \quad (2.2)
\]

Note, that \( \eta \) is a natural transformation. Unlike catify and buildify, the algorithm in [Hu96] for this transformation always succeeds in the sense that it is always terminates with a hylomorphism. However, the result may not be in the right form for the fusion theorems:

\[
\text{Cata-Hylo Fusion} \quad (2.3)
\tau :: \forall A.(FA \to A) \to FA \to A \Rightarrow \langle \varphi, \eta_1, out F \rangle_{G,F} \cdot \langle \tau in F, \eta_2, \psi \rangle_{F,L} = \langle \tau(\varphi \cdot \eta_1), \eta_2, \psi \rangle_{F,L}
\]

\[
\text{Hylo-Ana Fusion} \quad (2.4)
\sigma :: \forall A.(A \to FA) \to A \to FA \Rightarrow \langle \varphi, \eta_1, \sigma out F \rangle_{G,F} \cdot \langle in F, \eta_2, \psi \rangle_{F,L} = \langle \varphi, \eta_1, \sigma(\eta_2 \cdot \psi) \rangle_{G,F}
\]

Note, that fusion can only take place (in the Cata-Hylo case for example), if the hylomorphism on the left (\( \langle \varphi, \eta_1, out F \rangle_{G,F} \)) is really a catamorphism and the hylomorphism on the right is of a specific form \( \langle \tau in F, \eta_2, \psi \rangle_{F,L} \). Dually, the Hylo-Ana law is only applicable if the hylomorphism on the right is really an anamorphism and the one on the left is of a specific form.

In order to allow more fusion to take place via Equation 2.3 or Equation 2.4, hylomorphisms are restructured using the Hylo-Shift law

\[
\langle \varphi, \eta, \psi \rangle_{G,F} = \langle \varphi \cdot \eta, id, \psi \rangle_{F,F} = \langle \varphi, id, \eta \cdot \psi \rangle_{G,G} \quad (2.5)
\]
and appropriate $\tau, \sigma$ are derived. These last two steps may not find appropriate polymorphic functions, so the fusion transformation based on hylomorphisms can also miss opportunities for fusion.

Hylomorphisms have two fusion laws, the Cata-Hylo Fusion (Equation 2.3) and the Hylo-Ana Fusion (Equation 2.4) law. The problem of reduction ordering arises since the reduction system, with overlapping instances of Cata-Hylo and Hylo-Ana redexes which are sensitive to the reduction order, is clearly not confluent. Takano and Meijer [TM95] give a non-trivial algorithm to achieve the maximum deforestation opportunity, but they do not include a proof of this claim. The additional generality over the warm fusion method comes from two sources:

1. Hylomorphisms have been claimed to be sufficiently general to be used to express most functions of interest, that is they are more general than build-cata form. For the particular case of primitive recursive functions this was proved by Meertens in [Mee90].

2. The second transformation based on the hylo-shift law, which allows restructuring of hylomorphisms, and may expose further opportunities for fusion.

2.9 Deforestation for free

A completely different approach — in a rather different setting — is taken by Johnsson [Joh, Boq99] to remove intermediate data structures. The original aim of Boquist’s thesis is to develop an intermediate language (GRIN) for lazy functional languages which is suitable for program analysis and aggressive code optimisation using mostly control flow analysis and inter-procedural register allocation. His relatively simple (at least to the transformations presented in this thesis) transformations taken together can sometimes achieve effects of removing intermediate data structures. This is particularly interesting as in order to do the same we heavily rely on type information which he does not seem to use. One relatively small drawback of his approach is that it is essentially a whole program analysis, which limits its applicability somewhat.

2.10 Generic program transformation

Based on the observation that the fusion transformation itself a generic program (meta program) whose parameters are the distributivity conditions needed in its application, de Moor and Sittampalam [DMS99] proposed yet another approach to intermediate data structure
They found that the scope of the fusion transformation’s applicability is only marred by the limitations of the matching algorithm used to implement rewriting. The importance of higher-order matching has been studied by Huet and Lang [HL78] previously. Higher-order matching being undecidable in general, the most popular restriction is to second-order matching: this restricts pattern variables to be of base type \((\text{Int}, \text{Bool}, [\text{Int}])\), or functions between base types. Since this restriction is not a convenient one for the transformation of functional programs — in most modern functional languages functions are first-class citizens — the paper presents a new approach, which many believe to be more intuitive for the programmer. Instead of restricting the order of variables, they propose a one-step matching algorithm. This matching algorithm lifts many limitations of the original by Huet and Lang.

De Moor and Sittampalam’s paper [DMS99] also reports on a prototype implementation, the MAG system. MAG takes a program file written in a small subset of Haskell and a theory file, a set of conditional equations, which are the transformation rules and applies the rewrite rules until no more is applicable. It shows all the steps properly annotated, so its output also serves as documentation of the transformation. This addresses a frequently occurring problem, first noted by Marlow in his thesis [Mar95], the problem of transparency. Roughly speaking, the problem is that the outcome of various transformations is not predictable, so the user of an optimising compiler can rarely be sure that the resulting program really is better than the one she started with.

Their approach can be seen as a variation on the topic of this thesis. The theory is the same (most of their transformations are expressed in terms of the fusion law), but while we add the transformation to a compiler they do the transformation with a separate tool. The conditions which are needed to be satisfied for a given transformation to take place are easily identifiable in their system (the theory files), in ours it is hidden in the source code of the compiler itself. It would be interesting to see, if their theory files could automatically be incorporated into a compiler, either by recompiling the compiler every time new theories are added, or by ‘parametrising’ the compiler over theory files.
Chapter 3

The Theory of Warm Fusion

We develop a calculus for lazy functional programming based on recursion schemes associated with datatype definitions. For these operators we derive various algebraic laws that are useful in deriving and manipulating programs. [Meijer, Fokkinga and Paterson: Functional Programming with Bananas . . . [MFP91]]

With an analogy to the bananas paper, we could start by saying that we develop a program transformation based on one specific recursion operator and its associated laws. In order to understand the transformations and to establish that these transformations are indeed correct, we need to recall some theory. Since this thesis makes no contributions to the theory of catamorphisms or category theory this chapter serves purely as background material on the relevant theory and it is based on three sources: Chapter 2 of the book ‘Algebra of Programming’ by Richard Bird and Oege de Moor [BDM97] which is the smoothest introduction to the categorical treatment of datatypes and calculating programs, Fokkinga’s thesis [Fok92b] which is the most detailed introduction and contains a wealth of material, and the bananas paper [MFP91]. The order of definitions follows that of in the ‘Algebra of Programming’, but some notation is incorporated from the bananas paper.

Catamorphisms, and their associated laws, are well known in the literature [Fok92a, MFP91, Fok92b, FM94], therefore, instead of repeating all the categorical setup, theorems and their proofs we only state them. Only those proofs are given which prove the correctness of our transformations. The interested reader is referred to Fokkinga’s papers. The definitive work is Fokkinga’s thesis [Fok92b].
3.1 Preliminaries

In the following we assume that the reader is familiar with the basic notions of category theory: objects, arrows, (small) categories, initiality, products, sums, and functors. For those lacking this knowledge an easy introduction is Pierce’s book [Pie91] or Fokkinga’s Gentle Introduction to Category Theory [Fok92a]. Everything (and more) one ever needs is covered in [Mac71].

Some of the following definitions hold in any category, but we do not need that generality, so in order to avoid confusion when it matters we state that our default category for types is \( \text{CPO} \), the category of complete partial orders with continuous functions. This is a convenient choice to handle arbitrary recursive equations in a framework close to lazy functional programming languages.

**Definition 3.1** Let \( F \) be an endofunctor on a category \( C \). An \( F \)-algebra is an arrow of type \( FA \to A \), the object \( A \) is called the carrier of the algebra.

**Definition 3.2** An \( F \)-homomorphism to an algebra \( f : FA \to A \) from an algebra \( g : FB \to B \) is an arrow \( h : B \to A \) such that \( h \cdot g = f \cdot Fh \).

**Definition 3.3** The objects of the category \( \text{Alg}(F) \) are \( F \)-algebras and the arrows are homomorphisms in between those \( F \)-algebras.

The following class of functors can be used to model datatypes found in functional languages. We exploit this correspondence in Chapter 4, in the definition of polynomial datatypes.

**Definition 3.4** The class of polynomial functors is defined inductively by the following clauses:

- The identity functor \( \text{id} \) and the constant functors \( K_A \) are polynomial;
- If \( F \) and \( G \) are polynomial, then so are their composition \( FG \), their pointwise sum \( F + G \) and their pointwise product \( F \times G \). These pointwise functors are defined by

\[
(F + G)h = Fh + Gh
\]

\[
(F \times G)h = Fh \times Gh
\]
• If $F$ is polynomial, then so is the type functor for $F$

$$Tf = (\text{in} \cdot F(f, id))$$ (3.1)

Type functors only appear in Sections 4.5.1 and 5.2.1 in connection with datatypes like $T\alpha = T_1[T\alpha] \ldots$ where $T_1$ is a data constructor whose argument is of type $[T\alpha]$ (i.e. list of $T\alpha$).

### 3.2 Catamorphisms

Now we have all the definitions to describe one function, and its associated laws on which the rest of the thesis is built upon.

**Theorem 3.1** For polynomial functors, the category $\text{Alg}(F)$ has an initial object and it will be called the initial algebra. It will be denoted in : $FT \rightarrow T$. (The letter $T$ stands for ‘Type’ and also for ‘Term’ since such algebras are often called term algebras).

The proof of this theorem can be found in the book by Manes and Arbib [MA86]. The existence of an initial $F$-algebra means that for any other $F$-algebra $f : FA \rightarrow A$, there is a unique homomorphism from the initial algebra to $f$.

**Definition 3.5 (Catamorphism)** The unique homomorphism from the initial $F$-algebra to another $F$-algebra $f : FA \rightarrow A$ is called a catamorphism. We will denote this homomorphism by $(\text{in} f)$. $(\text{in} f) : T \rightarrow A$ is characterised by the universal property

$$h = (\text{in} f) \equiv h \cdot \text{in} = f \cdot Fh$$

Catamorphisms enjoy many useful properties. From the definition above we immediately obtain the reflection law ($\text{in}$ is called the copy function by Launchbury and Sheard [LS95] and we shall use it in Section 4.5.2)

$$\text{in} = \text{id}$$ (3.2)

The evaluation rule for catamorphisms states how to evaluate an application of $(\text{in} f)$ to an arbitrary element of $F$ (returned by in):

$$(\text{in} f) \cdot \text{in} = f \cdot F(\text{in} f)$$ (3.3)
3.2. CATAMORPHISMS

apply \( (\ell f) \) recursively to the argument of \( \text{in} \) and then apply \( f \) to the result. We shall appeal to this rule in Sections 4.5.2 and 5.2.2.

The induction principle for catamorphisms [Mei92, Page 35]

\[
f \cdot \bot = g \cdot \bot \quad (\forall x, y. f \cdot x = g \cdot y \Rightarrow f \cdot \varphi \cdot Fx = g \cdot \psi \cdot Fy)
\]

follows from the fixed point induction rule

\[
\begin{aligned}
P(\bot) & \quad (\forall a \in A.P(a) \Rightarrow P(f a)) \\
P(\mu f) & \quad (\mu \text{-ind})
\end{aligned}
\]

by \( P(x, y) = f \cdot x = g \cdot y \).

Then there is the very useful fusion law

\[
h \cdot (\ell f) = (\ell g) \iff h \cdot \bot = (\ell g) \cdot \bot \land h \cdot f = g \cdot Fh
\]

The fusion law states that the composition of any function \( h \) with a catamorphism can be reexpressed as a single catamorphism, so that intermediate data structures can be avoided. Operationally, the left-hand side traverses the data structure which \( (\ell f) \) is applied to and builds another one, which is then traversed by \( h \). The right-hand side however combines \( h \) and \( (\ell f) \) into one, and avoids the construction and traversal of the intermediate data structure. Intuitively, the program on the right-hand side is more efficient.

The proof for the general case is by the induction principle for catamorphisms

\[
\begin{aligned}
\langle \text{base case} \rangle \\
h \cdot \bot = i \cdot \bot \\
\langle i = \text{id} \rangle \\
= h \cdot \bot = \bot \\
\langle \text{induction step: assuming } h \cdot x = i \cdot y \rangle \\
= h \cdot f \cdot Fx = i \cdot g \cdot Fy \\
\langle i = \text{id} \rangle \\
= h \cdot f \cdot Fx = g \cdot Fy \\
\langle \text{hypothesis} \rangle \\
= h \cdot f \cdot Fx = g \cdot F(h \cdot x) \\
\langle \text{functor calculus} \rangle \\
= h \cdot f \cdot Fx = g \cdot Fh \cdot Fx
\end{aligned}
\]
\[\langle \text{assume } h \cdot f = g \cdot Fh \rangle = True\]

A more useful variation on the fusion law is to replace the condition \(h \cdot \bot = \langle g \rangle \cdot \bot\) by \(h \cdot \bot = \bot\), i.e. \(h\) is strict.

\[h \cdot \langle f \rangle = \langle g \rangle \iff h \text{ strict} \land h \cdot f = g \cdot Fh\]  \hspace{1cm} (3.5)

We use this form of the fusion law in the transformation catify. See Sections 4.5.5, 5.1.4 and 5.2.5 for details. While the calculational style proof above is perfectly sensible, it is hard to relate to the implementation, because in the proof recursion is made explicit, while in GHC it is not. An alternative proof, based on parametricity, makes the connection between the fusion law (Equation 3.5), catify and the rewrite system much clearer. We prove catify for the special case of lists, by using the free theorem of Wadler [Wad89]. The proof is spelt out in detail, because this shows how the need for the dynamic rewrite system of Section 5.3 arises. The proof for other datatypes is completely analogous.

\[
cata[\cdot] :: \forall \alpha \rho. \rho \to (\alpha \to \rho \to \rho) \to [\alpha] \to \rho
\]

\{ \text{parametricity} \}

\[
(cata[\cdot], cata[\cdot]) \in \forall A X. X \to (A \to X) \to [A] \to X
\]

\{ \forall \text{ on relations twice} \}

\[
= \forall A : A \leftrightarrow A', B : B \leftrightarrow B'.
\]

\[
(cata[\cdot]_A, cata[\cdot]_{A'}) \in B \to (A \to B \to B) \to [A] \to B
\]

\{ \to \text{ three times} \}

\[
= \forall A : A \leftrightarrow A', B : B \leftrightarrow B'.
\]

\[
\forall (n, n') \in B, (c, c') \in (A \to B \to B), (xs, xs') \in [A].
\]

\[
(cata[\cdot]_A n c xs, cata[\cdot]_{A'} n' c' xs') \in B
\]

\{ \forall \text{ on relations twice} \}

\[
\text{Case 1.} \langle \forall (n, n') \in B \rangle
\]

\[
b n = n'
\]

\[
\text{Case 2.} \langle \forall (c, c') \in (A \to B \to B) \rangle
\]

\[
\text{if } a z = z' \land b zs = zs'
\]

\[
\text{then}
\]

\[
b (c z zs) = c' (a z) (b zs)
\]

\[
\text{Case 3.} \langle \forall (xs, xs') \in [A] \rangle
\]

\[
\text{map[\cdot] a xs = xs'}
\]

\{ \text{this gives} \}

\[
= \forall a : A \to A', b : B \to B'.
\]
if \( \forall z : A, zs : B. b \ n = n' \land b(\langle z \rangle zs) = c'(\langle a \rangle (b \ zs)) \land \text{map}\ A \ xs = xs' \)

then

\[
\begin{align*}
b(\text{cata}_A^\Box n c xs) &= \text{cata}_B^{A'} n' c'. \text{map}\ A \ xs \\
\end{align*}
\]

\{ or slightly rewritten, in point-free style \}

\[
\forall a : A \to A', b : B \to B'.
\]

if \( \forall z : A, zs : B. b \ n = n' \land b(\langle z \rangle zs) = c'(\langle a \rangle (b \ zs)) \)

then

\[
\begin{align*}
b. \text{cata}_A^\Box n c &= \text{cata}_A^{A'B'} n' c'. \text{map}\ A \ a \\
\end{align*}
\]

Equation 3.6 has a number of premises. It may appear that in order to satisfy the premises we need some form of automatic theorem proving, but fortunately this is not the case. Instead of proving that the premises hold, we define the unknown variables, \( n' \) and \( c' \), in the right-hand side of Equation 3.6 to satisfy the premises by construction. In particular, if we define \( n' = b \ n \), then Case 1 automatically holds.

For Case 2, we take the conclusion as the implicit definition of \( c' \) and interpret the premises as rewrite rules. That is we define \( c' \) to be \( \lambda z' zs'. b(\langle z \rangle zs) \), simplify and apply the substitutions \( [z := z', b \ zs := zs'] \), which are the premises of Case 2.

To turn an arbitrary function into a catamorphism:

\[
\begin{align*}
\{ &\text{Take } n = [], c = (\cdot), a = \text{id} \} \\
= \forall b : B \to B'. b \ \text{strict} \\
\text{if } \forall z : A, zs : B. b [] = n' \land b(\langle \cdot \rangle z zs) = c' z (b \ zs) \\
\text{then} \\
\begin{align*}
b. \text{cata}_A^\Box [] (\cdot) &= \text{cata}_A^{A'B'} n' c' \\
\end{align*}
\}
\]

\{ Use the conditions as definition of \( n' \) and \( c' \) \}

\[
\begin{align*}
= \forall b : B \to B'. b \ \text{strict} \\
b. \text{cata}_A^\Box [] (\cdot) &= \text{cata}_A^{A'B'} n' c' \\
\text{where} \\
n' &= b [] \\
c' &= \lambda z' zs'. b(\langle \cdot \rangle z zs) [z := z', b \ zs := zs'] \\
\{ \text{cata}\ [] (\cdot) = \text{id} \} \\
= \forall b : B \to B'. b \ \text{strict} \\
b &= \text{cata}_A^{A'B'} n' c' \\
\end{align*}
\]

(3.7)

\{ \text{cata}\ [] (\cdot) = \text{id} \}
In the last clause, \([z := z', b \, zs = zs']\) denotes the substitution of \(z'\) for \(z\) and \(zs'\) for \(b \, zs\) in the body of \(b ((:) z \, zs)\).

Let's see an example in detail! This time we turn the definition of \(\text{map}\) for lists to a catamorphism.

\[
\text{map} = \Lambda \alpha \beta. \lambda xs f. \text{case } xs \text{ of } \]
\[
\quad [] \rightarrow [] \beta
\]
\[
\quad (:) x \, xs \rightarrow (:) \beta (f \, x) (\text{map } \alpha \beta xs f)
\]

(Equation 3.7)

\[
\text{map} = \Lambda \alpha \beta. \lambda xs f. \text{cata } \beta [\beta] \, n' \, c' \, xs \, f
\]

where

\[
\begin{align*}
\quad n' &= \text{map} \, \alpha \beta ([]) \alpha \\
\quad c' &= \lambda z' \, zs'. \text{map} \, \alpha \beta ((:) \alpha z \, zs)
\end{align*}
\]

(simplifies to)

\[
\text{map} = \Lambda \alpha \beta. \lambda xs f. \text{cata } \beta [\beta] \, n' \, c' \, xs \, f
\]

where

\[
\begin{align*}
\quad n' &= \lambda f. [] \beta \\
\quad c' &= \lambda z' \, zs'. (:) \beta (f \, z) (\text{map} \, \alpha \beta zs f)
\end{align*}
\]

(apply the substitutions \([z := z', \text{map} \, \alpha \beta zs := zs']\))

\[
\text{map} = \Lambda \alpha \beta. \lambda xs f. \text{cata } \beta [\beta] \, n' \, c' \, xs \, f
\]

where

\[
\begin{align*}
\quad n' &= \lambda f. [] \beta \\
\quad c' &= \lambda z' \, zs'. (:) \beta (f \, z') (zs' \, f)
\end{align*}
\]

The single most important theorem, which appears under the name \texttt{cata-build} rule in the rest of the thesis, is the Acid Rain theorem.

**Theorem 3.2 (Acid Rain for catamorphism)**

\[
g : \forall A. (FA \rightarrow A) \rightarrow B \rightarrow A \Rightarrow (\varphi)_F \cdot (g \, in_F) = g \, \varphi
\]

Proof by parametricity:

\[
\begin{align*}
\{ \text{ wish } \} \\
= (\varphi)_F (\text{build}^F g) = g \, \varphi \\
\quad \{ \text{ definition: } \text{build}^F g = g \, in_F \} \\
= (\varphi)_F (g \, in_F) = g \, \varphi
\end{align*}
\]
3.3. BUILD

\begin{align*}
\{ \text{the free theorem for } g\text{'s type } \} \\
f (g \psi) = g \varphi \iff f \cdot \psi = \varphi \cdot F f \\
\{ \text{take } f := (\| \varphi \|_F, \psi := in_F } \} \\
= (\| \varphi \|_F (g \text{ in}_F) = g \varphi \iff (\| \varphi \|_F \cdot \text{ in}_F = \varphi \cdot F (\| \varphi \|_F \\
\{ \text{premise trivially holds } \} \\
= \text{True}
\end{align*}

Takano and Meijer [TM95] gives another instance of the Acid Rain theorem (the dual of the one above the so called Acid Rain for anamorphism), but we do not use that theorem in this thesis.

3.3 Build

The function build — for a given datatype $F$ — does not have much theory behind it. It is a syntactic construct which was introduced in Gill, Launchbury and Peyton Jones [GLPJ93]. It serves two purposes: (1) it enforces the side condition on Theorem 3.2 and (2) it eases spotting opportunities for the application of the cata-build rule. Introducing $\text{build}^F g$ for $g \text{ in}_F$ the Acid Rain theorem can be restated as follows (provided the left-hand side is well-typed):

\begin{equation}
(\| \varphi \|_F \cdot (\text{build}^F g) = g \varphi \quad (3.8)
\end{equation}

If the definition of the catamorphism is expanded and $F$ is instantiated at the type of lists one gets Gill’s foldr/build rule (see [Gil96, page 19]):

\begin{equation}
\text{foldr } k z (\text{build } g) = g \ k \ z \quad (3.9)
\end{equation}

3.4 The correctness of buildify

The correctness of buildify (see sections 4.5.4, 5.1.3, and 5.2.4) is equally simple. The need for the worker-wrapper split is explained in the informal introduction to buildify on Page 29.

\begin{align*}
f \\
\{ \text{build introduction splits } f \text{ into two } \} \\
= \text{build}^F f' \\
f' = \lambda \varphi. (\| \varphi \|_F \cdot f
\end{align*}
\[
\{ \text{definition of } f' \} \\
= \text{build}^\mathcal{F} (\lambda \varphi. \| \varphi \|_\mathcal{F} \cdot f) \\
\{ \text{definition of } \text{build} \} \\
= (\lambda \varphi. \| \varphi \|_\mathcal{F} \cdot f)\text{ in}_\mathcal{F} \\
\{ \text{beta reduction} \} \\
= (\| \text{in}_\mathcal{F} \|_\mathcal{F} \cdot f) \\
\{ (\| \text{in}_\mathcal{F} \|_\mathcal{F} = id \} \\
= f
\]
Chapter 4

The Practice of Warm Fusion I: The Basics

Explaining the practice of warm fusion is a daunting task. It’s not that the concepts are hard to grasp, but there is incredible detail: type variables, polymorphic functions passed as arguments to functions, polymorphic functions returned etc. In order to help the reader we first start off with a completely informal introduction, just to show the ideas (Section 4.1). This informal introduction skips many important aspects of the transformation, those are introduced and explained later on. In Section 4.3 we put the ideas introduced in Section 4.1 into a proper framework.

4.1 Informal introduction to warm fusion

For some reason it appears that explaining warm fusion is much easier if one starts at the end of the process, that is at the application of the cata-build rule. This is what we shall do in this section. We are going to be completely informal, shall never use type variables and will only talk about lists. We shall try to answer questions of why instead of how.

In Haskell the type declaration

\[
\text{data List } a = \text{Nil} | \text{Cons } a \text{ (List } a)\
\]

introduces the parametrised type List with two data constructors: the nullary Nil, and Cons with two arguments, the first of which is of type a, that is a parameter, and the second which is of type List a. Notice, that this is the same as the type being declared, so List is in fact a recursive datatype. Examples of values of the type List are:
4.1. INFORMAL INTRODUCTION TO WARM FUSION

\[ \text{Nil} \quad \langle \text{The empty list} \rangle \]
\[ \text{Cons } 42 \text{ Nil} \quad \langle \text{The list containing one element: } 42 \rangle \]
\[ \text{Cons } 42 \text{ (Cons } 69 \text{ Nil)} \quad \langle \text{The list containing two elements: } 42 \text{ and } 69 \rangle \]
\[ \ldots \quad \langle \text{There are many more lists} \rangle \]

An important function which can naturally be associated with this type is called \textit{cata} (from catamorphism). The defining property of \textit{cata} is that when it is applied to a list it uses its arguments (\textit{nil} and \textit{cons}, we usually denote arguments to the cata with the corresponding constructor’s name lowercased and the first argument stands for the first constructor, the second argument for the second and so on) to replace all the constructors in the list. So applying the function \textit{cata} 0 (+), where (+) is the infix addition operator, to the empty list \textit{Nil} results in 0, since the catamorphism replaces \textit{Nil} with the first argument to the \textit{cata}, which is 0. The result of applying the \textit{cata} above to our second example:

\[
\text{cata } 0 (+) \text{ (Cons } 42 \text{ Nil)} \\
\rightarrow (+) 42 \text{ (cata } 0 (+) \text{ Nil)} \\
\rightarrow (+) 42 0 \\
\rightarrow 42
\]

The catamorphism traversed the entire list and replaced \textit{Cons} with the binary addition operator and \textit{Nil} with 0. The result of applying the same function to our third example \textit{Cons} 42 (\textit{Cons} 69 \textit{Nil}) shows that \textit{cata} 0 (+) sums all the elements of the list. The definition of \textit{cata} is:

\[
\text{cata } n \text{ c } \textit{Nil} = n \\
\text{cata } n \text{ c } (\textit{Cons } x \textit{ xs}) = c \ x \ (\text{cata } n \ c \ \textit{xs})
\]

The catamorphism for the datatype of lists is called \textit{foldr} in Haskell, with the minor difference that \textit{n} and \textit{c} are swapped.

Another function which can — not so naturally — be associated with the datatype of lists is called \textit{build}. The defining property of \textit{build} is that its argument, \textit{g}, builds its result only by using the arguments. The definition of \textit{build} is:

\[
\text{build } g = g \textit{ Nil Cons}
\]

It is easy to see what this definition means: build’s argument is a function which takes the constructors — it can of course take an arbitrary number of other arguments as well — of the given datatype, in our case \textit{Nil} and \textit{Cons}. For example,
4.1. INFORMAL INTRODUCTION TO WARM FUSION

\[
\text{map} = \text{build} (\lambda f \, x s \, n \, c. \text{case} \, x s \, \text{of} \\
\quad \text{Nil} \quad \to \quad n \\
\quad \text{Cons} \, x \, x s \to \quad c \,(f \, x) \,(\text{map} \, f \, x s \, n \, c))
\]

is a valid use of \text{build}, while

\[
\text{map} = \text{build} (\lambda f \, x s \, n \, c. \text{case} \, x s \, \text{of} \\
\quad \text{Nil} \quad \to \quad \text{Nil} \\
\quad \text{Cons} \, x \, x s \to \quad \text{Cons} \,(f \, x) \,(\text{map} \, f \, x s \, n \, c))
\]

is not, because \text{build}'s argument does not constructs its result with \(n\) and \(c\). This notion of validity will be formalised later.

Now we have two important functions concerning the datatype of lists, and the only thing we need is a theorem to connect them. This is the \text{cata-build} rule:

\[
\text{cata} \, \text{nil} \, \text{cons} \,(\text{build} \, g) = g \, \text{nil} \, \text{cons}
\]

The theorem says: if a list is built with \text{build} and consumed by a \text{cata} then this 'produce-consume' process can be replaced by a single function \(g\) which does not build the intermediate list. Intuitively, the right-hand side is more efficient, because the intermediate list need not be built, traversed and deallocated.

While it is possible to write programs in \text{build-cata} form it is somewhat tedious. What we need is an automatic way of transforming arbitrary functions into a form where consumption is made explicit by a catamorphism and production of a data structure is made explicit by a build. The transformations to achieve this do in fact exist: the transformation which introduces a build is called \text{buildify}, the other one which introduces a catamorphism is called \text{catify}. In the rest of this section we give an informal introduction how these two transformations can be performed.

4.1.1 Buildify informally

As its name suggest buildify is a transformation which turns functions to an equivalent one with an explicit \text{build} in it. The reason it is called buildify is that the transformation makes it explicit that the function \text{produces} its result in a certain way. Functions which can be transformed are often called \text{good producers}, meaning the presence of the build. We shall explain the transformation with the simplest possible function which builds a list of length \(n\) containing the number 42, where \(n\) is a parameter. One possible definition is:
repAnswer = \lambda n. \textbf{case } n \textbf{ of}
\begin{align*}
0 & \rightarrow \text{ Nil} \\
- & \rightarrow \text{ Cons } 42 (\text{repAnswer} (n - 1))
\end{align*}

One — wrong — way to do this transformation is to simply slap a \textit{build} around the definition of \textit{repAnswer}:

— The two new lambdas are needed because build’s argument must be
— a function which takes the two constructors as arguments

\begin{equation*}
\text{repAnswer} = \text{ build } (\lambda \text{ nil cons.} \lambda n. \textbf{case } n \textbf{ of}
\begin{align*}
0 & \rightarrow \text{ Nil} \\
- & \rightarrow \text{ Cons } 42 (\text{repAnswer} (n - 1)))
\end{align*}
\end{equation*}

When we introduced \textit{build}, we stated that its argument must not use the constructors of the resulting datatype directly: it should use the two arguments \textit{nil} and \textit{cons}\textsuperscript{1}. In other words in the body of \textit{build’s} argument \textit{Nil} and \textit{Cons} need to be replaced by the corresponding \textit{nil} and \textit{cons}. This ’... need to be replaced by the corresponding \textit{nil} and \textit{cons’} should ring the bell for anyone who read Page 28. This is exactly what a \textit{cata} is for! To make the transformation correct, we slap a \textit{cata} around the body of \textit{repAnswer} and get this:

— First correct definition of the transformation

\begin{equation*}
\text{repAnswer} = \text{ build } (\lambda \text{ nil cons n.} \text{cata nil cons (} \textbf{case } n \textbf{ of}
\begin{align*}
0 & \rightarrow \text{ Nil} \\
- & \rightarrow \text{ Cons } 42 (\text{repAnswer} (n - 1)))
\end{align*})
\end{equation*}

This is now a completely sensible and correct transformation, and it can be simplified by noting that \textit{cata} is \textit{strict} i.e. we might as well push it into the right-hand sides of the \textbf{case} alternatives. By doing so we get:

\begin{equation*}
\text{repAnswer} = \text{ build } (\lambda \text{ nil cons n.} \textbf{case } n \textbf{ of}
\begin{align*}
0 & \rightarrow \text{ cata nil con Nil} \\
- & \rightarrow \text{ cata nil cons (} \text{Cons } 42 (\text{repAnswer} (n - 1)))
\end{align*})
\end{equation*}

Using the definition of the catamorphism, the first alternative — \textit{cata} is applied to \textit{Nil} — can be further simplified to \textit{nil}. In the second alternative, the situation is similar: \textit{cata} is applied to \textit{Cons}, which by the definition of catamorphisms can be replaced by \textit{cons} and \textit{cata} applied to the rest of the list. So we get:

\textsuperscript{1}Notice, that \textit{Cons} is the constructor while \textit{cons} is its abstraction. We use the same name, lowercased, to help the reader.
repAnswer = build (\nil \cons n \. case n of
  0 \rightarrow \nil
  \_ \rightarrow \cons 42 (cata \nil \cons (\repAnswer \(n - 1\))))

The only thing which is somewhat worrying is the remaining cata in the second case alternative. The reason it is worrying is that it is the traversal of the rest of the list, which is intuitively unnecessary. What can we do about it? Not much, unless we modify the transformation the following way:

repAnswer = \lambda n \. build (\repAnswer' n)
repAnswer' = \lambda n \nil \cons. cata \nil \cons (\case n of
  0 \rightarrow \nil
  \_ \rightarrow \cons 42 (\repAnswer \(n - 1\))))

This is not too different from the first sensible and correct definition of the transformation (see above). The only difference is that now the cata is moved into another function. This sort of splitting a function into two is often called the worker-wrapper\(^2\) split [PJL91a]. The point of a worker-wrapper split is that by construction the wrapper is small so it can be inlined. It is so small in fact, that the wrapper can be inlined into the worker’s body, which would not be possible otherwise. To see why it does make a difference we note that the cata can be pushed into the case alternatives, where it is applied to the constructors \(\text{Nil}\) and \(\text{Cons}\). This gives:

repAnswer = \lambda n \. build (\repAnswer' n)
repAnswer' = \lambda n \nil \cons. case n of
  0 \rightarrow \nil
  \_ \rightarrow \cons 42 (\repAnswer \(n - 1\))))

What difference the worker-wrapper split makes? The difference is that now the cata is applied to a different function from the one being defined (\(\repAnswer\) instead of \(\repAnswer'\) which is the one being defined). In other words, the right-hand side of \(\repAnswer\), the wrapper, can be inlined into the body of \(\repAnswer'\) and doing so gives (in the process of inlining the definition of \(\repAnswer\) we renamed \(n\) to \(n'\) to avoid a name clash):

repAnswer = \lambda n \. build (\repAnswer' n)

\(^2\)While the terminology is not inappropriate it is getting rather confusing: in the original paper the worker-wrapper split is used to mark strictness properties of functions, therefore allowing subsequent optimisations. In buildify and catify we use it to allow aggressive inlining. In standardising argument ordering (Section 5.4) it is used to allow reordering of arguments.
4.1. INFORMAL INTRODUCTION TO WARM FUSION

\[ \text{repAnswer}', \lambda n \text{ nil cons. case } n \text{ of} \]
\[ \begin{align*}
0 & \rightarrow \text{ nil} \\
- & \rightarrow \text{ cons } 42 (\text{cata nil cons} ((\lambda n'. \text{build (repAnswer}' n')) (n - 1)))
\end{align*} \]

The function \( \lambda n'. \text{build (repAnswer}' n') \) has its argument \((n - 1)\) so this application can be beta-reduced which gives:

\[ \text{repAnswer} = \lambda n. \text{build (repAnswer}' n) \]
\[ \text{repAnswer}', \lambda n \text{ nil cons. case } n \text{ of} \]
\[ \begin{align*}
0 & \rightarrow \text{ nil} \\
- & \rightarrow \text{ cons } 42 (\text{cata nil cons (build (repAnswer}' (n - 1)'))}
\end{align*} \]

The astute reader will notice something exciting about the second \text{case} alternative. A \text{cata} is applied to a \text{build}! The \text{cata-build} rule applies and gives:

\[ \text{repAnswer} = \lambda n. \text{build (repAnswer}' n) \]
\[ \text{repAnswer}', \lambda n \text{ nil cons. case } n \text{ of} \]
\[ \begin{align*}
0 & \rightarrow \text{ nil} \\
- & \rightarrow \text{ cons } 42 (\text{repAnswer}' (n - 1) \text{ nil cons})
\end{align*} \]

We managed to transform a function into another one with a \text{build} in it without paying any penalty for the extra traversal by a remaining \text{cata}. The good thing about the worker-wrapper split is that it allows inlining of the wrapper into other functions thereby exposing applications of the \text{cata-build} rule. The only bad thing about the transformation is that the worker is now a function of three arguments instead of the original one. We shall see in later sections that this is indeed a problem and unfortunately it is quite hard to reverse the transformation.

Sections 4.5, 5.1, and 5.2 are variations on this simple example. Section 4.5 generalises the method above from lists to a large class of (recursive and non-recursive) datatypes while Section 5.2 extends it even further to include sets of mutually recursive datatypes. The correctness of buildify is proved in Section 3.4.

4.1.2 Catify informally

In contrast to buildify, which makes it explicit if a function produces its result in a certain way, catify makes it explicit if a function \text{consumes} its argument in a certain way. Accordingly, successfully transformed functions are often called \text{good consumers}, and to denote this
property we stick a \textit{cata} into the definition of the function. Of course, in this process we have to be careful not to change the meaning of the original function. To demonstrate the techniques which we refine in the rest of the thesis we shall be using the well-known \textit{sum} function:

\[
\text{sum} = \lambda \, \textit{xs}. \quad \text{\textbf{case} } \textit{xs} \text{ of} \\
\quad \text{Nil} \quad \rightarrow \ 0 \\
\quad \text{Cons } \textit{a} \text{ as} \quad \rightarrow \ a + \text{sum as}
\]

The game plan is to somehow change this definition to have a \textit{cata} in it. There are several places one could put a \textit{cata} into the right-hand side of \textit{sum}, but if we recall what a \textit{cata} does we might just find the right place. We already discussed, that a \textit{cata} is a special form of recursion (\textit{structural} for those who cannot wait) and its workings is such that as it traverses the list (the third argument) it replaces the constructors by its first two arguments. All the \textit{Cons} cells are replaced by the second argument, and \textit{Nil} is replaced by the first. In other words, the first argument to \textit{cata} must be equivalent what \textit{sum} does if it finds a \textit{Nil} constructor and the second argument must do the same what \textit{sum} does when it finds a \textit{Cons}. But how do we find out what \textit{sum} does in each case? We partially evaluate it!

\[
\text{sum} = \lambda \, \textit{xs}. \quad \text{\textbf{cata} sumnil sumcons \, \textit{xs}}
\]

\textit{sumnil} is a function which stands for the action of \textit{sum} if it finds a \textit{Nil} constructor, and \textit{sumcons} is also a function which represents the \textit{Cons} case. How do we find the definition of \textit{sumnil} and \textit{sumcons}? We apply \textit{sum} to \textit{Nil} to get \textit{sumnil} and apply \textit{sum} to \textit{Cons} to get \textit{sumcons}:

\[
\text{sumnil} \ = \ \text{sum \, \textit{Nil}} \\
\text{sumcons} \ = \ \text{sum \, (Cons \, t \, ts)}
\]

We assume that \textit{t} and \textit{ts} are fresh, appropriately typed variables. Next, we replace \textit{sum} by its right-hand side (unfolding) and get:

\[
\text{sumnil} \ = \ (\lambda \, \textit{xs}. \quad \text{\textbf{case} } \textit{xs} \text{ of} \\
\quad \text{Nil} \quad \rightarrow \ 0 \\
\quad \text{Cons } \textit{a} \text{ as} \quad \rightarrow \ a + \text{sum as}) \, \textit{Nil}
\]

\[
\text{sumcons} \ = \ (\lambda \, \textit{xs}. \quad \text{\textbf{case} } \textit{xs} \text{ of} \\
\quad \text{Nil} \quad \rightarrow \ 0 \\
\quad \text{Cons } \textit{a} \text{ as} \quad \rightarrow \ a + \text{sum as}) \, (\textit{Cons} \, t \, ts)
\]

Beta-reduction both on \textit{sumnil} and \textit{sumcons} gives:
This exposes further opportunities for simplification because the scrutinee of the case is known i.e. in the first case of \( \text{sum}_\text{Nil} \) the scrutinee is \( \text{Nil} \) therefore it cannot possibly be a \( \text{Cons} \) so we simplify, get rid of the entire case expression and get:

\[
\begin{align*}
\text{sum}_\text{Nil} &= 0 \\
\text{sum}_\text{Cons} &= t + \text{sum ts}
\end{align*}
\]

This is almost perfect. The only problem left is the call to \( \text{sum} \) in the right-hand side of \( \text{sum}_\text{Cons} \). Recall again what a catamorphism does. It does (structural) recursion, so catifying a function means that we replace explicit recursion (calls to the function being transformed, like \( \text{sum} \) in our case) with calls to the appropriate catamorphism. So we need to replace \( \text{sum ts} \) with something which does not mention the function \( \text{sum} \). But unfortunately, there does not seem to be anything to replace \( \text{sum ts} \) with.

Returning to the example, we note that the expression: \( \text{cata sum}_\text{Nil} \text{sum}_\text{Cons} \) is not well-typed. \( \text{sum}_\text{Nil} \) is OK, because \( \text{Nil} \) is nullary so the corresponding \( \text{sum}_\text{Nil} \) is a constant function with no arguments. In order to make \( \text{sum}_\text{Cons} \) well-typed we need to add two lambdas and we get:

\[
\begin{align*}
\text{sum}_\text{Nil} &= 0 \\
\text{sum}_\text{Cons} &= \lambda z zs. t + \text{sum ts}
\end{align*}
\]

If \( z \) and \( zs \) are well-typed this makes the entire expression a function of two arguments, but these two new variables do not occur in the body of \( \text{sum}_\text{Cons} \) and \( t \) and \( ts \) are free. We can solve the two problems (there was nothing to replace \( \text{sum ts} \) with and \( t \) and \( ts \) being free in the body of \( \text{sum}_\text{Cons} \)) if we replace \( t \) with \( z \) and \( \text{sum ts} \) with \( zs \). This may seem to be a somewhat arbitrary choice, but this replacement system happens to obey some very simple rules:

Rule 1 : For nullary constructors nothing needs to be done. There are no arguments to
nullary constructors, therefore there are no new variables.

Rule 2 : For a non-nullary constructor there are two sub-cases:
If the type of the argument \(ts\) is the same as the argument to the original function (i.e. \(\text{List Int}\) in our case) then replace the application of the original function to this argument \((\text{sum } ts)\) by a new, appropriately typed variable \(zs\).

- If the type of the argument \(t\) is different from the argument to the original function (i.e. \(\text{Int}\)) then replace \(t\) by a new, appropriately typed variable \(z\).

To put it even simpler: nuke calls to the function being transformed, when an argument has the same type as the argument to the original function, and replace variables by variables with the same type otherwise. In later sections this process is called the dynamic rewrite system. Its ‘rewrite systemness’ requires no further explanation, but why is it dynamic? With a bit of an abuse of the terminology it is called dynamic because the rewrite rules change from function to function: when \(\text{sum}\) is transformed, the expression \(\text{sum } ts\) is replaced by a new variable, when another function, say, \(f\) is transformed applications of \(f\) to the recursively occurring type are replaced. Section 5.3 formalises the rewrite system and Section 3.2 proves its correctness.

Let's see what happens if we replace variables according to the rules above:

\[
\begin{align*}
\text{sum} &= \lambda x . \text{cata sum}_\text{Nil} \text{ sum}_\text{Cons} x \\
\text{sum}_\text{Nil} &= 0 \\
\text{sum}_\text{Cons} &= \lambda z z . z + zs
\end{align*}
\]

Strangely enough all three functions are closed and there is no explicit recursion (calls to \(\text{sum}\)) so we seem to be done. To see that the result really is equivalent to the original definition of \(\text{sum}\) let's try to reverse what we have just done. We inline the definition of \(\text{cata}\) into the body of \(\text{sum}\):

- We renamed \(x\) in the body of \(\text{cata}\) to avoid a name clash

\[
\begin{align*}
\text{sum} &= \lambda x . (\lambda x\text{. cons } x\text{' of case } x\text{' of} \\
\quad &\quad \text{Nil } \rightarrow \text{ nil} \\
\quad &\quad \text{Cons } a\text{ as } \rightarrow \text{ cons } a (\text{cata sum}_\text{Nil} \text{ sum}_\text{Cons} as)) \\
\text{sum}_\text{Nil} &= 0 \\
\text{sum}_\text{Cons} &= \lambda z z . z + zs
\end{align*}
\]

We can now inline \(\text{sum}_\text{Nil}\) and \(\text{sum}_\text{Cons}\) into the body of \(\text{sum}\) and do three beta-reductions. This gives:

\[
\begin{align*}
\text{sum} &= \lambda x . \text{case } x\text{ of} \\
\quad &\quad \text{Nil } \rightarrow 0 \\
\quad &\quad \text{Cons } a\text{ as } \rightarrow a + (\text{cata sum}_\text{Nil} \text{ sum}_\text{Cons} as)
\end{align*}
\]
Earlier on we made the claim that $\text{sum}$ is equivalent to $\text{cata sum}_{\text{Nil}} \text{sum}_{\text{Cons}}$. If we assume that this is in fact the case, we can replace $\text{sum}$ by $\text{cata sum}_{\text{Nil}} \text{sum}_{\text{Cons}}$ or vice versa. We note that $\text{cata sum}_{\text{Nil}} \text{sum}_{\text{Cons}}$ does occur in the body and replacing it by $\text{sum}$ gives:

$$
\text{sum} = \lambda \text{xs}. \text{case } \text{xs of}
\quad \text{Nil} \rightarrow 0
\quad \text{Cons a as} \rightarrow a + \text{sum as}
$$

And this is equivalent to what we started with! We must have been doing something sensible. In plain words, catify abstracts a fixed pattern of recursion, a $\text{cata}$, out of the function being transformed.

This completes the informal introduction to buildify and catify.

### 4.2 Definitions

First, we need a few definitions. In Haskell, an algebraic datatype declaration introduces a new (possibly mutually recursive) type and constructors over that type and has the form (for the precise syntax and examples see the Haskell Report [PJH99]):

$$
\text{data } cx \Rightarrow T_i \text{tv}_{1} \ldots \text{tv}_{m} = K_{11} \text{ty}_{11} \ldots \text{ty}_{1k} | \ldots | K_{1n} \text{ty}_{n1} \ldots \text{ty}_{nk}
$$

where $cx$ is a context. Contexts play no role in this thesis, therefore their effect on the types of constructors will be omitted. We assume that the declarations are dependency analysed, so the index $i$ is greater then one only if, the group is genuinely mutually recursive.

The declaration introduces a new type constructor $T_i$ with data constructors $K_{i1}, \ldots, K_{in}$ whose types are given by:

$$
K_{ij} :: \forall \text{tv}_{1} \ldots \text{tv}_{m}, \text{ty}_{i1} \rightarrow \cdots \rightarrow \text{ty}_{ik} \rightarrow (T_i \text{tv}_{1} \ldots \text{tv}_{m})
$$

Polynomial datatypes are properly defined in Definition 4.1, here we give a purely syntactic definition.
4.3. OVERVIEW OF THE METHOD

Definition 4.1 (Polynomial datatype) A polynomial datatype is one that is built up according to the syntax given in Equation 4.1 and neither the function space constructor \((\to)\) nor quantifiers \(\forall\) appear in \(t_{i_1}, \ldots, t_{i_k}\) for all \(i, k\).

An example of non-polynomial datatype is:

\[
\text{data } T \alpha \beta = T1 (\alpha \to \beta) | \ldots
\]

because of the function space constructor in \(T1 (\alpha \to \beta)\).

Definition 4.2 (Regular datatype) A regular datatype is one in which the recursive uses of the type datatype being defined \((T\) above) have the same arguments, \(tv_1, \ldots, tv_m\), in the same order as the head of the definition.

Most of the usual datatypes (List, Tree, Maybe etc) in Haskell are regular. An example of a non-regular datatype is:

\[
\text{data } \text{Twist} \alpha \beta = \text{Twist} \alpha (\text{Twist} \beta \alpha) | \ldots
\]

because the order of type arguments in the head \((\alpha \beta)\) is different from the recursive use \(\text{Twist} \beta \alpha\).

\[
\text{data } \text{Nest} \alpha = N1 (\text{Nest} [\alpha]) | \ldots
\]

is also non-regular, because in the recursive use of the datatype being defined (the first argument \(\text{Nest} [\alpha]\) to the constructor \(N1\)) \(\text{Nest}\)’s argument is \([\alpha]\), while in the head of the definition is \(\alpha\). Bird [BM98] calls these datatypes nested datatypes.

4.3 Overview of the method

The design is centred around the idea of two stage fusion [LS95]. In the first stage, individual function definitions are preprocessed in an attempt to re-express their definitions in terms of a build and a catamorphism. In the second, invocations of the already transformed functions are fused using the one-step cata-build rule. In practice, there is third, preparatory stage: builds, maps, and catamorphisms are derived for each fusible datatype and every function which is a candidate for fusion has its arguments rearranged to simplify the first stage of fusion. We shall also see that, the transformation is not as beneficial as one might expect so we shall introduce some post-processing to reduce the overhead, which is the result of
the fusion transformation. The different stages and their ingredients are summarised in Figure 4.1.

This separation into two steps is not only for clarity. It is well known that the unfold-fold strategy (the classical Darlington/Burstall approach) of efficiency increasing transformations suffers from two major problems: one is that the fold step may lead to non-terminating recursion, the other that uncontrolled unfolding requires the later stages to search for arbitrary patterns of recursive calls. The two stage approach overcomes the difficulties with the second problem, because the fusion engine is limited to the body of one function, the one being processed. Inter-function fusion happens via the \texttt{cata-build} rule with the help of inlining wrappers. Neither the wrappers nor the \texttt{cata-build} rule are recursive, therefore nontermination becomes a non-issue.

Even though the fusion transformation is separated into two stages, in reality there is quite a bit of interplay between them. During the transformations in the second stage we often need to inline the wrappers of \textit{already transformed} functions to allow for more fusion.

4.3.1 The preprocessing stage

The \textit{preprocessing stage} comprises four steps. In the first, we derive maps — or type functors — for every parametrised, fusible datatype, from the datatype declarations. By deriving, we mean that given the datatype declaration we generate the corresponding code, which amounts to standard polytypic programming as provided by PolyP [JJ97]. The existence of these type functors is established in Equation 3.1. The definition of fusibility and the technicalities of how to derive maps are detailed in Sections 4.5.1 and 5.2.1.

Once we have maps, we can derive catamorphisms for fusible datatypes. Just as in deriving maps, our input consists of datatype declarations and our output is the corresponding code. Similarly to the case of deriving maps, this correspondence is based on the uniqueness property of catamorphisms (Definition 3.5). We need maps first, since catamorphisms which belong to datatypes involving other fusible datatypes involve their maps. We shall see an example of this shortly in Section 4.5.2.

Deriving builds is much simpler than deriving map or cata, because builds are not recursive and have a simple definition.

The need for the last step in the preprocessing stage, normalise, will only arise in the section dealing with the higher-order case, but its purpose is to rearrange the arguments of functions which are candidates for fusion. After the normalisation step every function’s first argument will be of a fusible datatype (provided of course that it originally
4.3. OVERVIEW OF THE METHOD

had any fusible argument) and one in which the function is strict. The newly derived map functions are also put through this transformation. The map for list for example will be changed to have type \( \text{map}^\dagger :: \forall \alpha \beta.[\alpha] \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta] \) as opposed to the usual \( \text{map}^\dagger :: \forall \alpha \beta.(\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta]). \)

4.3.2 First stage of fusion

It is a bit unfair to call the next stage, the first stage, since this is the very heart of the fusion transformation. This is when we automatically transform arbitrary recursive functions into explicit build-cata form, therefore paving the way to the second stage when the one-step fusion rule becomes applicable. We nicknamed the first transformation, which attempts to transform good producers of fusible datatypes to explicit build form, buildify. The second transformation, whose purpose is to transform good consumers of fusible datatypes into
explicit catamorphic form is named catify. We shall use these nicknames frequently in the rest of the thesis, as they are short and easy to remember. Without the first stage, there would be no catas and builds in our programs, unless as in the shortcut deforestation work [GLPJ93], the libraries were rewritten in terms of catas and builds, which limited the applicability of fusion for functions defined in the Prelude, and more importantly, it limited fusion to the only recursive datatype, lists, which is defined in the Prelude. Alternatively, forcing users to write their programs entirely in terms of catas, as in the programming language Charity [CF91], is an idea which never really caught on.

The transformations buildify and catify can both fail. Theoretically it is easy to see why: catamorphisms correspond to structural recursion, so it is not surprising that not every function can be transformed into this restrictive form. In practice, therefore, after both transformations we need to verify that the result is

1. equivalent to the original definition, and
2. the transformation is beneficial.

In the case of buildify, (1) trivially holds (just inline the worker back to the wrapper and we get back what we started with), but (2) needs to be checked: this is the case of the 'radioactive cata'. For catify (1) is important because during the transformation we temporarily produce ill-typed code. We shall say more about this in Sections 4.5.5, 5.1.4, and 5.2.5.

We shall identify this verification with a simple syntactic criteria, one for buildify and another for catify. It should be clear that these syntactic criteria cannot be both complete and sound at the same time, since if they were, we could solve the halting problem: we would attempt to transform the given function and if transformation is successful we could conclude that the function terminates (since functions defined by structural recursion always do). Completeness means that every function which can be written in structural recursive form will pass the criteria, while soundness means that only those functions which are really structurally recursive will pass. The bigger concern is of course the issue of soundness, which must be met. We have no direct proof of this property, but experience with the implementation shows that every single program we have tried so far has the same denotational behaviour with and without the transformations.

Details of these syntactic criteria will be given when we present the transformations: in Section 4.5 for the simplest scenario, in Section 5.1 for the higher order case, and finally in Section 5.2 when we extend the algorithm for mutually recursive datatypes.
4.3. OVERVIEW OF THE METHOD

4.3.3 Buildify detailed

The above discussed possibility of failure gives rise to the following three step approach to buildify.

1. Transform
2. Simplify
3. If the syntactic criterion holds replace the definition of the function with the newly simplified one. Otherwise, keep the original and give up on the possibility of fusion for this function.

The precise definition of the transformation step (Step 1), which is the application of a one-step rewrite rule, is given in Sections 4.5.4, 5.1.3, and 5.2.4; we only discuss the general idea behind it here.

The purpose of the build introduction is to expose that the given function is a good producer of some fusible datatype. build’s argument, $g$, is a (polymorphic) function, which builds its result using only the last arguments to $g$, which stand for the abstracted constructors of the result datatype. Introducing build the following way:

\[
\begin{align*}
(Pseudo \ code) \\
\lambda \bar{v}.e = \Rightarrow \\
\lambda \bar{v}.\text{build}(\lambda c_1 \ldots c_n.e)
\end{align*}
\]

does not suffice, because it does not guarantee that $e$ uses $c_1 \ldots c_n$ exclusively to construct its result. The observation that a catamorphism $\text{cata}^{T} c_1 \ldots c_n$ traverses its argument, and replaces the constructors by $c_1 \ldots c_n$ leads us to use the appropriate catamorphism to abstract the constructors out of $e$:

\[
\begin{align*}
(Pseudo \ code) \\
\lambda \bar{v}.e = \Rightarrow \\
\lambda \bar{v}.\text{build}(\lambda c_1 \ldots c_n.\text{cata} c_1 \ldots c_n.e)
\end{align*}
\]

For example, in the case of lists, build has type $(\forall \alpha. \rho.\rho \rightarrow (\alpha \rightarrow \rho \rightarrow \rho) \rightarrow \rho) \rightarrow [\alpha]$. By using the parametricity theorem [Rey83, Wad89], one can show that if $g$ has the given type,
it must work for any $\rho$. The intuitive explanation of this result is that $g$ is provided with no other operations of type $\rho$ than its two arguments and all it can do is use these arguments to construct its result. The reason for the strange worker-wrapper split is explained on Page 31.

build is not strictly necessary. It only serves as a syntactic construct to help the compiler spotting an opportunity for fusion. All we need to know to apply the \texttt{cata\text{-}build} rule is that a catamorphism is applied to an appropriately typed function. For example, in the case of lists:

$$\texttt{cata}[\alpha] \, \rho \, n \, c \, g \Rightarrow g \, \rho \, n \, c,$$

Of course, if we dispense build it would be somewhat meaningless to call Equation (4.2) the \texttt{cata\text{-}build} rule! Meijer [TM95] calls the equation Acid Rain theorem for catamorphisms.

The aim of the extra simplification step (Step 2) is to ease checking the syntactic criterion of Step 3: the examples later in this chapter will demonstrate that the Core Simplifier will simplify $f$ to some\textsuperscript{3} normal form.

### 4.3.4 Catify detailed

Catify is even more complicated, because of GHC’s limited rewriting capabilities. It requires a four step approach:

1. Transform
2. Simplify
3. Rewrite
4. According to the syntactic criteria replace the definition with the result of the rewrite step, or keep the original and give up on fusion.

Details of the first step are spelt out in Sections 4.5.5, 5.1.4, and 5.2.5. It is also the application of a one-step rewrite rule. The purpose of the transformation is to expose that the successfully transformed function is a \textit{good consumer}: it consumes its argument in a disciplined manner i.e. with a fixed pattern of recursion.

\textsuperscript{3}Precise definition is hindered by the fact that GHC’s rewrite engine is neither confluent, nor terminating. The simplifier is allowed to run a fixed number of times.
The transformation implements the *cata fusion* theorem [MFP91, Fok92b] (aka. promotion theorem of Malcolm [Mal89, Mal90]), which can be used to transform the composition of a *strict* function, \( f \), with a catamorphism into a single catamorphism. We compose \( f \) with the identity catamorphism — one which replaces the constructors of the given datatype with themselves — so its meaning, and termination properties, do not change. The strictness criteria is important, otherwise we may transform a terminating function into a non-terminating one.

Another view of the transformation is that we separate the action of \( f \) into \( n \) cases, one case for each constructor the argument’s datatype has. We do this by partially evaluating \( f \) with respect to its fusible argument.

Step 2, the extra simplification, again, has the purpose of easing the task of the third and fourth steps.

The astute reader will notice from the detailed rewrite rules (Section 5.3), that Step 1 produces *invalid* Core expressions. In GHC, top level Core expressions must be closed, but the rewrite rule introduces well-typed but free variables (we usually denote them by adding a \( t \) in front of the name of the variable they are introduced for). It also introduces extra binders (usually denoted by prefixing with a \( z \)) which are not used in the body. The purpose of Step 3, the rewrite step, is to replace combinations of the function being transformed and the free variables with the extra binders, which, if successful, makes the bindings valid again. Rules of this rewriting are *valid only* in the body of the current function and they are generated on the fly. We are forced to do it this way, because GHC’s rewriting capabilities, with respect to the generated rules, are limited. On the positive side, this separation of the second rewriting from the Core Simplifier allows us to prove termination and confluence of the former.

Buildify and catify are performed on a *per function basis*, i.e. one function at a time, because of the multi-step approach to these transformations. It would be desirable to do the entire program at once, because that is the way GHC is designed. However, the three(four)-step approach makes it nearly impossible to revert to the original definitions of functions in case of failure, because inlining may happen during the simplification. Also, very precise control (for example we would need to be able to instruct the Core Simplifier to simplify some bindings, and not to allow inlining to take place in the first pass, but to allow it in later passes) over inlining would be required and that is another thing GHC lacks.

As we mentioned earlier, there is an interplay between the first stage and the second. The wrappers of already transformed functions are sometimes required for the success of buildify and catify (for a detailed example of this in the case of the *append* function see page 65),
so these two transformations take place in an environment which holds the wrappers.

4.3.5 The second stage

The second stage is very simple as we do not need to do tricky transformations. We only let the Core Simplifier do its job. However, the Core Simplifier needs to be slightly extended: for example it needs to know about the cata-build and the handful of rules are given under the title of Cata-Core rules in the three main sections. Further care is required with regards to inlining. The first step of both buildify and catify is such that it splits functions into wrappers and workers [PJL91a]. The build and the cata functions are put into the wrappers. By construction wrappers are small\(^4\) and the preceding transformations mark them to encourage GHC to inline their definition whenever possible. Once they are inlined, the hope is that they expose opportunities for the cata-build rule. Every application of the cata-build rule eliminates an intermediate data structure and this is what we are aiming for.

4.3.6 Cleaning up

The post-processing stage is necessitated by the fact that the presence of builds result in an overhead which degrades performance badly. Once all the cata-build reductions take place, build is only an unnecessary level of abstraction: an extra function call and some extra arguments. By inlining build we hope to reduce the overhead. After this cleanup, we need one more pass of the Core Simplifier.

4.4 Discussion

This section contains a discussion of some fundamental questions about the implementation of warm fusion in GHC. As such, it is very compiler specific and it is probably of interest of compiler writers only. It also assumes that the ideas of warm fusion is well-understood so reading later parts of the thesis may be necessary.

The bits which are of any consequence later on marked Decision and denote the answer to the question discussed beforehand.

The Haskell compiler is a large piece of software. Being probably the largest application

\[^4\text{The exact definition changes with every release of GHC, but it essentially means, that the function is not recursive, by inlining it we do not risk duplicating computations, or if we do they are not expensive etc. For details of the inlining dilemma see for example [PJM99]}\]
written in Haskell so far, its complexity gives rise to the possibility of doing certain things more than one way. Different solutions often represent different trade-offs: for example simplicity for the compiler writer versus compilation time. Frequently, there is more than one design decision which shapes the entire compiler. Good decisions interact smoothly with the already built parts and with other decisions, others may require rewriting large pieces but in the end may lead to a better overall design. Unfortunately, these design decisions are rarely documented: they are only of interest to other compiler writers and most importantly they are intricate little details and require an in-depth knowledge of the entire compiler, or more precisely the philosophy behind the compiler.

Before we embark on the details of our design, we would like to discuss the overall picture and several decisions we needed to make. We discuss the different options, their advantages and disadvantages and try to justify why we made the choice that we did. In most cases, the decision is influenced by the existing infrastructure within GHC. Future implementors of the fusion transformation may well reach different conclusions for another compiler or later releases of GHC. This section, therefore, is mostly of interest to compiler writers and can be read before the rest of the chapter in strict sequential order or can be skipped on a first reading. In either case, it assumes a solid knowledge of the different passes of the compiler and what they do. Those who are not familiar with this will find an introduction in Appendix A.

4.4.1 Do catas deserve a special treatment or should they be ordinary Core bindings?

By the introduction of catamorphisms into programs – to allow transformation of functions to explicit catamorphic form – we are introducing a new construct into the compilation process. Two alternatives arise:

1. The new catas are introduced as ordinary Core bindings. This has the advantage that the runtime system need not be modified (only the Core Simplifier), but it makes life harder for the compiler writer since the new construct interacts with existing Core constructs, requiring it to be handled specially. We devote Section 4.5.3 to the discussion of how catas and other Core constructs interact and what modifications are required to the Simplifier.

2. Let the runtime system deal with the construct. Introduce cata as a new primitive in Core and propagate this information all the way to the runtime system. This has the huge disadvantage that all the passes have to be modified to accommodate
4.4. DISCUSSION

the new primitive Core construct. The motivation is that catamorphisms represent structural recursion – which can be implemented in a tail recursive manner, requiring only constant bounded space. If we could devise an improved STG [PJ92] machine or a better runtime system which exploits this information it may lead to a big performance benefit. Current trends in compiler construction suggest that the propagation of more information (e.g. type information [MWCG97, TMC+96]) to later stages of the compilation process and to runtime can be exploited.

Of the two alternatives 2 requires a ‘vertical’ change in the compiler, since if cata is a primitive Core construct then every pass which acts on Core needs to be modified. If it is also a primitive STG construct, then the STG machine and the runtime system also needs to be modified. Option 1 requires a change only in the simplifier, therefore it is vastly preferable. At the time of writing, no abstract machine, or runtime system extensions are known, which would exploit the additional information. It is also unknown, how much performance this modification would gain.

Decision: Based on the above, we chose 1, that is catamorphisms will be ordinary Core bindings.

4.4.2 When should catas, maps and builds be derived?

Looking at the overall structure of GHC (see Page 149) one can ask two questions which will lead to constraints on the placement for the derivation pass: what is the last phase when catas and builds need not be present and what is the first phase when these functions can be derived. The answer to the second question is simple: nothing can be done before the Reader and it is desirable to introduce the generated bindings before the Renamer, which will make sure that the new identifiers will be unique. Unfortunately, there is no type information before the Typechecker.

Regarding the first question, it should be absolutely clear that once the Simplifier is run, these bindings must be present: unless special care is taken, Core Lint will complain about non-existent, but referenced identifiers. Even if that special care was taken, deriving catas and maps before the Simplifier seems a more attractive option: the newly derived bindings would go through the same process of simplification as ordinary bindings. One situation in which this matters is the interaction of the new bindings with the full laziness transformation [PJL91b]: if we are not careful during the derivation of catamorphisms and maps we may, by accident, generate code which is not fully lazy, i.e. it repeats computations.

This leaves us with four options, which we will discuss in turn:
1. **Introduce the bindings after the Reader.** Very good candidate, because the newly introduced identifiers are guaranteed to be unique, and will be type checked. Since we are before the Desugaring phase we can generate Haskell source, just as if the user wrote the code. This also has the advantage that the user can refer to these derived functions. Another possible advantage is that we could make use of overloading to smoothly integrate the newly generated functions with user written code. The disadvantage is that we have no type information.

Generating Haskell source is somewhat tricky, perhaps generating some subset of Haskell is the solution.

2. **Introduce the bindings after the Renamer.** We lost the opportunity for automatically (by the compiler) ensuring the uniqueness of the new bindings but there is still no type information.

3. **Introduce the bindings after the Type checker.** Full type information is available and we know that the entire source is well-typed. We still can generate Haskell source, but now we need to give the precise type of every new identifier we generate. This is rather painful.

4. **Introduce the bindings after the Desugarer.** We have to generate Core, with full type information. Getting the types right is cumbersome, but we could possibly generate bindings which would not type check as Haskell source (e.g. functions involving polymorphic recursion). Newer versions of GHC [PJH99] allow polymorphic recursion in the source, — if an explicit type signature is given — which decreases the attractiveness of this route.

Options 2 and 3 are not too different, they don’t buy us much. So, the real candidates are 1 and 4. 1 is very attractive especially if the method can be made to work smoothly with the class mechanism and overloading can be used. This would lead to a limited form of polytypism: the same name, map, could be used with very different types. Unfortunately, the discovery of this option came at a late stage of the (re)design, well after the first implementation was ready which left us very little time to explore this idea thoroughly. In the context of new developments in the theory of fusion [BM98], 4 is still favourable as it allows more control over the type of generated identifiers.

**Decision:** Based on the above, the decision is that we will introduce catas and maps in Core (after the Desugarer).
4.4.3 When to transform functions to build-cata form

It is not unexpected that the transformation to explicit build-cata form interacts with other transformations in GHC, therefore we need to make sure that this interaction does not counteract with other optimisations. There are two principal issues:

- *Transformation to build-cata form vs full laziness.*
  
  Gill [Gil96] already observed that, in most cases, sharing is preferable to deforestation assuming that computing elements of an intermediate data structure is more expensive compared to building the data structure.

- *Transformation to build-cata form vs strictness analysis.*
  
  We would like to run strictness analysis after the transformation to build-cata form. This is because buildify and catify splits functions into workers and wrappers and the strictness properties of these newly generated functions needs to be determined to expose further transformations. By construction our workers are always strict in their first inductive argument and this may help the strictness analyser to do a better job.

**Decision:** Based on these two criteria the transformation to build-cata form is run after full laziness but before strictness analysis. The resulting sequence of transformations is shown in the Appendix on page 147.

4.4.4 Buildify-catify vs catify-buildify

In the first stage of the fusion transformation, see Figure 4.1, we have two separate steps: buildify and catify and we perform these in the given order. However, the question arises as to what happens if we change their order and perform catify first? Is there any difference in the results? Are there any functions which can be transformed in buildify, catify order (BC in the following) but not in CB order? Essentially, we are asking if the rewrite system, which results from adding catify and buildify (considering both of them as a one-step, conditional rewrite rules) to the usual set of rewrite rules, is confluent or not.

The answer is that this rewrite system is not confluent. Some functions can successfully be transformed in BC order, but doing it in CB order gives more efficient code. Other times BC order fails, while CB succeeds. The original paper on warm fusion [LS95] introduces these problems and note that CB order often requires something called second-order fusion. We
chose not to implement second-order fusion, because as shown by the results of Chapter 6, most functions can be transformed in the much simpler setting of first-order fusion.

**Decision:** We do the transformations in buildify, catify order.

### 4.5 First-order fusion

In this section we present the necessary steps for the simplest case of fusion. First, maps are derived, then catamorphisms. This may seem illogical because Equation 3.1 defines map in terms of its corresponding catamorphism. So in theory, once catamorphisms are derived we get maps for free. In practice, however, even if we use Equation 3.1, we still need to buildify (with the corresponding worker-wrapper split and normalise) the definition because map is also a good producer, unless we are prepared to go all the way and define map in build-cata form. There are two pragmatic reasons to derive the naive code for map:

- In later stages of the compilation (normalise and static argument transformation) the naive definitions are put through the very same sequence of transformations as user written functions. If we defined them in build-cata form buildify and catify would need to be aware that some functions may already be in build-cata form and not attempt the transformation.

- The code for deriving catamorphisms is very much the same as the code for deriving maps, so we get the naive definitions almost by cut and paste.

The following definition applies to the core of this chapter only. We will redefine fusibility in sections dealing with the extensions.

**Definition 4.3 (Fusible datatype)** Regular and polynomial and non-recursive or self-recursive datatypes are fusible. All other datatypes are not fusible.

The fusibility of a datatype is not a general property of the type constructor itself: it only states that these are the datatypes we know how to deal with; we simply give up on the possibility of fusion for all the others.

### 4.5.1 Deriving maps

In the example of rose trees (see Page 53), we demonstrate the need to have a map function for each parametrised, fusible datatype. In that case we need a map for lists. In the general
case, we may need a map for any parametrised, fusible datatype. The existence of maps is established in Chapter 3. Since the method is very similar to the one used to derive catamorphisms, we are not going to work out a detailed example.

Map functions — or type functors \([\text{Fok92b}]\) — are well known in functional programming. The usual reading of the type of map for lists, \(\text{map} :: \forall \alpha \beta. (\alpha \to \beta) \to ([\alpha] \to [\beta])\) is that \text{map} is a polymorphic function which takes a function \(f\) with type \(\alpha \to \beta\) and rewrites a data structure of type \([\alpha]\) to type \([\beta]\) by applying \(f\) to all the occurrences of \(\alpha\).

For each fusible, parametrised datatype, we are going to generate the following code:

\[
\text{map}^T = \Lambda \bar{\alpha} \bar{\beta}. \lambda f_1 \ldots f_m. \lambda t. \text{case } t \text{ of }
\{ T_i \bar{v} \to T_i \bar{\beta} (\text{map}^T \bar{\alpha} \bar{\beta} f_1 \ldots f_m) \bar{v} \}_{i=1}^n
\]

Note: by construction the number of \(\bar{\alpha}\)s is equivalent to the number of \(\bar{\beta}\)s, which is equal to the number of \(f\)’s and the number of type arguments to the datatype (in the head of the data declaration).

\(M\) is defined by induction on the type of its argument. For the syntax of types see Figure A.2. Recall that we do not attempt fusion, or to derive maps for non-polynomial types so foralls and the function space constructor \((\to)\) can not occur as argument type.

\[
M^T f_1 \ldots f_m g v = \mathcal{M}^T f_1 \ldots f_m g (\text{type0f } v) v
\]

where

\[
\begin{align*}
\mathcal{M}^T f_1 \ldots f_m g [\text{primitive}] &= \lambda x.x \\
\mathcal{M}^T f_1 \ldots f_m g [\alpha] &= \lambda x.\{ f_i x | \text{sourceType0f } f_i = \alpha \land i \in \{1 \ldots n\} \} \\
\mathcal{M}^T f_1 \ldots f_m g [T \alpha] &= \lambda x.g x \\
\mathcal{M}^T f_1 \ldots f_m g [K \tau] &= \lambda x.\text{map}^K (\text{tyVars0f}(\text{sourceType0f } g)) \\
&\quad (\text{tyVars0f}(\text{targetType0f } g)) \\
&\quad (\mathcal{M}^T f_1 \ldots f_m g [\tau]) \\
&\quad x
\end{align*}
\]

Note: here are as many functions in \(f_1 \ldots f_m\) as arguments to the type constructor \(T\).

Let’s see what \(M\) does! The first case deals with primitive types, for example the built-in \(\text{Int}\). These types have no maps, therefore \(M\) returns the identity function. The second case, the case of a type variable, is more interesting: we have to find the appropriate \(f\) which rewrites the given type variable. Two questions arise: can we be sure that we find \textit{at least one} \(f\) such that \text{sourceType0f } f is equal to the given type variable and can we be sure that we find \textit{at most one} such \(f\)? The existence and the uniqueness of such \(f\) is guaranteed by the construction of maps (see above).
The similarity between $M$ and $E$ (see Page 52) should be clear. Both functions perform similarly: they apply their argument $g$ recursively to the appropriate type. The reason we need $E$ and $M$ separately is that $M$ takes one function for each parameter (type variable) of the datatype. $E$ does not depend on the number of type arguments.

It is easy to see that Equation 4.3 expands to the well-known definition of map in the case of lists:

\[
\text{map} = \Lambda \alpha \beta. \lambda f t. \text{case } t \text{ of } \\
\langle \text{Equation 4.3} \rangle \\
\begin{array}{l}
\text{[] } \rightarrow \text{[] } \\
(:) a \ as \rightarrow (: \beta (M \text{map}[\alpha \beta f] a)) \\
\end{array}
\]

\[
\langle \text{Equation 4.4} \rangle \\
\begin{array}{l}
\text{[] } \rightarrow \text{[] } \\
(:) a \ as \rightarrow (: \beta (M \text{map}[\alpha \beta f] a)) \\
\end{array}
\]

\[
\langle \text{second and third clause of } M \rangle \\
\begin{array}{l}
\text{[] } \rightarrow \text{[] } \\
(:) a \ as \rightarrow (: \beta (\lambda x f x)) a \\
\end{array}
\]

\[
\langle \text{beta reductions} \rangle \\
\begin{array}{l}
\text{[] } \rightarrow \text{[] } \\
(:) a \ as \rightarrow (: \beta (f a)) \text{map[\alpha \beta f]} a)
\end{array}
\]

And we are done.

### 4.5.2 Deriving catas: implementing the cata evaluation rule

Our starting point is the datatype declarations in source programs (Equation 4.1). For each such declaration, provided the type constructor is fusible according to Definition 4.3, we generate the following code:

\[
cata^T = \Lambda \alpha \rho. \lambda c. \lambda t. \\
\text{case } t \text{ of } \\
\{ T_i \bar{v} \rightarrow c_i (E^T (\text{cata}^T \bar{\alpha} \bar{\rho} \bar{c}) \bar{v}) \}_{i=1}^n
\]
In the equation above, $n$ is the number of constructors the datatype $T\bar{\alpha}$ has, $\rho$ is a fresh type variable, $\bar{c}$ consists of exactly $n$ appropriately typed variables. Functions in $\bar{c}$ correspond to the constructors of $T\bar{\alpha}$, with the recursive occurrences of $T\bar{\alpha}$ replaced by $\rho$. If $\text{monoConstrs}(T\bar{\alpha})$ denotes the list of constructors (with their forall(s) stripped off), the substitution $[\rho/T\bar{\alpha}]$ — substitute $\rho$ for $T\bar{\alpha}$ — will give the right types.

For example, for lists

$$\text{data } [] \alpha = [] | \alpha : [\alpha]$$

$\text{monoConstrs}([\alpha])$ gives the list of monomorphic functions $[[],(:)]$ with types $[\alpha]$ and $\alpha \to [\alpha] \to [\alpha]$ respectively. Applying the substitution $[\rho/T\bar{\alpha}]$ to these two types gives $\rho$ and $\alpha \to \rho \to \rho$. Equipped with this notation, it is easy to give a type to $\text{cata}^T$.

$$\text{cata}^T :: \forall \bar{\alpha}. \forall \rho. \text{mon} \text{C} \text{onstrs}(T\bar{\alpha}) \to T\bar{\alpha} \to \rho$$

In the running example of lists we get

$$\text{cata}[] :: \forall \alpha. \forall \rho. (\alpha \to \rho \to \rho) \to [\alpha] \to \rho$$

We need to give a definition of $E$. For the syntax of types see Figure A.2.

$$E^T_g v = \mathcal{E}^T_g (\text{type0f } v) v \quad (4.6)$$

where

$$\begin{align*}
\mathcal{E}^T_g [\text{primitive type}] &= \lambda x. x \\
\mathcal{E}^T_g [\alpha] &= \lambda x. x \\
\mathcal{E}^T_g [T\bar{\alpha}] &= \lambda x. g x \\
\mathcal{E}^T_g [K \bar{\tau}] &= \lambda x. \text{map}^K (\text{sourceType0f } g) (\text{targetType0f } g) (\mathcal{E}^T_g [\bar{\tau}]) x
\end{align*}$$

Notice, that in the last clause we extended $\mathcal{E}$ from a single type to a list of types with the expected meaning: $\mathcal{E} f \bar{\tau}$ means $(\mathcal{E} f \tau_1) \ldots (\mathcal{E} f \tau_n)$.

For lists, we have

$$(\text{Equation 4.5})$$
This is in fact the familiar foldr function from the Standard Prelude, with its second and third argument swapped around.

A more substantial example, which involves the third clause in the definition of \( E \), is the derivation of the cata for Rose trees.

\[
\text{data } \text{Rose } \alpha = \text{Fork } \alpha [\text{Rose } \alpha]
\]

\[
\text{cata}^{\text{Rose}} :: \forall \alpha. \forall \rho. (\alpha \rightarrow [\rho] \rightarrow \rho) \rightarrow \text{Rose } \alpha \rightarrow \rho
\]

\[
\text{cata}^{\text{Rose}} = \Lambda \alpha. \Lambda \rho. \lambda \text{fork. } \lambda t.
\]

\[\text{case } t \text{ of}
\]

\[
\text{Fork} (a :: \alpha)
\]

\[
(\text{lt} :: [\text{Rose } \alpha]) \rightarrow \text{fork} (E^{\text{Rose}} (\text{cata}^{\text{Rose}} \alpha \rho \text{fork}) [a, \text{lt}])
\]

\[\text{cata}^{\text{Rose}} = \Lambda \alpha. \Lambda \rho. \lambda \text{fork. } \lambda t.
\]

\[\text{case } t \text{ of}
\]

\[
\text{Fork} (a :: \alpha)
\]

\[
(\text{lt} :: [\text{Rose } \alpha]) \rightarrow \text{fork} (E^{\text{Rose}} (\text{cata}^{\text{Rose}} \alpha \rho \text{fork}) \alpha)
\]

\[\text{cata}^{\text{Rose}} = \Lambda \alpha. \Lambda \rho. \lambda \text{fork. } \lambda t.
\]

\[\text{case } t \text{ of}
\]

\[
\text{Fork} (a :: \alpha)
\]

\[
(\text{lt} :: [\text{Rose } \alpha]) \rightarrow \text{fork} (E^{\text{Rose}} (\text{cata}^{\text{Rose}} \alpha \rho \text{fork}) \alpha)
\]

\[\text{cata}^{\text{Rose}} = \Lambda \alpha. \Lambda \rho. \lambda \text{fork. } \lambda t.
\]
\[
cata_{\text{Rose}} = \Lambda \alpha. \Lambda \rho. \lambda \text{fork}. \lambda t.
\]
\[
\text{case } t \text{ of}
\]
\[
\text{Fork } (a :: \alpha) \quad \text{→ fork } a
\]
\[
(lt :: [\text{Rose } \alpha]) \quad \text{→ \( \mathcal{E}_{\text{Rose}} (\cata_{\text{Rose}} \alpha \rho \text{fork}) \lt \)}
\]
\[\langle \mathcal{E} \text{ applied to a type constructor different from the one being defined} \rangle\]
\[
cata_{\text{Rose}} = \Lambda \alpha. \Lambda \rho. \lambda \text{fork}. \lambda t.
\]
\[
\text{case } t \text{ of}
\]
\[
\text{Fork } (a :: \alpha) \quad \text{→ fork } a
\]
\[
(lt :: [\text{Rose } \alpha]) \quad \text{→ \( \lambda \text{map} ([\text{Rose } \alpha]) \rho (\cata_{\text{Rose}} \alpha \rho \text{fork}) \lambda x \lt \)}
\]
\[\langle \beta\text{-reduction} \rangle\]
\[
cata_{\text{Rose}} = \Lambda \alpha. \Lambda \rho. \lambda \text{fork}. \lambda t.
\]
\[
\text{case } t \text{ of}
\]
\[
\text{Fork } (a :: \alpha) \quad \text{→ fork } a
\]
\[
(lt :: [\text{Rose } \alpha]) \quad \text{→ \( \text{map} ([\text{Rose } \alpha]) \rho (\cata_{\text{Rose}} \alpha \rho \text{fork}) \lt \)}
\]

Notice, \( \text{map} \) in the definition! This is a call to the familiar \( \text{map} \) function for lists. We have already shown how to derive the \( \text{map} \) function for arbitrary datatypes in Section 4.5.1. It is easy to verify that \( \cata_{\text{Rose}} \) is well-typed: \( \text{map} \) takes two type arguments \( \text{Rose } \alpha \) and \( \rho \), and a function from \( \text{Rose } \alpha \) to \( \rho \). \( \cata_{\text{Rose}} \alpha \rho \text{fork} \) does indeed have that type. \( \text{map} ([\text{Rose } \alpha]) \rho (\cata_{\text{Rose}} \alpha \rho \text{fork}) \lt \) has type \([\text{Rose } \alpha] \rightarrow [\rho]\) and \( \text{fork} \) has type \( \alpha \rightarrow [\rho] \rightarrow \rho \) which makes the entire expression well-typed.

### 4.5.3 Cata-Core rules

The Core Simplifier need to be extended with several rules to describe how catamorphisms and Core constructs interact. The cata of case rule follows from the strictness property of catamorphisms. The cata of known constructor rule is called cata evaluation rule in Equation (3.3). The \texttt{cata-build} rule is proved correct on Page 24.
4.5. FIRST-ORDER FUSION

\begin{align*}
\langle \text{cata of case rule} \rangle \\
\text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} \left( \text{case } \text{Expr of} \ \{ C \overline{v} \rightarrow e \} \right) & \rightarrow \text{case } \text{Expr of} \ \{ C \overline{v} \rightarrow \text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} e \}
\end{align*}

\begin{align*}
\langle \text{cata of known constructor rule} \rangle \\
\text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} \left( C_i \overline{v}_1 \ldots \overline{v}_n \right) & \rightarrow c_i \left( E^T \left( \text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} \right) \overline{v}_1 \right) \\
& \quad \vdots \\
& \quad \left( E^T \left( \text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} \right) \overline{v}_n \right)
\end{align*}

\begin{align*}
\langle \text{cata-build rule} \rangle \\
\text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} \left( \text{build}^T \overline{\rho} f \right) & \rightarrow f \overline{\rho} \overline{\epsilon}
\end{align*}

\begin{align*}
\langle \text{cata-of-error rule} \rangle \\
\text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} \text{ error} & \rightarrow \text{error}
\end{align*}

\textbf{Figure 4.2} Rules for the interaction of catamorphisms and Core

The local transformations (Table A.1) are still in effect. The notation \( \text{lhs} \rightarrow \text{rhs} \) has its standard meaning: \( \text{lhs} \) reduces to \( \text{rhs} \) in one step.

In the following, we will refer to these rules by their name.

4.5.4 Buildify

We now formally define the algorithm which attempts to transform a function to explicit build form. The transformation’s validity is proved in Section 3.4 and can also be seen by reversing it: if the wrapper is inlined and the definition of \( \text{build} \) is expanded we get back the same definition we started with.

1. Rewrite each function, which produces a fusible result, according to the following rule

\begin{align*}
\text{f} & :: \forall \overline{\alpha}.\overline{\sigma} \rightarrow T\overline{\tau} \\
\text{f} & = \Lambda \overline{\alpha}.\lambda \overline{\nu}.e \\
\implies & \\
\text{f} & :: \forall \overline{\alpha}.\overline{\sigma} \rightarrow T\overline{\tau} \\
\text{f} & = \Lambda \overline{\alpha}.\lambda \overline{\nu}.\text{build}^T \left( f' \overline{\alpha} \overline{\nu} \right) \\
\text{f'} & :: \forall \overline{\alpha}.\overline{\sigma} \rightarrow (\forall \rho. \text{monoConstrs}(T\overline{\tau})[\rho/T\overline{\tau}] \rightarrow \rho) \\
\text{f'} & = \Lambda \overline{\alpha}.\lambda \overline{\nu}.\Lambda \rho.\lambda \overline{\epsilon}.\text{cata}^T \overline{\tau} \overline{\rho} \overline{\epsilon} e
\end{align*}

In effect, we are splitting the definition of \( f \) into a wrapper \( f \) and a worker \( f' \). \( f \) also
gets marked as InlineMe. In GHC this will encourage the Core Simplifier to replace calls to \( f \) with the right hand side of \( f \).

A few remarks about the abundant variables: in the original definition of \( f \), \( \bar{\alpha} \) stands for an arbitrary number of type variables, \( \bar{\sigma} \) stands for the type of an arbitrary number of arguments, where the arguments themselves are denoted by \( \bar{v} \). \( T \bar{\tau} \) is the result type of \( f \), and \( \bar{\tau} \) is built up from type variables from \( \bar{\alpha} \) and applications of fusible type constructors and primitive types (Int, Bool, etc). In fact, \( \bar{\alpha} \) can be a subset of \( \bar{\tau} \) or the other way around. \( e \) stands for an arbitrary core expression that has no more lambdas.

In the resulting definitions of \( f \) and \( f' \), \( \bar{\alpha}, \bar{\tau}, e \) are as above, and \( \rho \) is a fresh, appropriately kinded type variable.

2. Simplify the resulting bindings by calling the Core Simplifier.

3. We check if the transformation is beneficial by traversing the resulting bindings and checking if the \( \text{cata}^T \) disappeared. Leaving the cata would mean an extra traversal. If it disappears then this function is a good producer and we replace the original definition with the newly simplified bindings. Otherwise, we revert to the original definition of \( f \). The machinery in the compiler gives a simple implementation for this step: we mark the cata as ‘radioactive’ [LS95] and when traversing the simplified bindings we check for the absence of the marked identifier.

Let’s look at an example to see how these rules work! We are going to demonstrate it with the simplest possible function: one which when applied to a positive number \( n \), delivers a list of numbers between \( n \) and 0 in decreasing order. We will use Core syntax, except that we are not going to observe the syntactic restriction on arguments, and assume that the corresponding cata has already been derived.

\[
\begin{align*}
downTo & :: \text{Int} \rightarrow \text{[Int]} \\
downTo & = \lambda n. \text{case } n > 0 \text{ of} \\
& \quad \text{True } \rightarrow (;) \text{ Int } n \text{ (downTo } (n-1)) \\
& \quad \text{False } \rightarrow [] \text{ Int}
\end{align*}
\]

We rewrite this binding according to Step 1 and get:

\[
\begin{align*}
downTo & :: \text{Int} \rightarrow \text{[Int]} \\
downTo & = \lambda n. \text{build}^\uplus \text{ (downTo' } n) \\
downTo' & :: \text{Int} \rightarrow (\forall \rho. \rho \rightarrow (\text{Int} \rightarrow \rho \rightarrow \rho) \rightarrow \rho) \\
downTo' & = \lambda n. \Delta \rho. \lambda \text{nil}. \lambda \text{cons}
\end{align*}
\]
4.5. FIRST-ORDER FUSION

\[
\begin{align*}
cata : \text{Int } \rho \text{ nil cons} & \ (\text{case } n > 0 \text{ of} \\
& \quad True \rightarrow (:) \text{Int} \ n \ (\text{downTo} \ (n - 1)) \\
& \quad False \rightarrow [] \text{Int}
\end{align*}
\]

Step 2 calls for the simplifier extended with the rules given in Figure 4.2, which in this case would deliver the result we are expecting, but would not show the intermediate steps. Instead, we detail the workings of the Core Simplifier. Nothing is going to happen to the wrapper \textit{downTo} apart from getting inlined, therefore we omit it. We also omit the type of \textit{downTo}' since it does not change.

\[
\begin{align*}
\langle\text{cata of case}\rangle \\
downTo' &= \lambda n.\Lambda \rho.\lambda \text{nil}.\lambda \text{cons}. \\
\text{case } n > 0 & \text{ of} \\
& \quad True \rightarrow \text{cata} \ \text{Int } \rho \text{ nil cons} ((:) \text{Int} \ n \ (\text{downTo} \ (n - 1))) \\
& \quad False \rightarrow \text{cata} \ \text{Int } \rho \text{ nil cons} ([] \text{Int})
\end{align*}
\]

\[
\langle\text{case of known constructor twice}\rangle \\
downTo' &= \lambda n.\Lambda \rho.\lambda \text{nil}.\lambda \text{cons}. \\
\text{case } n > 0 & \text{ of} \\
& \quad True \rightarrow \text{cons} \ (E \ (\text{cata} \ \text{Int } \rho \text{ nil cons} \ n)) \\
& \quad False \rightarrow \text{nil}
\]

\[
\langle\text{definition of } E \text{ twice}\rangle \\
downTo' &= \lambda n.\Lambda \rho.\lambda \text{nil}.\lambda \text{cons}. \\
\text{case } n > 0 & \text{ of} \\
& \quad True \rightarrow \text{cons} \ (E \ (\text{cata} \ \text{Int } \rho \text{ nil cons} \ n)) \ \text{Int} \ (\text{downTo} \ (n - 1)))
\]

\[
\langle\text{\& applied to a primitive type and } E \text{ applied to the recursive use of } [\alpha]\rangle \\
downTo' &= \lambda n.\Lambda \rho.\lambda \text{nil}.\lambda \text{cons}. \\
\text{case } n > 0 & \text{ of} \\
& \quad True \rightarrow \text{cons} \ ((\lambda x.x) \ n) \\
& \quad False \rightarrow \text{nil}
\]

\[
\langle\beta \text{ reductions}\rangle \\
downTo' &= \lambda n.\Lambda \rho.\lambda \text{nil}.\lambda \text{cons}. \\
\text{case } n > 0 & \text{ of} \\
& \quad True \rightarrow \text{cons} \ n \ (\text{cata} \ \text{Int } \rho \text{ nil cons} \ (\text{downTo} \ (n - 1)))
\]

\[
\langle\text{downTo gets inlined}\rangle \\
downTo' &= \lambda n.\Lambda \rho.\lambda \text{nil}.\lambda \text{cons}. \\
\text{case } n > 0 & \text{ of} \\
& \quad True \rightarrow \text{cons} \ n \ (\text{cata} \ \text{Int } \rho \text{ nil cons} \ (\text{downTo} \ (n - 1)))
\]

\[
\text{False} \rightarrow \text{nil}
\]
4.5. FIRST-ORDER FUSION

\[
downTo' = \lambda n.\Lambda \rho.\lambda \text{nil} \lambda \text{cons}.
\]

\[
case\ n > 0\ of\n\quad True \rightarrow \text{cons}\ n (cata\ Int\ \rho\ \text{nil}\ \text{cons} ((\lambda n'.\text{build} (\downTo'\ n')) (n - 1)))
\]

\[
\beta\ \text{reduction}
\]

\[
downTo' = \lambda n.\Lambda \rho.\lambda \text{nil} \lambda \text{cons}.
\]

\[
case\ n > 0\ of\n\quad True \rightarrow \text{cons}\ n (cata\ Int\ \rho\ \text{nil}\ \text{cons} (\text{build} (\downTo'\ (n - 1))))
\]

\[
\langle\text{cata-build rule}\rangle
\]

\[
downTo' = \lambda n.\Lambda \rho.\lambda \text{nil} \lambda \text{cons}.
\]

\[
case\ n > 0\ of\n\quad True \rightarrow \text{cons}\ n (\downTo'\ (n - 1)\ \text{rho}\ \text{nil}\ \text{cons})
\]

\[
\langle\text{The Core Simplifier finished}\rangle
\]

Simple examination (Step 3) shows that the ‘radioactive’ cata did indeed disappear via the \texttt{cata-build} rule. The wrapper \texttt{downTo} is not recursive anymore and is small. The worker \texttt{downTo'} is recursive, but does not call its wrapper. \texttt{downTo} therefore is a good producer and we replace its old definition with newly derived ones \texttt{downTo} and \texttt{downTo’}.

It maybe somewhat surprising that a program transformation technique applies equally to recursive and non-recursive datatypes. Very much the same thing happens as in the recursive case except that we eliminate a Maybe instead of say a list. Some might say that it is not worth using the big hammer for a single \texttt{Maybe}, but there are other reasons to consider. It gives us a uniform method to eliminate intermediate data structures whether they are recursive or not. Its success entirely depends on heavy inlining which we have to do anyway.

\textbf{data} Maybe \(\alpha\) = Nothing | Just \(\alpha\)

\[
\text{map}^{\text{Maybe}} :: \forall \alpha\beta.\text{Maybe}\ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{Maybe}\ \beta
\]

\[
\text{map}^{\text{Maybe}} = \lambda \alpha\beta.\lambda m\ f.\ \text{case}\ m\ of\n\quad \text{Nothing} \rightarrow \text{Nothing}
\quad \text{Just}\ a \rightarrow \text{Just}\ (f\ a)
\]

\[
\text{map}^{\text{Maybe}} :: \forall \alpha\beta.\text{Maybe}\ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \text{Maybe}\ \beta
\]
Our final example shows when the third clause of $E$ plays a role.

The rose tree data structure is interesting because its type constructor is 'embedded' into another one, that is, a rose tree is an element and a list of rose trees:

\[
data	ext{ Rose } a = \text{ Fork } a \text{ [Rose } a]\]

\[
map^{\text{Rose}} :: \forall \alpha \beta. \text{ Rose } \alpha \to (\alpha \to \beta) \to \text{ Rose } \beta
\]
\[
map^{\text{Rose}} = \Lambda \alpha \beta. \lambda \text{ r f. case r of}
\]
\[
\\quad \\text{Fork a rs \to let}
\]
\[
\quad \quad g = \lambda r'.\text{map}^{\text{Rose}} \alpha \beta r' f
\]
\[
\quad \quad \text{in}
\]
\[
\quad \quad \text{Fork } \beta (f a) (\text{map}^{\text{[]}}(\text{Rose } \alpha) (\text{Rose } \beta) g rs)
\]

After performing Steps 1 to 3 of buildify we get:

\[
map^{\text{Rose}} :: \forall \alpha \beta. \text{ Rose } \alpha \to (\alpha \to \beta) \to \text{ Rose } \beta
\]
\[
map^{\text{Rose}} = \Lambda \alpha \beta. \lambda \text{ r f. build}^{\text{Rose}} (\text{map}^{\text{Rose}} \alpha \beta r f)
\]
\[
map^{\#\text{Rose}} :: \forall \alpha \beta. \text{ Rose } \alpha \to (\alpha \to \beta) \to (\forall \rho. (\alpha \to \rho) \to \rho)
\]
\[
map^{\#\text{Rose}} = \Lambda \alpha \beta. \lambda \text{ r f. } \text{fork}
\]
let
\[ g = \lambda r'. \text{map}^{\text{Rose}} \alpha \beta r' f \]
\[ c = \text{case } r \text{ of} \]
\[ \text{Fork } a \text{ } rs \rightarrow \text{Fork } \beta (f \ a) \ (\text{map}^\| (\text{Rose } \alpha) (\text{Rose } \beta) g \ rs) \]
\[ \text{in} \]
\[ \text{cata}^{\text{Rose}} \beta \rho \text{fork } c \]

In the result, we observed the syntactic restriction in Core and let bound every argument. Notice, the \text{map}^\| in the body of \text{map}^\|^{\text{Rose}}.

### 4.5.5 Catify

The process of automatically turning arbitrary functions into catamorphisms is theoretically much simpler than buildify. Unfortunately, its implementation is definitely worse. Most of the problems are due to the way GHC is structured. Not that GHC is badly structured, but it takes an approach which seems to be hard to combine with the steps we need to take to implement this transformation.

1. Rewrite each function, which consumes a fusible argument, according to the following rewrite rule

\[ f :: \forall \alpha. T \bar{\tau} \rightarrow \sigma \]
\[ f = \Lambda \bar{\alpha}. \lambda t. e \]
\[ \Rightarrow \]
\[ f :: \forall \alpha. T \bar{\tau} \rightarrow \sigma \]
\[ f = \Lambda \bar{\alpha}. \lambda t. \text{cata}^T \bar{\tau} \ (f_{C_1} \bar{\alpha}) \ldots (f_{C_n} \bar{\alpha}) t \]

\[ f_{C_1} = \Lambda \bar{\alpha}. \lambda \bar{\tau} \text{ let} \]
\[ f'_{C_1} = \lambda t. e \]
\[ \text{in} \]
\[ f'_{C_1} (C_1 \bar{\tau} \bar{t_1}) \]
\[ :\]
\[ f_{C_n} = \Lambda \bar{\alpha}. \lambda \bar{\tau} \text{ let} \]
\[ f'_{C_n} = \lambda t. e \]
\[ \text{in} \]
\[ f'_{C_n} (C_n \bar{\tau} \bar{t_n}) \]

The additional criterion that \( f \) is strict in \( t \) (see Equation 3.5), can be discovered in two ways: either the annotation for \( f \) tells us or \( e \) is a case expression on \( t \). In Core,
4.5. FIRST-ORDER FUSION

case expressions always perform evaluation (see Appendix A for details), therefore they are strict.

A few comments about the variables: In the original binding for \( f \), \( \bar{\alpha} \) stands for an arbitrary number of type variables. \( T\bar{\tau} \) is a fusible type with the corresponding variable \( \bar{\tau} \). \( \bar{\tau} \) is built up from type variables from \( \bar{\alpha} \) and applications of fusible type constructors and primitive types (Int, Bool, etc). \( \sigma \) is the result type of \( f \). Notice, that \( f \) has only one argument\(^5\) and this argument is fusible.

In effect, the rewrite rule splits \( f \) into a wrapper (also denoted \( f \), since we need a definition for it) and \( n \) workers (denoted \( f_{C_1} \ldots f_{C_n} \)). By construction, \( n \) is equal to the number of constructors \( T\bar{\alpha} \) has. In the examples, we will use the name of the constructor instead of numbers, so for example a function \( g \) consuming a list will be split into \( g \), the wrapper, worker \( g_{\text{Nil}} \) for the \text{Nil} constructor and worker \( g_{\text{Cons}} \) for the \text{Cons} constructor.

In the rewritten bindings (after the \( \Rightarrow \)), \( \bar{\alpha} \), \( T\bar{\tau} \), \( \sigma \) and \( e \) are as above. There are two new sets of variables (in each worker), \( \bar{\tau}_n \) and \( \bar{t}_n \). The variables denoted by \( t \) are appropriately typed (with respect to the type argument \( \tau \) to \( C_n \)), fresh variables and \( \bar{\tau}_n \) and \( \bar{t}_n \) have equal number of (similarly) typed elements. \( n \) does not refer to the number of elements, but to the constructor this particular \( z \) belongs to. In other words, the \( z \)'s and \( t \)'s are different in each worker. Notice that \( z \)'s are never used in the body of their respective bindings and \( t \)'s are free.

Notice, that the wrapper \( f \) is small and non-recursive, while the workers can be arbitrarily big.

The astute reader will notice that Core syntax (see Appendix A.2) does not allow the formation of the right hand sides of the rewritten bindings. In particular, arguments to the application of an expression \( e \) are restricted to be Atoms while in our case the argument is an expression \( C_1 \bar{\tau} \bar{t}_1 \). The usual way around this restriction is to let bind the expression \( C_1 \bar{\tau} \bar{t}_1 \) to an appropriately typed variable and mark it as used once only (linear). The Core Simplifier then will do its job.

It may be somewhat worrying for those afraid of code explosion that in the workers we duplicate the entire body of the original function. This is not an issue however, since these are simplified which makes the case go away. We shall see an example of this below.

2. Simplify the resulting bindings, by calling the Core simplifier.

\(^5\) We will relax this condition in Section 5.1
3. Construct the rules of a rewrite system on-the-fly and do a second rewriting. We define the rules and study the rewrite system in Section 5.3. These rules will be built up from combinations of $t$’s with $f$ on the left hand sides and $z$’s on the right-hand sides.

4. Traverse the rewritten bindings and check for free occurrences of $t$. The presence of any $t$ denotes failure of the transformation, in which case revert to the original definition of $f$. If no $t$’s occur (the bindings are closed) then we succeeded in transforming $f$ to an explicit catamorphic form, so replace the original definition with the result of the previous step.

We will go through a detailed example to show how these rules work; later in this chapter we relax most of the restrictions to make the process of turning functions to explicit catamorphic form more general. The example we are going to use is the well known $length$ function for lists from the Prelude. Note, that $length$ satisfies all of the restrictions: it has only one argument, and that is fusible.

\[
length :: \forall \alpha. [\alpha] \rightarrow Int
\]

\[
length = \Lambda \alpha. \lambda l. \begin{cases} 
[\ ] \rightarrow 0 \\
(x : xs) \rightarrow 1 + length \alpha xs
\end{cases}
\]

According to Step 1, we rewrite this definition to:

\[
length :: \forall \alpha. [\alpha] \rightarrow Int
\]

\[
length = \Lambda \alpha. \lambda l. \text{case } l \text{ of } \\
[\ ] \rightarrow 0 \\
(x : xs) \rightarrow 1 + length \alpha xs
\]

\[
length_\parallel = \Lambda \alpha. \text{let } length^* = \Lambda \alpha. \lambda l. \begin{cases} 
[\ ] \rightarrow 0 \\
(x : xs) \rightarrow 1 + length \alpha xs
\end{cases} \text{ in } \\
length_\parallel = \Lambda \alpha. \lambda l. \text{case } l \text{ of } \\
[\ ] \rightarrow 0 \\
(x : xs) \rightarrow 1 + length \alpha xs
\]

According to Step 2, call the Core Simplifier: the definitions marked with $*$ will get inlined, and two $\beta$ reductions happen (in both bindings).
4.5. FIRST-ORDER FUSION

\[ length :: \forall \alpha.\{\alpha\} \rightarrow \text{Int} \]
\[ length = \Lambda \alpha.\lambda l. \text{cata}[] \alpha [\alpha] (\text{length}[] \alpha) (\text{length}_{(:)} \alpha) l \]
\[ \text{length}[] = \Lambda \alpha.0 \]
\[ \text{length}_{(:)} = \Lambda \alpha.\lambda z zs.1 + \text{length} \alpha ts \]

In effect, we partially evaluated the definition of \( length \) with respect to its known first argument.

We construct the rules of a rewrite system according to the definition in Section 5.3; the function we are transforming, \( length \), and the free variables \( t \) and \( ts \) will be on the left-hand sides, while \( z \) and \( zs \) will be on the right-hand sides. There are no rules corresponding to the [] case, since this constructor has no arguments. However, there are two rules for (:), because it has two arguments: one of type \( \alpha \) and another of type \([\alpha]\).

\[ \{ t \rightarrow z, \text{length} \alpha ts \rightarrow zs \} \]

We rewrite the simplified bindings using these rules and get:

\[ length :: \forall \alpha.\{\alpha\} \rightarrow \text{Int} \]
\[ length = \Lambda \alpha.\lambda l. \text{cata}[] \alpha [\alpha] (\text{length}[] \alpha) (\text{length}_{(:)} \alpha) l \]
\[ \text{length}[] = \Lambda \alpha.0 \]
\[ \text{length}_{(:)} = \Lambda \alpha.\lambda z zs.1 + zs \]

Simple examination shows, that combinations of \( length \) with pre-recursion variables, \( t \) and \( ts \) have been eliminated. Therefore we succeeded in transforming \( length \) into an explicit catamorphic form; the original definition of length can be replaced by the newly derived bindings.

The example also demonstrates that the catify split is not as good as it could be: in the \( \text{length}[] \) wrapper, the type variable is unnecessary. It is a simple modification to the rewrite step, but it would complicate the notation considerably to ensure that no unused type or value arguments are passed to the wrappers. The implementation never passes unused arguments to wrappers.
Chapter 5

The Practice of Warm Fusion II: Extensions

This chapter is devoted to two extensions of the basic case: fusion for higher-order catamorphisms and fusion for mutually recursive datatypes. We also introduce a transformation, the normalisation of the order of arguments, which seems rather simple — and new in the literature — but has the surprising effect of simplifying other transformations. This is discussed in Section 5.4.

Finally, in Section 5.3 we present and study the ‘dynamic’ rewrite system. Section 5.5 discusses two closely related issues: fusion in the presence of separate compilation and how fusion could simplify the Desugaring (see Figure A.1) phase of the compiler.

5.1 Functions with more than one argument

In the previous sections, we thoroughly explored the two transformations which, if they succeed, turn arbitrary functions into explicit catamorphic and explicit build forms. We also discussed the modifications which were required to be made the Core Simplifier to make the transformations useful. The restrictions we imposed on functions (only one argument and that is fusible) are rather severe and limit the usefulness of the fusion. In this section, we relax this criteria and allow functions with more than one argument. We shall see that there are several ways to do this, and each approach comes with its own limitation. The definition of fusibility remains as in Section 4.5.
5.1. FUNCTIONS WITH MORE THAN ONE ARGUMENT

5.1.1 Avoiding more than one argument

With regards to functions with more than one argument, one surprisingly frequent viable way to cope with them is to avoid them. The technique is similar to that of Wadler [Wad90], where he uses higher-order macros to extend his first-order deforestation to apply to certain higher-order functions. The required transformation is called the static argument transformation (SAT)[San95]; it stems from the observation that in many cases arguments to functions in the recursive call do not change: they are static. Consider the well known append function for lists:

\[
\text{append} :: \forall \alpha. [\alpha] \rightarrow [\alpha] \\
\text{append} = \lambda \alpha. \lambda xs ys. \text{case } xs \text{ of} \\
\hspace{1cm} [] \rightarrow ys \\
\hspace{1cm} (:) x xs \rightarrow (:\alpha) x (\text{append } \alpha xs ys)
\]

In the body of append and in recursive calls to append itself, \(\alpha\) and \(ys\) are the same as the binders. Therefore, these arguments need not be passed around in recursive calls and we can transform append into:

\[
\text{append} :: \forall \alpha. [\alpha] \rightarrow [\alpha] \\
\text{append} = \Lambda \alpha. \lambda xs ys. \text{let} \\
\hspace{1cm} \text{append'} :: [\alpha] \rightarrow [\alpha] \\
\hspace{1cm} \text{append'} = \lambda xs. \text{case } t \text{ of} \\
\hspace{2cm} [] \rightarrow ys \\
\hspace{2cm} (x : xs) \rightarrow (:\alpha) x (\text{append'} xs) \\
\hspace{2cm} \text{in} \\
\hspace{3cm} \text{append'} xs
\]

We created a local function, append' which does not pass the static arguments around. The static arguments are free in the body of append', but this does not cause any problems, since they are bound by the outer lambdas. Section 5.1.6 formalises the transformation.

Now, we can perform catify on the local append' function using the techniques of the previous section, since it has only one fusible argument and we get:

\[
\text{append} :: \forall \alpha. [\alpha] \rightarrow [\alpha] \\
\text{append} = \Lambda \alpha. \lambda xs ys. \text{let} \\
\hspace{1cm} \text{append'} :: [\alpha] \rightarrow [\alpha] \\
\hspace{1cm} \text{append'} = \lambda xs. \text{case } t \text{ of} \\
\hspace{2cm} [] \rightarrow ys \\
\hspace{2cm} (x : xs) \rightarrow (:\alpha) x (\text{append'} xs) \\
\hspace{2cm} \text{in} \\
\hspace{3cm} \text{append'} xs
\]
append = \alpha. \lambda xs ys. let

append' :: [\alpha] \to [\alpha]
append' = \alpha xs. cata [\alpha] \alpha [\alpha] append' xs
append' :: [\alpha]
append' = ys
append'(1) :: \alpha \to [\alpha] \to [\alpha]
append'(1) = \lambda z zs. (:) \alpha z zs

in
append' xs

If we inline append' in the body of the let expression, we have:

append :: \forall \alpha. [\alpha] \to [\alpha] \to [\alpha]
append = \alpha. \lambda xs ys. let

append' :: [\alpha]
append' = ys
append'(1) :: \alpha \to [\alpha] \to [\alpha]
append'(1) = \lambda z zs. (:) \alpha z zs

in
cata [\alpha] xs append'(1) append'(1) xs

This is very good indeed! We transformed a function with two arguments into a first-order catamorphism. The approach we are advocating in the rest of this section will derive a slightly different form of the append function (provided buildify is not run before catify):

append :: \forall \alpha. [\alpha] \to [\alpha] \to [\alpha]
append = \alpha. \lambda xs ys. let

append' :: [\alpha] \to [\alpha]
append' = ys. ys
append'(1) :: \alpha \to [\alpha] \to [\alpha]
append'(1) = \lambda z zs. (:) \alpha z (zs ys)

in
cata [\alpha] xs append'(1) append'(1) xs ys

The second argument, ys, is now passed around in the recursive calls and the type of the local functions have changed accordingly. Intuitively, this definition is slightly less efficient because of the additional argument. We would, therefore, prefer to use static argument transformation whenever possible. The usefulness of this approach, using SAT
whenever possible, is amply demonstrated by the fact that a large number of functions, 
\textit{map}, \textit{span}, \textit{break}, \textit{takeWhile}, \textit{filter}, \textit{init} etc, from the Standard Prelude shown in the next 
section, in the presence of mutually recursive datatypes, SAT makes it nearly impossible to 
successfully transform a group of mutually recursive functions.

There is another drawback of using SAT: fusion does not happen on static arguments:

\[
\text{append } e \left( \text{build}^{[]} \alpha \rho (g \ldots) \right) \not\rightarrow \text{append } e(g \alpha \ldots \rho n c)
\]

Section 5.1.5 shows a method to achieve fusion for more than one argument.

\subsection{Higher-order catas}

Let us consider now the situation when the function being transformed has more than one 
non-static argument. There are dozens of well known, Standard Prelude functions we could 
use, but for the sake of showing that all these techniques work for other datatypes than 
lists, we are going to use the \textit{level} function for trees. \textit{level} has type \( \text{Tree } \alpha \rightarrow \text{Int } \rightarrow \llbracket \alpha \rrbracket \); 
it takes a tree and a number and returns the elements on that level of the tree. The root 
of the tree is at level 0. \textit{level} genuinely requires higher-order catamorphisms as none of its 
arguments are static, so the techniques detailed in the previous section would not work.

Given the datatype declaration for trees

\[
\textbf{data } \text{Tree } \alpha = \text{Empty } | \text{Branch } \alpha (\text{Tree } \alpha)(\text{Tree } \alpha)
\]

the corresponding catamorphism (as derived by the algorithm in Section 4.5.2) is:

\[
cata^{\text{Tree}} \:: \forall \alpha \rho. \rho \rightarrow (\alpha \rightarrow \rho \rightarrow \rho \rightarrow \rho) \rightarrow \text{Tree } \alpha \rightarrow \rho
\]

\[
cata^{\text{Tree}} = \Lambda \alpha \rho. \lambda e b t. \textbf{case } t \textbf{ of}
\]

\[
\text{Empty } \rightarrow e
\]

\[
\text{Branch } x \text{ \lt } \text{rt } \rightarrow b x (\text{cata}^{\text{Tree}} \alpha \rho \text{lt}) (\text{cata}^{\text{Tree}} \alpha \rho \text{rt})
\]

the naive definition of level:

\[
\text{level } \text{Empty } n = []
\]

\[
\text{level } (\text{Branch } x \text{ \lt } \text{rt}) 0 = [x]
\]

\[
\text{level } (\text{Branch } x \text{ \lt } \text{rt}) n \mid n > 0 = \text{level } \text{lt } (n - 1) \; \text{ ++ } \; \text{level } \text{rt } (n - 1)
\]

which, in turn translates to (Desugarer):
The reader is invited to verify that buildify succeeds and we get:

\[
\text{level} :: \forall \alpha. \text{Tree } \alpha \to \text{Int} \to [\alpha] \\
\text{level} = \Lambda \alpha. \lambda t i. \text{case } t \text{ of } \\
\text{Empty} \to [] \alpha \\
\text{Branch } x \text{ lt rt} \to \text{case } i == 0 \text{ of } \\
\text{True} \to (:) \alpha x ([]) \alpha \\
\text{False} \to \text{case } i > 0 \text{ of } \\
\text{True} \to \text{append } \alpha \\
\text{False} \to (\text{level }\alpha \text{ lt } (i - 1)) \alpha \\
\text{False} \to (\text{level }\alpha \text{ rt } (i - 1)) \alpha
\]

The syntactic criteria for the success of the transformation holds, so we accept this definition. It’s interesting to note the third argument, which stands for the \([\ ]\) constructor, to \text{level’}. It is the traversal of the right branch, which is an artefact of \text{append} getting inlined. We start to catify this definition of \text{level}. After rewriting and simplification we have:

\[
\text{level} :: \forall \alpha. \text{Tree } \alpha \to \text{Int} \to [\alpha] \\
\text{level} = \Lambda \alpha. \lambda t i. \text{build}() (\text{level’ }\alpha t i) \\
\text{level’} :: \forall \alpha. \text{Tree } \alpha \to \text{Int} \to (\forall \rho. \rho \to (\alpha \to \rho) \to \rho) \\
\text{level’} = \Lambda \alpha. \lambda t i. \Lambda \rho. \lambda e b. \text{cata}^{\text{Tree }\alpha \rho} (\text{level’}_\text{Empty }\alpha) (\text{level’}_\text{Branch }\alpha) t i e b \\
\text{level’}_\text{Empty} = \Lambda \alpha. \lambda i e b. e
\]
5.1. FUNCTIONS WITH MORE THAN ONE ARGUMENT

\[ \text{level}'_{\text{Branch}} = \lambda \alpha. \lambda z z \text{lt} z \text{rt} i e b. \text{case } i == 0 \text{ of } \]
\[ \text{True } \rightarrow b \, t x \, e \]
\[ \text{False } \rightarrow \text{case } i > 0 \text{ of } \]
\[ \text{True } \rightarrow \text{level}' \, \alpha \, \text{lt} \, (i - 1) \]  
\[ \text{(level}' \, \alpha \, \text{trt} \, (i - 1) \, e \, b) \]  
\[ \text{False } \rightarrow e \]

The rewrite system is constructed from the function level', the new, free variables \( tx, \text{lt} \) and \( \text{trt} \) and the new appropriately typed variables \( zx, \text{lt} \) and \( \text{rt} \).

\[ \mathcal{R} = \{ tx \rightarrow zx, \text{level}' \, \alpha \, \text{lt} \rightarrow zl, \text{level}' \, \alpha \, \text{trt} \rightarrow zr \} \]

The second rewriting then gives:

\[ \text{level} :: \forall \alpha. \text{Tree} \, \alpha \rightarrow \text{Int} \rightarrow [\alpha] \]
\[ \text{level} = \Lambda \alpha. \Lambda t \, i. \text{build}^{1} \, (\text{level}' \, \alpha \, t \, i) \]
\[ \text{level}' :: \forall \alpha. \text{Tree} \, \alpha \rightarrow \text{Int} \rightarrow (\forall \rho. \rho \rightarrow (\alpha \rightarrow \rho) \rightarrow \rho) \]
\[ \text{level}' = \Lambda \alpha. \Lambda t \, i. \Lambda \rho. \lambda e \, b. \text{cata}^{\text{Tree}} \, \alpha \, \rho \, (\text{level}'_{\text{Empty}} \, \alpha \, t \, i \, e \, b) \]
\[ \text{level}'_{\text{Empty}} = \Lambda \alpha. \lambda i \, e \, b. e \]
\[ \text{level}'_{\text{Branch}} = \lambda \alpha. \lambda z z \text{lt} z \text{rt} i e b. \text{case } i == 0 \text{ of } \]
\[ \text{True } \rightarrow b \, z x \, e \]
\[ \text{False } \rightarrow \text{case } i > 0 \text{ of } \]
\[ \text{True } \rightarrow \text{zlt} \, (i - 1) \, (\text{zrt} \, (i - 1) \, e \, b) \, b \]
\[ \text{False } \rightarrow e \]

Notice, that \( \text{cata}^{\text{Tree}} \, \alpha \, \rho \, (\text{level}'_{\text{Empty}} \, \alpha \, t) \) is a function, which traverses the structure \( t \) and constructs a function.

5.1.3 Buildify

Now we formalise the method we applied in the previous example. As we mentioned earlier in this section, the transformation to explicit build form requires very little change if we want to allow more than one argument. In fact, when we gave the precise algorithm and the rewrite rule on page 55, we already allowed for an arbitrary number of arguments. Only Step 3 changes:

3. Traverse the resulting bindings and check if the \( \text{cata}^{T} \) disappeared from arguments it was originally introduced on. If it did, then this function is a good producer and
we replace the original definition with the newly simplified bindings. Otherwise, we revert to the original definition of \( f \).

As the changes are not substantial from the first-order case, we do not give a detailed example.

### 5.1.4 Catify

Transforming unary functions to explicit catamorphic form is simple: the function has only one argument which is fusible, so it is immediately obvious on which argument we need to introduce the cata. When the function has more than one argument we need to decide which one we want to fuse on. In some cases, when there is only one fusible argument, like in the case of \( \text{map} \), the choice is still obvious.

But what happens, if we have two or more fusible arguments? There seems to be several options:

1. Pick the first one
2. Pick one in which the function is strict

Using the first fusible argument (1) is a rather good choice since it is simple. However, the fusion law for catamorphisms, Equation 3.4 (aka. promotion theorem), on which we based the catify transformation, requires the function to be strict in the given argument. This would force us to use the first fusible datatype in which the function is strict, which would complicate the implementation: for one function we would introduce the cata on its first argument, for another on its fifth. Instead, we rearrange the order of arguments to functions. The transformation described in Section 5.4 details this simple process.

From now on, we will assume that every function which is a candidate for the transformation had its arguments rearranged so that the function is strict in its first fusible argument. In other words, we will always try to introduce the cata on the first argument. With this assumption it is easy to extend the catify transformation. The skeleton of the algorithm remains the same as in Section 4.5.5, only the rewrite step changes.

1. Rewrite each function which consumes a fusible argument, according to the following rewrite rule:

\[
\forall \alpha. T \tau \rightarrow \sigma
\]
5.1. FUNCTIONS WITH MORE THAN ONE ARGUMENT

\[ f = \lambda \tilde{\alpha}. \lambda \bar{t} \bar{v}. e \]

\[ \implies f :: \forall \tilde{\alpha}. T \tilde{\bar{r}} \rightarrow \bar{\sigma} \]

\[ f = \lambda \tilde{\alpha}. \lambda \bar{t} \bar{v}. \text{cata}^T \tilde{\bar{r}} (T \tilde{\bar{r}}) (f_{C_1} \tilde{\alpha}) \ldots (f_{C_n} \tilde{\alpha}) \bar{t} \]

\[ f'_{C_1} = \lambda \bar{t} e \]

\[ \text{in} \]

\[ f'_{C_1} (C_1 \bar{\bar{r}} \bar{t}_1) \]

\[ \vdots \]

\[ f'_{C_n} = \lambda \bar{t} e \]

\[ \text{in} \]

\[ f'_{C_n} (C_n \bar{\bar{r}} \bar{t}_n) \]

The comments we made when we first gave this algorithm also apply here (see page 60.) The difference is that now we allow an arbitrary number of arguments, denoted \( \bar{v} \), to \( f \). The workers change accordingly.

2. Simplify the resulting bindings.

3. Construct the rules of a rewrite system on-the-fly and do a second rewriting. We define the rules and study the rewrite system in Section 5.3. These rules will be built up from combinations of \( t \)'s with \( f \) on the left hand sides and \( z \)'s on the right-hand sides.

4. Traverse the rewritten bindings and check for free occurrences of \( t \). The presence of any \( t \) denotes failure of the transformation, in which case revert to the original definition of \( f \). If no \( t \)'s occur (the bindings are closed) we succeeded in transforming \( f \) to an explicit catamorphic form, so replace the original definition with the result of the previous step.

An example will nicely demonstrate the workings of the above algorithm. We will use map again, for simplicity. According to the assumption that the function’s arguments are rearranged before catify is attempted, the transformation which performs this is formalised in Section 5.4, we start with the following definition.

\[ \text{map}^{[]} :: \forall \alpha \beta. [\alpha] \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta] \]

\[ \text{map}^{[]} = \Lambda \alpha \beta. \lambda t. \lambda f. \text{case } t \text{ of} \]

\[ [] \rightarrow [] \beta \]

\[ (x : xs) \rightarrow (;) \beta (f x) (\text{map}^{[]} \alpha \beta xs f) \]
According to Step 1 we split this definition into three:

\[
\text{map} \mapsto \forall \alpha \beta. [\alpha] \to (\alpha \to \beta) \to [\beta] \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{cata} \alpha [\alpha] (\text{map} \mapsto \alpha \beta) (\text{map} \mapsto_{} \alpha \beta) t f \\
\text{map}_1 \mapsto \forall \alpha \beta. (\alpha \to \beta) \to [\beta] \\
\text{map}_1 = \Lambda \alpha \beta. \lambda f. \text{let} \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{case } t \text{ of} \\
[\!\!\!] \to [\!\!] \beta \\
(x : xs) \to (:) \beta (f x) (\text{map} \mapsto \alpha \beta xs f)) \\
\text{map}_2 \mapsto \forall \alpha \beta. [\alpha] \to (\alpha \to \beta) \to (\alpha \to \beta) \to [\beta] \\
\text{map}_2 = \Lambda \alpha \beta. \lambda z zs. \lambda f. \text{let} \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{case } t \text{ of} \\
[\!\!\!] \to [\!\!] \beta \\
(x : xs) \to (:) \beta (f x) (\text{map} \mapsto \alpha \beta xs f)) \\
\text{map}_3 \mapsto \forall \alpha \beta. \alpha \to (\alpha \to \beta) \to [\beta] \\
\text{map}_3 = \Lambda \alpha \beta. \lambda t. \text{case } t \text{ of} \\
[\!\!\!] \to [\!\!] \beta \\
(f x) (\text{map} \mapsto \alpha \beta xs f)) \\
\text{map}_1 = \Lambda \alpha \beta. \lambda f. \text{let} \\
\text{map} = \Lambda \alpha \beta. \lambda z zs. \lambda f. (:) \beta (f t) (\text{map} \mapsto \alpha \beta ts f)
\]

The Simplifier is called, which performs a few $\beta$ reductions:

\[
\text{map} \mapsto \forall \alpha \beta. [\alpha] \to (\alpha \to \beta) \to [\beta] \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{cata} \alpha [\alpha] (\text{map} \mapsto \alpha \beta) (\text{map} \mapsto_{} \alpha \beta) t f \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{cata} \alpha [\alpha] (\text{map} \mapsto \alpha \beta) (\text{map} \mapsto_{} \alpha \beta) t f \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{let} \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{case } t \text{ of} \\
[\!\!\!] \to [\!\!] \beta \\
(x : xs) \to (:) \beta (f x) (\text{map} \mapsto \alpha \beta xs f)) \\
\text{map} = \Lambda \alpha \beta. \lambda t. \text{case } t \text{ of} \\
[\!\!\!] \to [\!\!] \beta \\
(f x) (\text{map} \mapsto \alpha \beta xs f)) \\
\text{map} = \Lambda \alpha \beta. \lambda f. \text{let} \\
\text{map} = \Lambda \alpha \beta. \lambda z zs. \lambda f. (:) \beta (f t) (\text{map} \mapsto \alpha \beta ts f)
\]

Just like in the earlier case, we have new unused variables $z$ and $zs$ and free variables $t$ and $ts$. The rewrite system will replace the pre-recursion variables $t$ and $\text{map} \mapsto \alpha \beta ts$ with $z$ and $zs$. The rules are:

\[
\{ t \to z, \text{map} \mapsto \alpha \beta ts \to zs \}
\]

After rewriting we get:
5.1. FUNCTIONS WITH MORE THAN ONE ARGUMENT

\[
\text{map} :: \forall \alpha \beta. [\alpha] \to (\alpha \to \beta) \to [\beta]
\]

\[
\text{map} = \Lambda \alpha \beta. \lambda t. \lambda f. \text{cata} [\alpha] (\text{map} [\alpha] \beta) (\text{map} (:) \alpha \beta) t f
\]

\[
\text{map} :: \forall \alpha \beta. (\alpha \to \beta) \to [\beta]
\]

\[
\text{map} = \Lambda \alpha \beta. \lambda f. [] \beta
\]

\[
\text{map} :: \forall \alpha \beta. \alpha \to ((\alpha \to \beta) \to [\beta]) \to (\alpha \to \beta) \to [\beta]
\]

\[
\text{map} = \Lambda \alpha \beta. \lambda z zs. \lambda \cdot (:) \beta (f \cdot z) (zs \cdot f)
\]

Notice the similarity with the second definition of `append` given on page 66. In particular, the type of `zs` has changed from `[\beta]` to `((\alpha \to \beta) \to [\beta])`, so the catamorphism instead of building a list it builds a function, which when applied to the missing argument `f` produces the final list. The drawback we noted earlier, that no fusion happens on the second argument, remains.

5.1.5 Higher-order fusion

To see what goes wrong if we try to catify a function with more than one argument and still expect fusion, consider the example of the reverse function for lists, this time written with an accumulating argument.

\[
lrev :: \forall \alpha.[\alpha] \to [\alpha] \to [\alpha]
\]

\[
lrev = \Lambda \alpha. \lambda xs ys. \text{case xs of}
\]

\[
\begin{align*}
\text{[]} & \to ys \\
(,:) x xs & \to lrev \alpha x s ((,:) \alpha x ys)
\end{align*}
\]

Parametricity, we used to prove the validity of fusing an arbitrary (strict) function with a catamorphism, now gives a different theorem:

\[
\forall a : A \to A', a : A \to A', r : R \to R',
\]

\[
\begin{align*}
\text{if } & r \cdot n = n' \cdot b \\
\land & r \cdot c x xs = c' x' xs' \cdot b \leftarrow a x = x' \land r \cdot xs = xs' \cdot b \\
\land & a ys = ys' \cdot \text{map} [\alpha] a \\
\land & b w = w' \\
\land & r \text{ strict}
\text{ then }
\end{align*}
\]

\[
r \left( \text{cata} n c ys w \right) = \text{cata} n' c' ys' (b w) \tag{5.1}
\]

While Equation 5.1 — the second-order fusion theorem — does not look very different from its first-order counterpart (Equation 3.4) the premises are much more complicated.
Since we interpret these premises as rewrite rules, the rewrite system needs to be more elaborate. Even if we were prepared to accept that additional complication, coming up with an appropriate $b$ in the general case is rather difficult. Because of this difficulty, we do not attempt higher-order fusion.

### 5.1.6 Static argument transformation

Santos [San95] devotes a whole chapter of his thesis to the static argument transformation and its relation to lambda lifting. He notes that lambda lifting undoes the effect of static argument transformation. This, however, doesn’t need to concern us: we use SAT as a temporary solution. Once catify succeeds, the function is in explicit catamorphic form. If lambda lifting is used afterwards (like in GHC) that will float out local bindings but will not affect the fusion transformation. The algorithm below formalises the static argument transformation:

- We record the name of the bound variables (both value $\lambda$ and type $\Lambda$) in the function right hand side.
- For every recursive call of the function we check if this call repeats any arguments in the same position as they were in the function definition.
- We define a local, recursive function which uses the static arguments as free variables.

\[
\begin{align*}
f &= \Lambda \bar{\alpha}.\lambda \bar{v}.e \\
\implies f &= \Lambda \bar{\alpha}.\lambda \bar{v}. \text{let} \\
& \quad \quad \quad \quad f' = \Lambda \text{notStatic}(\bar{\alpha}).\lambda \text{notStatic}(\bar{v}).e' \\
& \quad \quad \quad \quad \text{in} \\
& \quad \quad \quad \quad f' \text{notStatic}(\bar{\alpha}) \text{notStatic}(\bar{v})
\end{align*}
\]

In $e'$ calls to $f$ are replaced by calls to $f'$ and the static arguments are dropped. We only perform SAT for functions which have one non-static argument.

### 5.2 Mutually recursive datatypes

After the first-order case and the higher-order extension we finally consider the extension to mutually recursive datatypes. The order of presenting these extensions is important.
5.2. MUTUALLY RECURSIVE DATATYPES

As we shall see, transformations of groups of mutually recursive functions require that our machinery can handle the higher-order case.

There are two ways to deal with mutually recursive datatypes. One way is to reduce mutual recursion on the type level to direct recursion by standard techniques. This is thoroughly investigated in Fokkinga’s thesis [Fok92b]. The other technique is to deal with the additional complexity and have mutually recursive terms as well.

The standard technique to reduce mutual recursion to single recursion is to invent a new datatype which encompasses all the constructors of the mutually recursive group and redefine all the functions which act on the original group of datatypes in terms of the newly invented one. For example,

```
data T a = T1 a | T2 (K a) | T3 (T a)
data K a = K1 | K2 a (T a) | K3 a a (K a)
```

(iis transformed to)

```
data RTK a = RT1 a | RT2 (RTK a) | RT3 (RTK a)
| RK1 | RK2 a (RTK a) | RK3 a a (RTK a)
```

that is, the new type constructor $RTK a$ has as many constructors as $T a$ and $K a$ together. The constructors need to be renamed and their type appropriately changed. This part of the transformation is simple. The next step is to redefine every function in terms of $RTK a$.

The mutually recursive group of $map^T$ and $map^K$:

\[
\begin{align*}
map^T f (T1 x) &= T1 (f x) \\
map^T f (T2 k) &= T2 (map^K f k) \\
map^T f (T3 t) &= T3 (map^T f t) \\
map^K f K1 &= K1 \\
map^K f (K2 x t) &= K2 (f x) (map^T f t) \\
map^K f (K3 x y k) &= K3 (f x) (f y) (map^K f k)
\end{align*}
\]

(becomes)

```
map^{TK} f (RT1 x) = RT1 (f x)
map^{TK} f (RT2 k) = RT2 (map^{TK} f k)
map^{TK} f (RT3 t) = RT3 (map^{TK} f t)
map^{TK} f RK1 = RK1
map^{TK} f (RK2 x t) = RK2 (f x) (map^{TK} f t)
map^{TK} f (RK3 x y k) = RK3 (f x) (f y) (map^{TK} f k)
```

While this is not too complicated either, it does involve a lot of work (i.e. all the functions
need to be transformed), which could unduly increase compilation times. So, instead of transforming to single recursion, buildify and catify, and going back to mutual recursion we leave recursion as it is.

In what follows, the order of presentation — deriving maps, catas, Cata-Core rules, buildify, catify — is the same as in the first-order case.

**Definition 5.1 (Fusible datatype)** Regular, polynomial and non-recursive or self-recursive or mutually recursive (groups of) datatypes are fusible. All other datatypes are considered not fusible.

Our starting point, just as in the first-order case is the datatype declaration. In the most general case, the syntax of a group of $m$ mutually recursive datatypes is given in Equation 4.1.

First we introduce new notation. As we mentioned above, for mutually recursive datatypes, the corresponding catamorphisms and maps are also mutually recursive. To denote this, we put all the datatypes of the recursive group in the superscript. This makes it clear that the cata under consideration belongs to a datatype which is part of a mutually recursive group. It does not tell us however, which datatype it applies to. Therefore we add one additional piece of information to the superscript: $\text{cata}^{\{T_1 \bar{\alpha} + \ldots + T_n \bar{\alpha}\}}$. $T_j$ will stand for the catamorphism which reduces a data structure of type $T_j$. Sometimes for convenience, we will use the notation $\text{cata}^{\bar{T}}$. If we wanted to be overly precise, we could repeat the type variables $\bar{\alpha}$ for $T_j$, as in $\text{cata}^{\bar{T}, T_j \bar{\alpha}}$, but we will refrain from doing so. We could even write $\text{cata}^{T \bar{\alpha}, T_j \bar{\alpha}}$ to emphasise that $T$ is a set of type constructors that can have more than one type argument $\bar{\alpha}$.

### 5.2.1 Deriving maps

The process is very similar to that of the first-order case. Additional superscripts are used for exactly the same purpose as in the case of catas. For a set of mutually recursive datatypes we generate the following code:

$$\text{map}^{T, T_1} = \Lambda \bar{\alpha} \Lambda \bar{\beta} \lambda \bar{f}. \lambda t.$$  

\[
\text{case } t \text{ of } \\
\{ T_{1,i} \bar{v} \rightarrow T_{1,i} \bar{\beta} (M \bar{T}, T_1 \bar{f} (\text{map}^{\bar{T}, T_1 \bar{\alpha} \bar{\beta} \bar{f}} \bar{v})) \}_{i=1}^n \\
\vdots \\
\text{map}^{\bar{T}, T_m} = \Lambda \bar{\alpha} \Lambda \bar{\beta} \lambda \bar{f}. \lambda t. \\
\text{case } t \text{ of } \\
\{ T_{m,i} \bar{v} \rightarrow T_{m,i} \bar{\beta} (M \bar{T}, T_m \bar{f} (\text{map}^{\bar{T}, T_m \bar{\alpha} \bar{\beta} \bar{f}} \bar{v})) \}_{i=1}^n 
\]
5.2. MUTUALLY RECURSIVE DATATYPES

\( \bar{f} \), the first argument to the functor \( M \), contains the functions which rewrite the individual type variables in \( \bar{\alpha} \), so by construction there are as many functions in \( \bar{f} \) as many type variables the group has. The second argument is the set of maps being generated: the left hand sides above.

For simplicity we generate maps in their natural form. We could instead generate the build-cata form of maps directly, but that is not worth the trouble. Catify (Section 5.2.5) and buildify (Section 5.2.4) will transform these appropriately.

The extension of \( M \) to the mutually recursive datatype case is similar to that of extending \( E \): instead of one function, there is a group of functions and the clause which checks if the current type constructor is in the recursive group, applies the appropriate map. We employ set comprehension notation, with its standard meaning, to pick the right \( f_i \) and \( g_i \).

\[
M^{\bar{T}, \bar{\alpha}}, \bar{f} \bar{g} \bar{v} = \mathcal{M}^{\bar{T}, \bar{\alpha}, \bar{f} \bar{g} \langle \text{typeOf } v \rangle} \quad \text{v}
\]

where

\[
\mathcal{M}^{\bar{T}, \bar{\alpha}, \bar{f} \bar{g} \langle \text{primitive} \rangle} = \lambda x.x
\]

\[
\mathcal{M}^{\bar{T}, \bar{\alpha}, \bar{f} \bar{g} \langle \alpha \rangle} = \lambda x.\{f_i \mid \text{sourceTypeOf } (f_i) = \alpha \land i \in \{1 \ldots n\}\}
\]

\[
\mathcal{M}^{\bar{T}, \bar{\alpha}, \bar{f} \bar{g} \langle \text{tyConOf } g \rangle} = \lambda x.\{g_i \mid \text{tyConOf } (g) = T \land i \in \{1 \ldots m\}\}
\]

\[
\mathcal{M}^{\bar{T}, \bar{\alpha}, \bar{f} \bar{g} \langle K \rangle} = \lambda x.\text{map}^{K \bar{\alpha}}(\text{tyVarsOf}(\text{sourceTypeOf } g_1))(\text{tyVarsOf}(\text{targetTypeOf } g_1)) (\mathcal{M}^{\bar{T}, \bar{\alpha}, \bar{f} \bar{g} \langle \tau \rangle}) \quad x
\]

The index in the second clause goes to \( n \) because there can be \( n \) type variables, while the index in the third clause goes to \( m \) because there are \( m \) types in the group. The second clause applies when \( M \) is applied to a type variable: we select the appropriate \( f \) to rewrite the given occurrence. The third clause applies when \( M \) is applied to a type constructor within the mutually recursive group. Clause four applies otherwise. It is possible to combine these last two clauses, at the expense of some notational difficulty.

Clause four perhaps deserves some explanation. Assume that the mutually recursive group consists of three types \( T_1 \), \( T_2 \) and \( T_3 \), all three quantified over the same set of type variables \( \alpha \) and \( \beta \). Assume furthermore, that one of the data constructors refers to a fourth type, say \( K \) with two type arguments and its map\(^K\) has already been derived. map\(^K\) has the following type:

\[
\text{map}^K :: \forall \alpha \beta \gamma \delta. (\alpha \to \gamma) \to (\beta \to \delta) \to K \alpha \beta \to K \gamma \delta
\]

Applications of \( \text{map}^K \), when applied to functions of type \((\alpha \to \gamma)\) and \((\beta \to \delta)\), rewrite data structures of type \( K\alpha\beta \) to data structures of type \( K\gamma\delta \). \( \bar{g} \) typically consists of
functions of the form: $\text{map}^{\bar{T} \bar{\alpha}, T_1} \alpha_1 \alpha_2 \alpha_3 \alpha_4 f_1 f_2, f_1$ and $f_2$ having types $(\alpha_1 \rightarrow \alpha_3)$ and $(\alpha_2 \rightarrow \alpha_4)$ respectively. In this example, there are three functions of this form, one for each member of the mutually recursive group. The type of these functions therefore is $T_1 :: T_1 \alpha_1 \alpha_2 \rightarrow T_1 \alpha_3 \alpha_4$ and so on for $T_2$ and $T_3$. Notice that only the type constructor differs in the three cases. The type arguments at which they are instantiated at are the same! This explains why is it enough to take $\text{tyVarsOf}(\text{targetTypeOf } g_1)$ and $\text{tyVarsOf}(\text{sourceTypeOf } g_1)$. With a slight abuse of the notation we extend $\mathcal{M}$ to apply to a list of types $([\bar{T}])$, which explains the ‘bar’ over $\text{tyVarsOf}(\text{targetTypeOf } g_1)$: the sources (and the targets) need to be repeated as many times as type variables $K$ has.

5.2.2 Deriving catas

For a mutually recursive group of datatypes we define the also mutually recursive group of catamorphisms as follows:

$$\text{cata}^{\bar{T}, T_1} = \Lambda \bar{\alpha} \Lambda \bar{\rho} \lambda \bar{c}_1 \ldots \bar{c}_m \lambda t.
  \text{case } t \text{ of }
  \{ T_1, i \bar{v} \rightarrow c_1, i (E^{\bar{T}} (\text{cata}^{\bar{T} \bar{\alpha} \bar{\rho} \bar{c}_1 \ldots \bar{c}_m} \bar{v}) \})_{i=1}^{n_1}
  \vdots 
  \text{cata}^{\bar{T}, T_m} = \Lambda \bar{\alpha} \Lambda \bar{\rho} \lambda \bar{c}_1 \ldots \bar{c}_m \lambda t.
  \text{case } t \text{ of }
  \{ T_m, i \bar{v} \rightarrow c_m, i (E^{\bar{T}} (\text{cata}^{\bar{T} \bar{\alpha} \bar{\rho} \bar{c}_1 \ldots \bar{c}_m} \bar{v}) \})_{i=1}^{n_m}
$$

It is interesting to note, that the entire group is quantified over the same set of type variables, and the argument to $E$ is now a group, not just a single function. We denoted this by dropping the second superscript in $E^{\bar{T}}$. $\text{cata}^{\bar{T} \bar{\alpha}}$ stands for $\text{cata}^{\bar{T} \bar{\alpha}, T_1} \ldots \text{cata}^{\bar{T} \bar{\alpha}, T_n}$. $\bar{\rho}$ has exactly $m$ type variables. Every catamorphism in the recursive group takes as argument one function for each constructor of the mutually recursive group: if there are $n$ type constructors in the group and $\text{NumOfConstrs}(T)$ denotes the number of constructors $T$ has, then $\text{cata}^{\bar{T} \bar{\alpha}, T_m}$ will have $\sum_{i=0}^{n} \text{NumOfConstrs}(T_i)$ arguments.

We also need to give a definition for $E$:

$$E^{\bar{T}} \bar{g} \bar{v} = E^{\bar{T}} \bar{g} (\text{typeOf } \bar{v}) \bar{v}$$
5.2. MUTUALLY RECURSIVE DATATYPES

\begin{align*}
\langle \text{cata of case rule} \rangle & \quad \text{cata}^{T,T_i,\bar{\rho}^c_1 \ldots \bar{\tau}_n} (\text{case } \text{Expr} \text{ of } \{ C \bar{v} \rightarrow e \}) \rightarrow \text{case } \text{Expr} \text{ of } \{ C \bar{v} \rightarrow \text{cata}^{T,T_i,\bar{\rho}^c_1 \ldots \bar{\tau}_n} e \} \\
\langle \text{cata of known constructor rule} \rangle & \quad \text{cata}^{T,T_i,\bar{\rho}^c_1 \ldots \bar{\tau}_n} (C_i \bar{v}) \rightarrow c_i (\text{E}^{T,T_i} \bar{\rho}_f (\text{cata}^{T,T_i,\bar{\rho}^c_1 \ldots \bar{\tau}_n} \bar{v})) \\
\langle \text{cata-build rule} \rangle & \quad \text{cata}^{T,T_i,\bar{\rho}^c_1 \ldots \bar{\tau}_n} (\text{build}^{T,T_i,\bar{\rho} f}) \rightarrow f \bar{\rho}_f \bar{\tau}_1 \ldots \bar{\tau}_n \\
\langle \text{cata of error rule} \rangle & \quad \text{cata}^{T,T_i,\bar{\rho}^c_1 \ldots \bar{\tau}_n} \text{ error} \rightarrow \text{error}
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.1}
\caption{Cata-Core rules in the presence of mutually recursive datatypes}
\end{figure}

\textbf{where}
\begin{align*}
\text{E}^{T} \bar{g} \llbracket \text{primitive type} \rrbracket & = \lambda x.x \\
\text{E}^{T} \bar{g} \llbracket \alpha \rrbracket & = \lambda x.x \\
\text{E}^{T} \bar{g} \llbracket T \bar{\alpha} \rrbracket & = \lambda x.g_i x, \text{ if } T = T_i \\
\text{E}^{T} \bar{g} \llbracket K \bar{\tau} \rrbracket & = \lambda x.\text{map}^{K\bar{\rho}} (\text{sourceTypeOf } g) \\
& \quad \quad \quad (\text{targetTypeOf } g) \\
& \quad \quad \quad (\text{E}^{T} \bar{g} \llbracket \bar{\tau} \rrbracket) \\
& \quad \quad \quad x
\end{align*}

The third clause of \( \text{E} \) selects the appropriate \text{cata} from the mutually recursive group. \( \bar{g} \) typically consists of other catas from the mutually recursive group, all applied to the type arguments and value arguments, except the one which stands for the data structure being traversed.

5.2.3 New Cata-Core rules

There is no fundamental change in the rules from the original rules given in Figure 4.2, apart from the extra superscripts. The definition of \text{build} does change to

\[ \text{build}^{T,T_i} g = g \text{ Constrs}(T_1) \ldots \text{ Constrs}(T_n) \]

which is reflected in the \text{cata-build} rule: \text{build} now applies its argument, \( g \), to \textit{all} the constructors of the mutually recursive group. The extended rules are shown in Figure 5.1.
5.2. MUTUALLY RECURSIVE DATA TYPES

5.2.4 Buildify

The algorithm is the same as in the higher-order case, except that the builds are introduced simultaneously and the syntactic check involves checking for the occurrence of any 'radioactive' cata within the recursive group. We only give the rewrite step.

1. Rewrite each group of functions, which produces a fusible result according to:

\[ f_1 :: \forall \bar{\alpha}. \sigma \rightarrow T_1 \bar{\tau} \]
\[ f_1 = \Lambda \bar{\alpha}. \lambda \bar{v}. e_1 \]
\[ \vdots \]
\[ f_n :: \forall \bar{\alpha}. \sigma \rightarrow T_n \bar{\tau} \]
\[ f_n = \Lambda \bar{\alpha}. \lambda \bar{v}. e_n \]

\[ \Rightarrow \]

- The wrappers

\[ f_1 :: \forall \bar{\alpha}. \sigma \rightarrow T_1 \bar{\tau} \]
\[ f_1 = \Lambda \bar{\alpha}. \lambda \bar{v}. \text{build}_{T_1} (f'_1 \bar{\alpha} \bar{v}) \]
\[ \vdots \]
\[ f_n :: \forall \bar{\alpha}. \sigma \rightarrow T_n \bar{\tau} \]
\[ f_n = \Lambda \bar{\alpha}. \lambda \bar{v}. \text{build}_{T_n} (f'_n \bar{\alpha} \bar{v}) \]

- The workers

\[ f'_1 :: \forall \bar{\alpha}. \sigma \rightarrow (\forall \bar{\rho}. \text{monoConstrs}(T_1 \bar{\tau})[\rho_1/T_1 \bar{\tau}] \rightarrow \]
\[ \vdots \]
\[ \text{monoConstrs}(T_n \bar{\tau})[\rho_n/T_n \bar{\tau}] \rightarrow \rho_1) \]
\[ f'_1 = \Lambda \bar{\alpha}. \lambda \bar{v}. \Lambda \bar{\rho}. \lambda \bar{c}_1 \ldots \lambda \bar{c}_n. \text{cata}_{T_1} \bar{\tau} \bar{\rho} \bar{c}_1 \ldots \bar{c}_n e_1 \]
\[ \vdots \]
\[ f'_n :: \forall \bar{\alpha}. \sigma \rightarrow (\forall \bar{\rho}. \text{monoConstrs}(T_1 \bar{\tau})[\rho_1/T_1 \bar{\tau}] \rightarrow \]
\[ \vdots \]
\[ \text{monoConstrs}(T_n \bar{\tau})[\rho_n/T_n \bar{\tau}] \rightarrow \rho_n) \]
\[ f'_n = \Lambda \bar{\alpha}. \lambda \bar{v}. \Lambda \bar{\rho}. \lambda \bar{c}_1 \ldots \lambda \bar{c}_n. \text{cata}_{T_n} \bar{\tau} \bar{\rho} \bar{c}_1 \ldots \bar{c}_n e_n \]

Remarks we made in Section 4.5.4 regarding arguments and type variables all apply here. The extra type variables \( \bar{\rho} \) and abstracted constructors \( \bar{c}_1 \ldots \bar{c}_n \) to the workers are a consequence of catamorphisms being mutually recursive. We were a bit sloppy with the notation: \( \bar{v} \)'s do not necessarily denote the same arguments in the different workers, nor need \( \bar{\alpha} \)'s be the same.
5.2. MUTUALLY RECURSIVE DATATYPES

5.2.5 Catify

1. Rewrite each group of functions, which consumes a fusible argument, according to the following rewrite rule

\[
\begin{align*}
  f_1 &:: \forall \alpha. T_1 \tau \rightarrow \sigma \\
  f_1 &= \Lambda \alpha. \lambda t \, \text{cata} \, T_1 \, \tau \\
  \vdots \\
  f_n &:: \forall \alpha. T_n \tau \rightarrow \sigma \\
  f_n &= \Lambda \alpha. \lambda t \, \text{cata} \, T_n \, \tau
\end{align*}
\]

\[\Rightarrow \]

— The wrappers

\[
\begin{align*}
  f_1 &:: \forall \alpha. T_1 \tau \rightarrow \sigma \\
  f_1 &= \Lambda \alpha. \lambda t \, \text{cata} \, T_1 \, \tau \\
    &= (T_1 \, \tau) \ldots (T_n \, \tau) \quad \text{— Type arguments} \\
    &= (f_{C_1}^{T_1} \, \alpha) \ldots (f_{C_m}^{T_1} \, \alpha) \quad \text{— Constructors of } T_1 \\
    \vdots \\
    &\text{— The workers} \\
  f_n &:: \forall \alpha. T_n \tau \rightarrow \sigma \\
  f_n &= \Lambda \alpha. \lambda t \, \text{cata} \, T_n \, \tau \\
    &= (T_1 \, \tau) \ldots (T_n \, \tau) \quad \text{— Type arguments} \\
    &= (f_{C_1}^{T_n} \, \alpha) \ldots (f_{C_m}^{T_n} \, \alpha) \quad \text{— Constructors of } T_n
\end{align*}
\]

The comments we made when we first gave this algorithm also apply here (see page 60.) There are as many workers \( f_{C_i}^{T_i} \) as constructors the of entire mutually recursive group and as many type arguments \( T_i \tau \) as type constructors in the group.

2. Simplify the resulting bindings.
3. Construct the rules of a rewrite system on-the-fly and do a second rewriting. We define the rules and study the rewrite system in Section 5.3. These rules will be built up from combinations of $t$’s with $f$ on the left hand sides and $z$’s on the right-hand sides.

4. Traverse the rewritten bindings and check for free occurrences of $t$. The presence of any $t$ denotes failure of the transformation, in which case revert to the original definition of $f$. If no $t$’s occur (the bindings are closed) we succeeded in transforming $f$ to an explicit catamorphic form, so replace the original definition with the result of the previous step.

We continue our example with the previously builtified maps: $map^{(T,K),T}$ and $map^{(T,K),K}$. Originally, they were mutually recursive. After buildify, the wrappers, $map^{(T,K),T}$ and $map^{(T,K),K}$ are not mutually recursive anymore, but the workers are. We leave out the wrappers $map^{(T,K),T}$ and $map^{(T,K),K}$ as they play no role. If we performed static argument transformation we would be in trouble here: mutual recursion would occur within local bindings where calls to the other function would have all the arguments while calls to the local function would have its static arguments dropped. This would not only complicate the definition of the rewrite system, but also Step 1.

Step 1 splits the workers $map^{(T,K),T}$ and $map^{(T,K),K}$ into two wrappers:

$$map^{#(T,K),T} = \Lambda \alpha \beta \lambda t f. \Lambda \tau \rho \lambda t_1 t_2 k_1 k_2.$$  
$$cata^{(T,K),T} \beta \tau \rho  
  (map^{#(T,K),T}_{T_1}) \beta  
  (map^{#(T,K),T}_{T_2}) \beta 
  (map^{#(T,K),K}_{K_1}) \beta 
  (map^{#(T,K),K}_{K_2}) \beta 
  t f \tau \rho t_1 t_2 k_1 k_2$$

$$map^{#(T,K),K} = \Lambda \alpha \beta \lambda k f. \Lambda \tau \rho \lambda t_1 t_2 k_1 k_2.$$  
$$cata^{(T,K),K} \beta \tau \rho  
  (map^{#(T,K),T}_{T_1}) \beta  
  (map^{#(T,K),T}_{T_2}) \beta 
  (map^{#(T,K),K}_{K_1}) \beta 
  (map^{#(T,K),K}_{K_2}) \beta 
  k f \tau \rho t_1 t_2 k_1 k_2$$

and four workers:

$$map^{#(T,K),T}_{T_1} = \Lambda \alpha \beta \lambda z_1 \lambda f. \Lambda \tau \rho \lambda t_1 t_2 k_1 k_2.$$
5.2. MUTUALLY RECURSIVE DATATYPES

\[
\text{let } l' = \lambda t. \text{case } t \text{ of}
\]
\[
T1 a \rightarrow t_1 (f \ a)
\]
\[
T2 a k \rightarrow t_2 (f \ a) (\text{map}^{\{T,K\},K} \alpha \beta k f \beta \tau \rho t_1 t_2 k_1 k_2)
\]

\[
\text{in } l' \ (T1 \ ta)
\]
\[
\text{map}^{\{T,K\},T}_{T_2} = \lambda \alpha \beta. \lambda z_1 z_2. \lambda f. \lambda \tau \rho. \lambda t_1 t_2 k_1 k_2.
\]
\[
\text{let } l' = \lambda t. \text{case } t \text{ of}
\]
\[
T1 a \rightarrow t_1 (f \ a)
\]
\[
T2 a k \rightarrow t_2 (f \ a) (\text{map}^{\{T,K\},K} \alpha \beta k f \beta \tau \rho t_1 t_2 k_1 k_2)
\]

\[
\text{in } l' \ (T2 \ ta \ tk)
\]
\[
\text{map}^{\{T,K\},K}_{K_1} = \lambda \alpha \beta. \lambda z_1 z_2. \lambda f. \lambda \tau \rho. \lambda t_1 t_2 k_1 k_2.
\]
\[
\text{let } l' = \lambda k. \text{case } k \text{ of}
\]
\[
K1 \rightarrow k_1
\]
\[
K2 t \rightarrow k_2 (\text{map}^{\{T,K\},T} \alpha \beta t f \beta \tau \rho t_1 t_2 k_1 k_2)
\]

\[
\text{in } l' \ K1
\]
\[
\text{map}^{\{T,K\},K}_{K_2} = \lambda \alpha \beta. \lambda z_1. \lambda f. \lambda \tau \rho. \lambda t_1 t_2 k_1 k_2.
\]
\[
\text{let } l' = \lambda k. \text{case } k \text{ of}
\]
\[
K1 \rightarrow k_1
\]
\[
K2 t \rightarrow k_2 (\text{map}^{\{T,K\},T} \alpha \beta t f \beta \tau \rho t_1 t_2 k_1 k_2)
\]

\[
\text{in } l' \ (K2 \ tt)
\]

The local functions, which we denoted \(l'\), will get inlined and the ”case of known constructor” rule applies. We get:

\[
\text{map}^{\{T,K\},T}_{T_1} = \lambda \alpha \beta. \lambda z_1. \lambda f. \lambda \tau \rho. \lambda t_1 t_2 k_1 k_2. \ k_1 (f \ ta)
\]
\[
\text{map}^{\{T,K\},T}_{T_2} = \lambda \alpha \beta. \lambda z_2. \lambda f. \lambda \tau \rho. \lambda t_1 t_2 k_1 k_2.
\]
\[
t_2 (f \ ta) (\text{map}^{\{T,K\},K} \alpha \beta tk f \beta \tau \rho t_1 t_2 k_1 k_2)
\]
\[
\text{map}^{\{T,K\},K}_{K_1} = \lambda \alpha \beta. \lambda z_1 \lambda f. \lambda \tau \rho. \lambda t_1 t_2 k_1 k_2.
\]
\[
\text{map}^{\{T,K\},K}_{K_2} = \lambda \alpha \beta. \lambda z_1 \lambda f. \lambda \tau \rho. \lambda t_1 t_2 k_1 k_2.
\]
\[
k_2 (\text{map}^{\{T,K\},T} \alpha \beta tt f \beta \tau \rho t_1 t_2 k_1 k_2)
\]
5.3. THE DYNAMIC REWRITE SYSTEM

Rewriting with the rules generated by the rewrite system in Section 5.3 finally gives:

\[ \text{map}^{\{T,K\},T} = \lambda \alpha \beta, \lambda t. f. \lambda \tau, \lambda t_1 t_2 k_1 k_2. \]
\[ \text{cata}^{\{T,K\},T} \beta \tau \rho \]
\[ (\text{map}^{\{T,K\},T} \beta) (\text{map}^{\{T,K\},T} \beta) \]
\[ (\text{map}^{\{T,K\},K} \beta) (\text{map}^{\{T,K\},K} \beta) \]
\[ t f \tau \rho t_1 t_2 k_1 k_2 \]

\[ \text{map}^{\{T,K\},K} = \lambda \alpha \beta, \lambda k. f. \lambda \tau, \lambda t_1 t_2 k_1 k_2. \]
\[ \text{cata}^{\{T,K\},K} \beta \tau \rho \]
\[ (\text{map}^{\{T,K\},T} \beta) (\text{map}^{\{T,K\},T} \beta) \]
\[ (\text{map}^{\{T,K\},K} \beta) (\text{map}^{\{T,K\},K} \beta) \]
\[ k f \tau \rho t_1 t_2 k_1 k_2 \]

All the pre-recursion variables and recursive calls have been eliminated, the transformation is successful, therefore we replace the original definition of \( \text{map}^{\{T,K\},T} \) and \( \text{map}^{\{T,K\},K} \) with the ones above.

5.3 The dynamic rewrite system

The previous three sections detailing the first-order, the higher-order and the mutually recursive case referred to the rewrite system which we study in this section. The reason to share the definition and study of properties is that the rewrite system does not depend on the transformations, provided that we define it the right way: that is including all the extensions. The idea behind the rewrite system has been explained very informally in Section 4.1 and it is related to theory on Page 22.

5.3.1 The details

Recall that the purpose of the rewrite system is to eliminate combinations of the function being transformed with pre-recursion variables (we denoted them \( t \)), in favour of the new appropriately typed variables (\( z \)).

**Definition 5.2 (Rewrite System)** Given \( \bar{g} \), the functions being transformed, \( \bar{T} \), appropriately typed — with respect to the constructor they belong to — variables, and \( \bar{z} \) fresh,
appropriately typed variables, we define the rules of the rewrite system to be:

\[ \mathcal{R} = \{ E^{\overline{T}, T_i} \; \overline{g} \; \overline{t_i} \rightarrow E^{\overline{T}, T_i} \; \text{id} \; \overline{z_i} \mid i \in \text{Constrs}(T_j) \land j \in \{ \overline{T} \overline{\alpha} \} \} \]

This set of rewrite rules are valid only in the body of the function being transformed. Typing of \( t_i \) and \( z_i \) is such that the resulting expressions are well typed.

This is rather compact definition! It generates a set of rewrite rules, which we use on a per-function basis. \( \overline{g} \) is(are) the function(s) being transformed. In the case of a group of mutually recursive datatypes, functions acting over any of the types will be mutually recursive. A self recursive datatype will have one function in \( \overline{g} \). Note that, \( j \) varies over type constructors (members of a possibly mutually recursive group \( \overline{T} \overline{\alpha} \)), while \( i \) varies over the data constructors of the given type constructor. We also rely on the slight abuse of notation we introduced on page 52: \( E \) is applied to a list of variables instead of a single variable. Note, that as far as the rewrite system is concerned, the vectors of new variables \( \overline{t_i} \) and \( \overline{z_i} \) are treated as literals, and not as term rewriting variables.

The terminology used in the following is standard and follows [Klo96].

**Definition 5.3** A TRS is **non-erasing** if in every rule \( t \rightarrow s \) the same variables occur in \( t \) and in \( s \).

**Theorem 5.1** Every orthogonal TRS is confluent.

For the reference to the proof see Klop [Klo96].

**Theorem 5.2** The rewrite system generated by Definition 5.2 is confluent.

**Proof 5.1 (Confluence)** First, we observe that the rules generated by Definition 5.2 form a ground TRS (no term rewriting variables, only function symbols and constants). Left-linearity of the rules and the absence of critical-pairs is an easy consequence. By definition, a TRS where all the rules are left-linear and there are no critical pairs is orthogonal, therefore the dynamic rewrite system generated by Definition 5.2 is orthogonal. From Theorem 5.1 confluence follows. \( \square \)

For termination it is much easier argue informally: we are rewriting a finite tree with rules which have no term rewrite variables (i.e. ground rules). The RHS of each rule is fully reduced, that is once a subtree is rewritten no other rule applies to it. Consequently, visiting each node of the tree once and performing a rewrite step if there is an applicable one will rewrite the tree completely.
5.4. STANDARDISING ARGUMENT ORDERING

To spell this informal argument out in detail we need to recall a few definitions from [Der93].

Definition 5.4 Let $\tau_0, \ldots, \tau_{i-1} (i \geq 0)$ be monotonic homomorphisms, all but possibly $\tau_{i-1}$ strict, and let $\tau_i, \ldots, \tau_k$ be any other kinds of termination functions. The induced path ordering $\succ$ is as follows:

$$s = f(s_1, \ldots, s_m) \succ g(t_1, \ldots, t_n) = t$$

if either of the following hold:

1. $s_i \succeq t$ for some $s_i$, $i = 1, \ldots, m$; or
2. $s \succ t_1, \ldots, t_n$ and $\langle \tau_1 s, \ldots, \tau_k s \rangle$ is lexicographically greater than or equal to $\langle \tau_1 t, \ldots, \tau_k t \rangle$, where function symbols are compared according to their precedence, homomorphic images are compared in the corresponding well-founded ordering, and subterms are compared recursively in $\succ$.

Theorem 5.3 A rewrite system terminates if $l\sigma \succ r\sigma$ in a path ordering $\succ$ for all rules $l \rightarrow r$ and substitutions $\sigma$, and also $\tau(l\sigma) = \tau(r\sigma)$ for each of the non-monotonic homomorphisms among its termination functions.

Theorem 5.4 All the rules generated by Definition 5.2 are size decreasing, if nullary function symbols (constants) are compared such that $y_i < z_i$, for all $i$.

Proof 5.2 By induction on 5.2.

Proof 5.3 (Termination) To prove that the rewrite system given in Definition 5.2 terminates, let the termination function be the size (strictly monotonic) of the term and note that all of the rules show a decrease for $\succ$ by virtue of clause (1) and Theorem 5.4. Termination follows by application of Theorem 5.3.

One technical question remains open. What sort of reduction strategy can we use to implement the rewrite system? Fortunately, the answer is easy. The combination of orthogonality and the property that all rules are non-erasing (since there are no variables) guarantees that either leftmost-innermost or leftmost-outermost strategy will work.

5.4 Standardising argument ordering

The need for standardising argument ordering has been explained in Section 5.1.4. The idea is rather simple: we transform every function which
• has more than one argument and
• has at least one fusible argument and
• the function is strict in the fusible argument

to a form where the fusible argument is the first argument to the function. We do this by
splitting the function into a wrapper and a worker [PJL91a]. The wrapper has the original
argument ordering while the worker has the 'better' one. We also mark the wrapper as
Inline — this will encourage the Core simplifier to inline the small wrapper at call sites —
which ensures that the wrapper is also inlined into its own worker. This way the worker is
recursive and has the 'better' argument ordering.

The transformation is formalised as follows: we rewrite every function according to the
following rewrite rule ($\bar{v}$ stands for all the arguments to $f$, i.e. body is not a function):

$$f = \Lambda\alpha.\lambda\bar{v}.\text{body}$$

$$\Rightarrow$$

$$(\bar{v}' \text{ denotes the better ordering of arguments})$$

$$f = \Lambda\alpha.\lambda\bar{v}.f' \bar{\bar{v}}'$$

$$f' = \Lambda\alpha.\lambda\bar{v}'.\text{body}$$

Lets see an example for the transformation:

$$\text{mapFilter} :: \forall\alpha\beta. (\alpha \to \text{Bool}) \to [\alpha] \to (\alpha \to \beta) \to [\beta]$$

$$\text{mapFilter} = \Lambda\alpha\beta.\lambda p\ xs\ f.$$ 

$$\text{case}\ xs\ \text{of}$$

$$\ [] \to [\ ]\ \beta$$

$$y : ys \to \text{case}\ p\ y\ \text{of}$$

$$\text{True} \to (:)\ \beta\ (f\ y)\ (\text{mapFilter}\ \alpha\ \beta\ p\ ys\ f)$$

$$\text{False} \to \text{mapFilter}\ \alpha\ \beta\ p\ ys\ f$$

We collect all the explicit binders of the function and rewrite the above definition of
$\text{mapFilter}$ into a worker ($\text{mapFilter}'$) and a wrapper ($\text{mapFilter}^*$).

$$\text{mapFilter}' :: \forall\alpha\beta. (\alpha \to \text{Bool}) \to [\alpha] \to (\alpha \to \beta) \to [\beta]$$

$$\text{mapFilter}' = \Lambda\alpha\beta.\lambda p\ xs\ f.\text{mapFilter}'\ \alpha\ \beta\ xs\ p\ f$$

$$\text{mapFilter}^* :: \forall\alpha\beta. (\alpha \to \text{Bool}) \to [\alpha] \to (\alpha \to \beta) \to [\beta]$$

$$\text{mapFilter}^* = \Lambda\alpha\beta.\lambda xs\ f.$$
5.5. **TWO PRACTICAL ISSUES**

After simplification, the wrapper (marked with a *) is inlined into the body of the worker (and to all the call sites), a few beta reductions happen and we get:

\[
\begin{align*}
\text{mapFilter}^* &: \forall \alpha \beta. (\alpha \to \text{Bool}) \to [\alpha] \to (\alpha \to \beta) \to [\beta] \\
\text{mapFilter}^* &= \Lambda \alpha \beta. \lambda p xs f. \text{mapFilter'} \alpha \beta xs p f \\
\text{mapFilter'} &: \forall \alpha \beta. [\alpha] \to (\alpha \to \text{Bool}) \to (\alpha \to \beta) \to [\beta] \\
\text{mapFilter'} &= \Lambda \alpha \beta. \lambda xs p f.
\end{align*}
\]

\[
\begin{align*}
\text{case } xs \text{ of } \\
\quad [] & \to [] \beta \\
\quad y : ys & \to \text{case } p \ y \text{ of } \\
\quad \quad \text{True } & \to (:) \beta (f \ y) (\text{mapFilter} \alpha \beta p \ ys \ f) \\
\quad \quad \text{False } & \to \text{mapFilter} \alpha \beta p \ ys \ f
\end{align*}
\]

Since the wrapper is inlined at every call site — if the function is exported then both the wrapper and the worker are exported — this transformation does not result in indirections, so it does not degrade performance. The only disadvantage is a minute increase in code size if the function is exported, because the wrapper needs to be kept as well. However, if the function is not exported then at the end of compilation process there are no calls to the wrapper and it is discarded by the occurrence analyser.

### 5.5 Two practical issues

In this section we examine to practical issues related to warm fusion. The first one is warm fusion in the presence of separate compilation, and the second one is the use of warm fusion to remove intermediate lists from one prominent and useful feature of functional languages: list comprehensions.

#### 5.5.1 Separate compilation

In previous sections, we have presented fusion for a large class of datatypes and detailed the necessary transformations to fuse compositions of functions defined in a single module.
Fusion between functions within the same module is well understood and lies on sound theoretical foundations.

Any nontrivial piece of software will however spread over more than one module. Module systems have at least two roles:

1. they allow splitting up projects into manageable pieces, and
2. they enforce a layer of abstraction.

If a module system is only used to exploit benefits of 1, then from the compiler’s point of view there is no difference between definitions in separate modules: the compiler sees all the code, that is all the code defined in all the modules at once. On the other hand, if a module system is used to enforce a layer of abstraction, it can hide information (types, constructors of a type, definitions etc) from other modules. In this case, the compiler can only deal with the module it is instructed to compile. This has the benefit that if a hidden entity changes in module $X$, modules depending on $X$ need not be recompiled, in other words separate compilation is possible. Separate compilation is therefore a Good Thing because it can reduce recompilation times.

Haskell’s module system is defined in Chapter 5 of the Haskell Report [PJH99]. Most of the constructs of the module system (imports, some forms of exports, hiding) do not interfere with the fusion transformation. For example, if a type $T$ is not exported from the module $X$ and no functions over $T$ are exported from $X$, then one can reasonably expect that fusion will happen within the module, but there could be no opportunities for fusion outside the module.

Difficulties arise when a module abstractly (without the constructors) exports a type $T$, which is the typical situation in the case of libraries. For example, an abstract datatype (ADT) for sets could be defined as:

```
module Set where

data Set a = EmptySet | Insert a (Set a)
<Implementation based on lists>

empty :: Set a
empty = EmptySet

insert :: a -> Set a -> Set a
insert x s = Insert x s
<more functions (destructors and predicates) on Sets>

isEmpty :: Set a -> Bool
```
5.5. TWO PRACTICAL ISSUES

\[
\begin{align*}
\text{isEmpty EmptySet} &= \text{True} \\
\text{isEmpty } _- &= \text{False}
\end{align*}
\]

Modules importing \( Set \) cannot construct values of type \( Set \) because they do not have access to the constructors of the type. Since \( Set \) is abstract, the writer of the module is free to change the implementation: modules depending on the \( Set \) module need not be recompiled unless the interface changes. Despite the abstraction, one would like to have fusion to make sure that compositions of functions over the datatype \( Set \) build no intermediate data structures. For example, given the expression \( \text{isEmpty \ (insert 1 \ empty) } \) one would like to use fusion to avoid building the intermediate \( Set \), containing the number 1. In order to make this happen, one would need to export the \( \text{cata}^{\text{Set}} \) and \( \text{build}^{\text{Set}} \) and the wrappers of \( \text{insert}, \text{empty}, \text{isEmpty} \). This would, however break the abstraction barrier and separate compilation because the type of these functions encode the types of the constructors. If the implementor of module \( Set \) changes the implementation but not the signature one would expect that recompilation of modules importing \( Set \) is not necessary, which is not the case if the wrappers and \( \text{cata}^{\text{Set}}, \text{build}^{\text{Set}} \) is exported. To make this argument more concrete consider the following example.

Given the module \( Set \), fusion transformation would derive the following functions:

\[
\begin{align*}
\text{cata}^{\text{Set}} &:: \forall \alpha \rho. \rho \to (\alpha \to \rho \to \rho) \to \text{Set } \alpha \to \rho \\
\text{build}^{\text{Set}} &:: \forall \alpha \forall \rho. (\rho \to (\alpha \to \rho \to \rho) \to \rho) \to \text{Set } \alpha
\end{align*}
\]

According to the Haskell Report these functions would end up in interface files. Now consider a change to the implementation of \( Set \):

\begin{verbatim}
module Set where

data Set a = EmptySet | Insert a (Set a) (Set a) — !!! CHANGED !!!
  (Implementation based on trees)

empty :: Set a
empty = EmptySet

insert :: a -> Set a -> Set a
insert x s = (insertion to a balanced Tree)
  (more functions (destructors and predicates) on Sets)

isEmpty :: Set a -> Bool
isEmpty EmptySet = True
isEmpty _ = False
\end{verbatim}
The interface of the module is not changed, so one could reasonably expect that modules importing \( \textit{Set} \) need not be recompiled. This is not the case, if \( \textit{build}^\textit{Set} \) and \( \textit{cata}^\textit{Set} \) is 'silently' exported to expose opportunities of fusion, because the type of these functions change to:

\[
\begin{align*}
\textit{cata}^\textit{Set} &:: \forall \alpha \rho. (\alpha \rightarrow \rho \rightarrow \rho \rightarrow \rho) \rightarrow \textit{Set} \alpha \rightarrow \rho \\
\textit{build}^\textit{Set} &:: \forall \alpha. \forall \rho. (\rho \rightarrow (\alpha \rightarrow \rho \rightarrow \rho \rightarrow \rho) \rightarrow \rho) \rightarrow \textit{Set} \alpha
\end{align*}
\]

Since the \( \textit{Set} \) is exported abstractly we are not expecting recompilation of modules which import \( \textit{Set} \). But this leads to anomalies because those modules are built with the assumption that the catamorphism and build have types as in Equation 5.2, as opposed to the types shown in Equation 5.3.

The proper type theoretical interpretation of abstract datatypes is that of existential quantification [Car82]. Fusion for existentially quantified datatypes is an unexplored area, therefore, in order to avoid anomalies we will refrain from attempting fusion for abstract datatypes.

Instead, we shall take the following simple, conservative approach:

- \( T \) and all of its constructors are exported: \( \textit{cata}^T \), \( \textit{build}^T \) are both exported. A function \( f \)'s wrappers are exported if the function is exported. For functions which are not exported, at the end of fusion transformation wrappers are inlined to minimise the overhead of extra function calls.

- \( T \) is not exported: \( \textit{cata}^T \), \( \textit{build}^T \) are derived for intramodule fusion but they are not exported. Wrappers are not exported, but at the end of fusion transformation they are inlined so the overhead of extra function calls can be minimised.

### 5.5.2 List comprehensions

List comprehensions are a syntactic feature of Haskell, which can greatly increase the ease with which one can read and write Haskell programs. Since they have such a prominent role it is important that their use is as efficient as it can be. Translating list comprehensions from Haskell to Core has long been studied and several optimal desugaring schemes (see Figure 5.3) have been proposed [Wad87b, Aug87]. The criterion for a translation scheme to be optimal is that only \textit{one cons cell} is used for each element in the result, in other words the translation scheme is such that \textit{no intermediate lists are produced}. Extensions to the basic schemes often include features such as provision for optimising a chain of appended list comprehensions, upholding the optimality criterion.
5.5. TWO PRACTICAL ISSUES

If these translation schemes produce no intermediate lists, the question arises why do list comprehensions need to be discussed in a thesis which deals with the removal of intermediate data structures? The reason is that we would like to ensure that the resulting list can also be avoided. As we shall see, optimal translations are nothing more than applying fusion to the semantics given for clarity.

Consider the following expression:

\[
f n = \text{sum} \ [p \mid p \leftarrow [1..n], \text{odd} p]
\]

which computes the sum of odd numbers between 1 and \(n\). If the semantics of Haskell list comprehensions (see Figure 5.2 and the Haskell Report [PJH99]) were used to translate this to Core, an intermediate list would be produced by the inner list comprehension \([1..n]\) and would be immediately consumed by the traversal (a filter) which applies the predicate \(p\) to each element. This traversal then would build another list which would be consumed by \(\text{sum}\).

The optimal translation scheme, given in Figure 5.3 avoids the first intermediate list, but the second remains.

Gill gives a desugaring scheme [Gil96, page 44], which is optimal for his cheap deforestation technique, and proves it correct with respect to the semantics of list comprehensions (see Figure 5.2). Contrary to his approach we calculate the optimal translation scheme from the semantics by using only one program transformation technique: the technique of warm fusion. We shall use exactly the same steps we advocate in this thesis: we turn functions to explicit build form, then when possible to explicit cata form and will apply the cata-build rule whenever possible.

We shall use the definition of Haskell list comprehensions in Figure 5.2 but for convenience we put a \(TE\) or \(TQ\) in front of untranslated subexpressions. Also for convenience we do not
Figure 5.3 Traditional list comprehension desugaring scheme

use explicit type variables and we drop the superscripts from the cata and the build: it will be understood that we mean \( \text{cata}^{[\alpha]} \) and \( \text{build}^{[\alpha]} \) with their expected types and definitions.
5.5. TWO PRACTICAL ISSUES

\[ \text{cata nil cons (if } \mathcal{T} \mathcal{E} \ [ \mathcal{B} ] \text{ then } \mathcal{T} \mathcal{E} \ [ [ \mathcal{E} \ | \ \mathcal{Q} \mathcal{S} ] ] \text{ else } []) \]

\[ = \text{cata nil cons (case } \mathcal{T} \mathcal{E} \ [ \mathcal{B} ] \text{ of} \]
\[ \quad \text{True } \rightarrow \mathcal{T} \mathcal{E} \ [ [ \mathcal{E} \ | \ \mathcal{Q} \mathcal{S} ] ] \]
\[ \quad \text{False } \rightarrow [] \]

\[ = \text{cata nil cons (case } \mathcal{T} \mathcal{E} \ [ \mathcal{B} ] \text{ of} \]
\[ \quad \text{True } \rightarrow \text{cata nil cons } (\mathcal{T} \mathcal{E} \ [ [ \mathcal{E} \ | \ \mathcal{Q} \mathcal{S} ] ] ) \]
\[ \quad \text{False } \rightarrow \text{cata nil cons } [] \]

\[ = \text{case } \mathcal{T} \mathcal{E} \ [ \mathcal{B} ] \text{ of} \]
\[ \quad \text{True } \rightarrow \text{cata nil cons } (\text{build } (\mathcal{T} \mathcal{Q} \ [ [ \mathcal{E} \ | \ \mathcal{Q} \mathcal{S} ] ] )) \]
\[ \quad \text{False } \rightarrow \text{nil} \]

\[ \text{Case 3. (generator)} \]
\[ \mathcal{T} \mathcal{Q} \ [ [ \mathcal{E} \ | \ P \leftarrow \mathcal{L}, \ \mathcal{Q} \mathcal{S} ] ] \text{ nil cons} \]

\[ = \text{cata nil cons (let} \]
\[ \quad \text{ok } \mathcal{P}' = \mathcal{T} \mathcal{E} \ [ [ \mathcal{E} \ | \ \mathcal{Q} \mathcal{S} ] ] \]
\[ \quad \text{ok } \_ = [] \]
\[ \quad \text{in} \]
\[ \quad \text{concatMap ok } (\mathcal{T} \mathcal{E} \ [ \mathcal{L} ]) \]

\[ = \text{cata nil cons (let} \]
\[ \quad \text{ok } = \lambda \mathcal{P}'. \text{case } \mathcal{P}' \text{ of} \]
\[ \quad \quad \mathcal{P} \rightarrow \mathcal{T} \mathcal{E} \ [ [ \mathcal{E} \ | \ \mathcal{Q} \mathcal{S} ] ] \]
\[ \quad \quad \_ \rightarrow [] \]
\[ \quad \text{in} \]
\[ \quad \text{concatMap ok } (\mathcal{T} \mathcal{E} \ [ \mathcal{L} ]) \]

\[ = \text{cata nil cons (let} \]
\[ \quad \text{ok } = \lambda \mathcal{P}'. \text{build } (\lambda \text{nil' cons'} \cdot \text{case } \mathcal{P}' \text{ of} \]
\[ \quad \quad \mathcal{P} \rightarrow \mathcal{T} \mathcal{Q} \ [ [ \mathcal{E} \ | \ \mathcal{Q} \mathcal{S} ] ] \]
\[ \quad \quad \_ \rightarrow \text{nil'} \text{ nil' cons'} \]
\[ \quad \text{in} \]
\[ \quad \text{concatMap ok } (\mathcal{T} \mathcal{E} \ [ \mathcal{L} ]) \]
\[ \langle \text{cata of let} \rangle \]

\[ = \text{let} \]
\[ \quad \text{ok} = \lambda P'. \text{build} (\lambda \text{nil'} \text{ cons'}). \textbf{case} P' \textbf{ of} \]
\[ \quad P \rightarrow TQ [[E \mid QS]] \text{ nil'} \text{ cons'} \]
\[ \quad \_ \rightarrow \text{nil'} \]
\[ \text{in} \]
\[ \text{cata nil cons (concatMap ok (T}E [[L]]) \rangle \]
\[ \langle \text{build-cata form definition of concatMap} \rangle \]

\[ = \text{let} \]
\[ \quad \text{ok} = \lambda P'. \text{build} (\lambda \text{nil'} \text{ cons'}). \textbf{case} P' \textbf{ of} \]
\[ \quad P \rightarrow TQ [[E \mid QS]] \text{ nil'} \text{ cons'} \]
\[ \quad \_ \rightarrow \text{nil'} \]
\[ \text{in} \]
\[ \text{cata nil cons} \]
\[ \quad ((\lambda f \text{ xs}. \text{build} (\lambda \text{nil'} \text{ cons'}. \text{cata cm}_{n} \text{ cm}_{c} \text{ xs f nil'} \text{ append})) \text{ ok (T}E [[L]]) \rangle \]
\[ \langle \beta \text{ reductions} \rangle \]

\[ = \text{let} \]
\[ \quad \text{ok} = \lambda P'. \text{build} (\lambda \text{nil'} \text{ cons'}). \textbf{case} P' \textbf{ of} \]
\[ \quad P \rightarrow TQ [[E \mid QS]] \text{ nil'} \text{ cons'} \]
\[ \quad \_ \rightarrow \text{nil'} \]
\[ \text{in} \]
\[ \text{cata nil cons (build (\lambda \text{nil'} \text{ cons'}. \text{cata cm}_{n} \text{ cm}_{c} (T}E [[L]]) \text{ ok nil'} \text{ append})) \rangle \]
\[ \langle \text{cata-build rule} \rangle \]

\[ = \text{let} \]
\[ \quad \text{ok} = \lambda P'. \text{build} (\lambda \text{nil'} \text{ cons'}). \textbf{case} P' \textbf{ of} \]
\[ \quad P \rightarrow TQ [[E \mid QS]] \text{ nil'} \text{ cons'} \]
\[ \quad \_ \rightarrow \text{nil'} \]
\[ \text{in} \]
\[ \text{cata cm}_{n} \text{ cm}_{c} (T}E [[L]]) \text{ ok nil append.} \]

\[ \text{Case 4.} \langle \text{let} \rangle \]
\[ TQ [[E \mid \text{let} \ DS, QS]] \text{ nil cons} \]
\[ \langle \text{definition} \rangle \]

\[ = \text{cata nil cons (} \text{let} \]
\[ \quad \DS \]
\[ \text{in} \]
\[ T}E [[E \mid QS ]] \]
\[ \langle \text{case of let} \rangle \]
5.5. TWO PRACTICAL ISSUES

\[\begin{align*}
\text{let} & \quad DS \\
in & \quad \text{cata nil cons} (\mathcal{T} \mathcal{E} \left[ [E | QS] \right]) \\
& \quad \langle\text{cata-build rule}\rangle \\
= & \quad \text{let} \\
in & \quad TQ \left[ [E | QS] \right] \text{ nil cons}.
\end{align*}\]

Cases 1, 2 and 4 are clearly optimal: only one list is built with the abstracted constructors \(\text{nil}\) and \(\text{cons}\). If a good consumer (a function which consumes its argument with a \(\text{cata}\)) is applied to the resulting list the \(\text{cata-build}\) rule applies the intermediate list is not built.

Case 3 is somewhat subtle. In particular, the presence of \(\text{append}\) is worrying. Consider however, that \(L\) is a piece of source program and as such is always finite. Its translation proceeds via the \(\mathcal{T} \mathcal{E}\) scheme resulting in a list valued (if it was not list valued the source program would not be well typed) expression which starts with a \(\text{build}\) (see the definition of \(\mathcal{T} \mathcal{E}\) on Page 93). So the translation of Case 3, produces a chain of applications of \(\text{cata}\) to build. This chain is then reduced via the \(\text{cata-build}\) rule.

To see one example that \(\text{append}\) does indeed disappear consider the translation of the list comprehension below. The example is of course artificially small, but anything longer would fill up many pages.

\[\begin{align*}
\mathcal{T} \mathcal{E} \left[ [x | x \leftarrow [1, 2]] \right] \\
& \quad \langle\text{definition}\rangle \\
= & \quad \text{build} (\lambda \text{nil cons}. TQ \left[ [x | x \leftarrow [1, 2]] \right]) \\
& \quad \langle\text{generator}\rangle \\
= & \quad \text{build} (\lambda \text{nil cons. let} \\
ok & = \lambda x'. \text{build} (\lambda \text{nil'} \text{ cons'}. \text{case} x' \text{ of} \\
x & \rightarrow TQ \left[ [x | ] \right] \text{ nil'} \text{ cons'} \\
- & \rightarrow \text{nil'}) \\
in & \quad \text{cata cm}_{c} \text{ cm}_{c} (\mathcal{T} \mathcal{E} \left[ [1, 2] \right]) \text{ ok nil append} \\
& \quad \langle\text{definition of } TQ[[x]], \text{ and the variable } x \text{ always matches in the case}\rangle \\
= & \quad \text{build} (\lambda \text{nil cons. let} \\
ok & = \lambda x'. \text{build} (\lambda \text{nil'} \text{ cons'}. \text{cons'} x \text{ nil'}) \\
in & \quad \text{cata cm}_{c} \text{ cm}_{c} (\mathcal{T} \mathcal{E} \left[ [1, 2] \right]) \text{ ok nil append} \\
& \quad \langle\text{skipping several steps: } \mathcal{T} \mathcal{E}[[1,2]] \text{ gives}\rangle
\end{align*}\]
=build (\nil \cons. let
  ok = \x'. build (\nil' \cons'. \cons' \x \nil')
  in
  cata cm_n cm_c (build (\n \c 1 (c 2 n)) ok \nil \append)
\langle cata-build rule and beta reductions \rangle
=build (\nil \cons. let
  ok = \x'. build (\nil' \cons'. \cons' \x \nil')
  in
  cm_c 1 (cm_c 2 cm_n) ok \nil \append)
\langle definition: cm_c = \lambda \z zs f n. c(c(f(z))(zs f n c)) and beta reductions \rangle
=build (\nil \cons. let
  ok = \x'. build (\nil' \cons'. \cons' \x \nil')
  in
  append (ok 1) (cm_c 2 cm_n ok \nil \append))
\langle definition: cm_c = \lambda \z zs f n. c(c(f(z))(zs f n c)) again and beta reductions \rangle
=build (\nil \cons. let
  ok = \x'. build (\nil' \cons'. \cons' \x \nil')
  in
  append (ok 1) (append (ok 2) (cm_n \nil \append))
\langle definition: cm_n = \lambda f n. c.n and beta reductions \rangle
=build (\nil \cons. let
  ok = \x'. build (\nil' \cons'. \cons' \x \nil')
  in
  append (ok 1) (append (ok 2) \nil)
\langle definition of append for empty list \rangle
=build (\nil \cons. let
  ok = \x'. build (\nil' \cons'. \cons' \x \nil')
  in
  append (ok 1) (ok 2)
\langle definition of append = \lambda xs ys n. c.cata (cata n c ys) c xs and beta reductions \rangle
\langle definition of ok \rangle
=build (\nil \cons.cata (\nil \cons (build (\nil' \cons'. \cons' 2 \nil')))) cons (ok 1))
\langle cata-build and the definition of ok again \rangle
=build (\nil \cons.cata (\cons 2 \nil) cons (build (\nil' \cons'. \cons' 1 \nil')))\langle cata-build rule \rangle
=build (\nil \cons.cons 1 (\cons 2 \nil)).

Simple examination shows that append is gone from the result, so the list is built optimally. Other cases are completely similar to the example above, but the derivation is rather te-
5.5. TWO PRACTICAL ISSUES

\[
\begin{align*}
T E \,[\,[E \mid QS]\,]\, & \equiv \text{build} \,(T Q \,[\,[E \mid QS]\,]) \\
T Q \,[\,[E \mid ]\,\,n \,c \, & \equiv c \,(T E \,[\,E\,]) \,n \\
T Q \,[\,[E \mid B, \,QS]\,\,n \,c \, & \equiv \text{case} \,T E \,[\,B\,] \,\text{of}
\hspace{1cm} \text{True} \rightarrow T Q \,[\,[E \mid QS]\,\,n \,c \\
\hspace{1cm} \text{False} \rightarrow n
\end{align*}
\]

\[
T Q \,[\,[E \mid P \leftarrow L, \,QS]\,\,n \,c \equiv \text{let}
\hspace{1cm} \text{ok} = \lambda \,P'. \text{build} \,\,(\lambda \,n' \,c'. \,\text{case} \,P' \,\text{of}
\hspace{2cm} P \rightarrow T Q \,[\,[E \mid QS]\,\,n' \,c'
\hspace{3cm} \rightarrow n')
\hspace{1cm} \text{— cm}_n \,\text{and cm}_c \,\text{are part of the build-cata form}
\hspace{1cm} \text{— of concatMap}
\hspace{1cm} \text{cm}_n = \lambda \,f \,\,n \,c.n
\hspace{1cm} \text{cm}_c = \lambda \,z \,\,zs \,\,n \,c. \,c \,(f \,\,z) \,(zs \,\,f \,\,n \,c)
\text{in}
\hspace{1cm} \text{cata cm}_n \,\text{cm}_c \,(T E \,[\,L\,]) \,\text{ok} \,\,n \,\text{append}
\hspace{1cm}
T Q \,[\,[E \mid \text{let} \,DS, \,QS]\,\,n \,c \equiv \text{let}
\hspace{1cm} \,T E \,[\,DS\,]
\hspace{1cm} \text{in}
\hspace{1cm} T Q \,[\,[E \mid QS]\,\,n \,c
\end{align*}
\]

**Figure 5.4** Optimal list comprehension desugaring scheme

The function definitions in \text{build-cata} form we implicitly used in the derivation for \text{map, concat, append} are the same what warm fusion would derive given their naive definition. The reader is invited to verify that this is indeed the case. The rules which are optimal if warm fusion is applied to the translation are summarised in Figure 5.4.
Chapter 6

Measuring Warm Fusion

We devote this chapter to the analysis of the effect of fusion transformation. The general aspects of optimisation are discussed in Chapter 2.

6.1 Measuring warm fusion

In order to allow us to quantify the effect of the different transformations we turn on them one by one.

1. **Control run.** As our control we use a version of the compiler (GHC-4.06) which includes our optimisations but they are totally disabled. To demonstrate that the inclusion of the transformations does not affect compilation times and binary sizes when they are not used, we should include a complete set of numbers gathered from compiling the benchmarks with an unmodified compiler. The reason we do not do this is that there is no significant difference between a modified and unmodified compiler.

2. **Normalised.** Our first set of numbers are aimed to show that the normalisation process does not affect execution speed of the resulting programs, because the extra wrappers get inlined to the call sites. It does affect binary sizes as some of the extra wrappers (the exported ones) need to remain.

3. **Buildified only.** The second set of numbers show how bad the result of buildify is. As we noted earlier, buildify splits functions into two and more importantly adds extra arguments to the recursive workers. The number of extra arguments depends on the number of constructors the result type of the given function has. The importance of reporting the results of buildify only is that if catify is unsuccessful, this gives us a clue how bad things can get.
We expect both total heap allocation and max heap residency to increase considerably as no fusion is taking place.

4. **Catified only.** The same as above for catify. We expect a similar, but even greater increase of both total allocation and max heap residency, because the transformation splits a single function into as many functions as the inductive arguments type has. No fusion is taking place.

5. **Buildify and Catify.** The effect of catifying the already buildified functions. This results not only in splitting the original binding into many workers and wrappers, but also that all the newly introduced catamorphisms are higher-order.

   We expect a great increase of total heap allocation and max heap residency. This is the situation when we transform programs to \texttt{build-cata} form but for some reason fusion is not taking place i.e. these numbers are the ones we get in the worst possible case.

6. **All the transformations and fusion.** How much fusion can gain on the results of the previous runs. The hope is that total allocation and heap residency are both smaller than in the control run. We also expect reduced execution times.

7. **build inlined.** This should improve on the results of the previous test as one level of indirection is eliminated. Still the transformed functions do take the extra arguments.

This leaves us with seven different runs of the compiler for the four sets of benchmarks.

### 6.2 What we want to measure

Having decided on the number of different runs of the compiler, we need to decide what to measure. In choosing the aspects we are trying to quantify we use the following principles:

- The numbers we gather should allow comparison with similar work. In particular we use the very same benchmarks as used in Gill [Gil96] and report almost the same set of data.

- We need measurements which substantiate claims we made earlier in this thesis.

The data we collect can be subdivided into two sets: the first set is about the programs produced by the modified compiler while second is about the compiler itself. The first set allows us to quantify the effect of the transformations, the second provides a clue if the transformations are worthwhile. Both sets affect the user of the modified compiler.
6.3. HOW TO MEASURE IT?

- *Execution speed.* The whole point of performing an optimisation is to improve performance! We measure Unix user time.

- *Total heap allocation.* Because warm fusion is an optimisation technique which eliminates intermediate data structures, we expect it to reduce total heap allocation.

- *Max Heap Residency.* Another measure of memory usage.

- *Binary size.* During the first stage of fusion we often duplicated code (Page 61). Measuring the size of the object files is therefore important.

- *Compilation time.* We would like to demonstrate that our implementation of warm fusion is practical.

For the various sets of benchmarks we also report the minimum, the maximum, and the geometric mean of the above. For a thorough discussion why geometric mean is preferable to arithmetic mean see [FW86].

6.3 How to measure it?

According to the above mentioned second aspect of an optimisation we need to reduce some resource requirement compared to the unoptimised program. But what is an unoptimised program? In GHC there are several levels of optimisations, these can be set with flags to the compiler:

- *Unoptimised.* Fiddling with the compiler switches one can turn off all optimisations which are on by default. This results in horrendously inefficient code.

- The *default* is gotten by simply typing the compiler’s name followed by the programs name. This results in relatively fast compilation, but slow programs.

- *-0* optimised. Compilation takes visibly longer, but the code resulting code is definitely faster. This is the most frequently used level of optimisation.

- *-02* optimised. This is a higher level of optimisation, because it uses analysis and techniques which are not used at previous levels and it is also more aggressive with for example inlining.

There can be several arguments about which level to chose as our control run, but for simplicity we use the third *-0*. As for the version of the compiler it is GHC-4.06. This
differs from the version we used in previous chapters because with the release of GHC-4.00 that one has become obsolete. In fact, GHC 3.03 does not build anymore on newer versions of Linux (as of RedHat 6.0). Luckily this does not affect the transformations in any major way. For the sake of completeness, all the programs were run on a machine with an Intel Celeron 330MHz processor and 128MBytes of memory, running SuSe Linux 6.3.

6.4 A detailed example

In this section we give a full example of how fusion happens. It is full in the sense that code which follows is copied straight out of the compiler’s output and has only been formatted to take up less space. The example is of course artificially small, but anything reasonable would take up just too much space.

We start with the program what the user writes. It uses a user defined datatype, which is the same as the list datatype in the Standard Prelude. The definitions are also from the Prelude.

```haskell
module Main where
import Prelude hiding (map, length, iterate, take)

data List a = Nil | a :+: (List a) deriving (Show, Ord, Eq)
infixr 5 :+:

map f Nil = Nil
map f (x :+: xs) = (f x) :+: (map f xs)

length = foldl' (\n _ -> n + 1) 0
foldl' f a Nil = a
foldl' f a (x :+: xs) = (foldl' f $! f a x) xs

iterate f x = x :+: iterate f (f x)

take 0 _ = Nil
take _ Nil = Nil
take n (x :+: xs) | n>0 = x :+: take (n-1) xs
take _ _ = error "Prelude.take: negative argument"

main = print (length . map (+1) . map (*2) . take 1000 . iterate (+1) $ 1) >> return ()
```

The first step (see Figure 4.1) is deriving the map, the catamorphism, and the build for this datatype. The name of each definition is composed from its functionality (cata, map, build) with the name of the datatype attached to it. So, the map function for the list datatype becomes `map_List`. First, each function’s type is shown, then its body. In between,
6.4. A DETAILED EXAMPLE

[NoDiscard] says that the definition should not be dropped even if it is never referenced in the rest of the program. For cataList, LLS describes the strictness properties of this function: it is lazy in its first two argument, and strict in the third one.

Rec

mapList :: (forall t_x2u5 t_x2u6.(t_x2u5 -> t_x2u6) -> List t_x2u5 -> List t_x2u6)
[NoDiscard]
mapList
  = \ @ t_x2u2 @ t_x2uI f_x2uJ :: (t_x2u2 -> t_x2uI)
    scrut_x2uH :: (List t_x2u2) ->
      case scrut_x2uH of wild_B1 {
        Nil -> $wNil @ t_x2uI;
        _: ^: a_x2u1 b_x2u3 -> $w:^: @ t_x2uI (f_x2uJ a_x2u1)
          (mapList @ t_x2u2 @ t_x2uI f_x2uJ b_x2u3)
      }
    end Rec

Rec

cataList :: (forall t_x2us t_x2ur.t_x2ur -> (t_x2us -> t_x2ur -> t_x2ur) -> List t_x2us -> t_x2ur)
[NoDiscard] __S LLS

cataList
  = \ @ t_x2uh @ t_x2uK nil_x2uL :: t_x2uK
    zczuzc_x2uM :: (t_x2uh -> t_x2uK -> t_x2uK)
    scrut_x2up :: (List t_x2uh) ->
      case scrut_x2up of wild_B1 {
        Nil -> nil_x2uL;
        _: ^: a_x2ug b_x2uj -> zczuzc_x2uM ((\ id_x2uN :: t_x2uh -> id_x2uN) a_x2ug)
          (cataList @ t_x2uh @ t_x2uK nil_x2uL zczuzc_x2uM b_x2uj)
      }
    end Rec

Rec

buildList :: (forall t_x2uw.(forall t_x2uv. t_x2uv -> (t_x2uv -> t_x2uv -> t_x2uv) -> t_x2uw)
  -> List t_x2uw)
[NoDiscard]
buildList
  = \ @ t_x2uw g_x2uu :: (forall t_x2uw.t_x2uw -> (t_x2uw -> t_x2uw -> t_x2uw) -> t_x2uw) ->
    g_x2uu @ (List t_x2uw) ($wNil @ t_x2uw) ($w:^: @ t_x2uw)
end Rec

The implementation uses GHC’s built-in transformation rules. Three rules need to be derived: the cata-build rule, and the two rules for the catamorphism applied to the constructors of the datatype. These are called cata of known constructor rules in the thesis.

"cata/build(List)" __forall {\ @ t_x2us @ t_x2ur a_x2uQ :: t_x2ur b_x2uR :: (t_x2us -> t_x2ur -> t_x2ur)
  c_x2uS :: (forall t_x2us.t_x2us -> (t_x2us -> t_x2us) -> t_x2us)}
  cataList @ t_x2us @ t_x2ur a_x2uQ b_x2uR (buildList @ t_x2us c_x2uS)
  = (c_x2uS @ t_x2ur a_x2uQ b_x2uR)
"cata/Nil" __forall {\ @ t_x2uE @ t_x2uT nil_x2uU :: t_x2uT zczuzc_x2uV :: (t_x2uE -> t_x2uT -> t_x2uT)}
  cataList @ t_x2uE @ t_x2uT nil_x2uU zczuzc_x2uV (\nil_x2uU @ t_x2uE)
  = nil_x2uU ;
6.4. A DETAILED EXAMPLE

"cata":" forall (@ t_x2uE @ t_x2uT nil_x2uU :: t_x2uT zczuzc_x2uV :: (t_x2uE -> t_x2uT -> t_x2uT)
   a_x2uD :: t_x2uE b_x2uG :: (List t_x2uE))
cata_List @ t_x2uE @ t_x2uT nil_x2uU zczuzc_x2uV (forall @ t_x2uE a_x2uD b_x2uG)
   = (zzucz_x2uV a_x2uD c (cata_List @ t_x2uE @ t_x2uT nil_x2uU zczuzc_x2uV b_x2uG))

The normalisation pass generates the function called nmap from the definition of the user
supplied map, and buildify generates wmap. nmap and the original map are not shown because
after normalisation the wrapper is inlined at every call site (in the body of main) and
becomes dead. So wmap is the worker for map and all the wrappers have been eliminated.

Rec {
  wmap :: (forall a b.List a -> (a -> b) -> __u - (forall t_s2CW.t_s2CW -> __u - ((b -> t_s2CW -> t_s2CW) -> t_s2CW)))

  _AL 4
  wmap
  = \ @ a @ b x_s2Cr :: (List a) x_s2Co :: (a -> b) @ t_s2CW
c1_s2CX OneShot :: t_s2CW c2_s2CY OneShot :: (b -> t_s2CW -> t_s2CW) ->
case x_s2Cr of wild_B1 {
   Nil -> c1_s2CX;
   :: x xs -> c2_s2CY (x_s2Co x) (wmap @ a @ b xs x_s2Co @ t_s2CW c1_s2CX c2_s2CY)
}
end Rec }

The same thing happens to the generated map_List function. It’s wrappers however are
not dropped, because we may need them at later stages, i.e. in catify. nmap_List is the worker of map_List, but it becomes a wrapper during buildify. wmap_List is the worker of the generated map.

Rec {
  wmap_List :: (forall t_x2u5 t_x2u6.List t_x2u5 -> (t_x2u5 -> t_x2u6) -> __u - (forall t_s2CS.t_s2CS -> __u - ((t_x2u6 -> t_s2CS -> t_s2CS) -> t_s2CS)))

  _AL 4
  wmap_List
  = \ @ t_x2u5 @ t_x2u6 x_s2Cc :: (List t_x2u5) x_s2C9 :: (t_x2u5 -> t_x2u6)
   @ t_s2CS c1_s2CT OneShot :: t_s2CS c2_s2CU OneShot :: (t_x2u6 -> t_s2CS -> t_s2CS) ->
case x_s2Cc of wild_B1 {
   Nil -> c1_s2CT;
   :: a_x2u1 b_x2u3 ->
c2_s2CU (x_s2C9 a_x2u1)
   (wmap_List @ t_x2u5 @ t_x2u6 b_x2u3 x_s2C9 @ t_s2CS c1_s2CT c2_s2CU)
}
end Rec }

mmap_List :: (forall t_x2u5 t_x2u6.List t_x2u5 -> (t_x2u5 -> t_x2u6) -> List t_x2u6)

  _AL 2
  mmap_List
  = __inline_me (\ @ t_x2u5 @ t_x2u6 x_s2Cc :: (List t_x2u5) x_s2C9 :: (t_x2u5 -> t_x2u6) ->
   build_List @ t_x2u6 (wmap_List @ t_x2u5 @ t_x2u6 x_s2C9))

map_List :: (forall t_x2u5 t_x2u6.(t_x2u5 -> t_x2u6) -> List t_x2u5 -> List t_x2u6)
The same thing happened to the function `take` what happened to `map`. Its wrappers have also been eliminated. `wntake` as expected is a good producer, so the newly introduced `cata` fused with the build of its own wrapper.

```
Rec {
  wntake :: (forall a.List a -> Int
            -> _u = (forall t_s2D4.t_s2D4 -> _u = ((a -> t_s2D4 -> t_s2D4) -> t_s2D4)))
  wntake
  = \ a x_s2CB :: (List a) x_s2Cz :: Int @ t_s2D4
c1_s2D5 OneShot :: t_s2D4 c2_s2D6 OneShot :: (a -> t_s2D4 -> t_s2D4) ->
    case x_s2Cz of wild_B1 { I# ds_d2nA ->
        case ds_d2na of ds_X2nA {0 -> c1_s2D5;
          __DEFAULT ->
            case x_s2CB of wild_X1 {
              Nil -> c1_s2D5;
              `": x xs ->
                case ># ds_X2nA 0 of wild_X2 {
                  True ->
                    c2_s2D6
                    x
                    (let {s_s2BI :: Int#
                      _AL 0
                      s_s2BI
                      = `-# ds_X2nA 1
                        } in wntake \ a xs $(wI# s_s2BI) @ t_s2D4 c1_s2D5 c2_s2D6);
                  False -> __coerce t_s2D4 (error @ (List a) lvl_s2AA)
                    })
                })
            }
          }
        }
      }
end Rec }
```

The original `iterate` function is also a good producer, but it is not affected by normalisation because that is only performed for functions which are good consumers. This explains the name `witerate`: there is no `wniterate` as that would be generated by the normalisation pass.

```
Rec {
  witerate :: (forall a.(a -> a) -> a
               -> _u = (forall t_s2D0.t_s2D0 -> _u = ((a -> t_s2D0 -> t_s2D0) -> t_s2D0)))
  witerate
  
end Rec }
```
6.4. A DETAILED EXAMPLE

```haskell
= \ a f :: (a -> a) x :: a @ t 
  c1 t OneShot :: t c2 t OneShot :: (a -> t -> t) 
  c2 x (witerate @ a f (f x) @ t c1 c2) 
end Rec }
```

Finally, **main**. All the normalised wrappers and the build wrappers are inlined so only calls to the workers remain.

```haskell
main :: (IO ()
[NoDiscard] __AL 1
main
  = __coerce (IO ()
  (\ s5 :: (State RealWorld) ->
   case nfoldl’
     0 Int
     0 Integer
     (build_List
      0 Integer
      (wnmap
       0 Integer
       0 Integer
       (build_List
        0 Integer
        (untake
         0 Integer
         (build_List
          0 Integer
          (witerate
           0 Integer
           (\ s_s2wb :: Integer ->
             PrelNum.+1 s_s2wb lit_a1Yu)
            lit_a1Yu))
          ($wI# 1000))
          ($wI# 0)
          ($wI# 0)
          of w { I# wv ->
            case PrelIO.$whPutStr
```
6.4. A DETAILED EXAMPLE

Static argument transformation (after a pass of simplification) transforms the workers such that there is a new local definition with only one argument. This helps to generate first-order catamorphisms which are more efficient than their higher-order counterparts. In general, SAT drops as many static arguments as possible, but it does not always succeed.

\[
\text{wnmap} :: (\forall a \, b. \text{List } a \to (a \to b) \\
\rightarrow \_u = (\forall t_s2C6, t_s2C7 \rightarrow \_u = ((b \rightarrow t_s2C6 \rightarrow t_s2C7) \rightarrow t_s2C6)))
\]

\[
\text{wnmap} = \\{ a \in b \to x_s2Cr :: (\text{List } a) \times_s2Co :: (a \to b) \to t_s2C6 \}
\]

Notice, that \text{wnmap} has two non-static arguments, so the local function has two arguments. This results in a higher-order catamorphism, which passes its integer argument around.

\[
\text{wntake} :: (\forall a \, \text{List } a \to \text{Int}\to (\forall t_s2D4. \text{t}_s2D4 \rightarrow \_u = ((a \rightarrow t_s2D4 \rightarrow t_s2D4) \rightarrow t_s2D4)))
\]

\[
\text{wntake} = \\{ a \in b \to x_s2CB :: (\text{List } a) \times_s2Cz :: \text{Int} \times t_s2D4 \}
\]
6.4. A DETAILED EXAMPLE

```
:~: x xs ->
  case >! ds_x2nA 0 of wild_x2 (
    True ->
    c2_s2D6
    x
    (let {
      s_s2BI :: Int#
      __AL 0
      s_s2BI
      = -$! ds_x2nA 1
    ) in _sat_s37o xs ($vI# s_s2BI));
    False -> __coerce t_s2D4 (error @ (List a) lvl_s2AA)
  } in _sat_s37o x_s2CB x_s2Cz
```

Catify does two things. First, it transforms the local bindings into a catamorphism, then
it encourages the simplifier to inline the now non-recursive local binding. This results in
the most efficient definitions for `wnmap` and `wntake`. The catamorphism for `wnmap` does not
pass its static argument, the function, around.

```
wnmap :: (forall a b.List a -> (a -> b) -> t_s2CW) -> ((forall t_s2CW t_s2CW) -> t_s2CW)
```

The catamorphism for `wntake` is higher-order (see the second type argument to `cata_List`!)

```
wntake :: (forall a.List a -> Int -> t_s2D4) -> ((forall t_s2D4) -> t_s2D4)
```

```
:~: x xs ->
  case >! ds_x2nA 0 of wild_x2 (
    True ->
    c2_s2D6
    x
    (let {
      s_s2BI :: Int#
      __AL 0
      s_s2BI
      = -$! ds_x2nA 1
    ) in _sat_s37o xs ($vI# s_s2BI));
    False -> __coerce t_s2D4 (error @ (List a) lvl_s2AA)
  } in _sat_s37o x_s2CB x_s2Cz
```
After catify, the already transformed workers \texttt{wnmap} and \texttt{wntake} have become non-recursive and therefore they could be inlined into \texttt{main}. After applying the \texttt{cata-build} rule we get:

\begin{verbatim}
$\texttt{main} :: \text{State\# RealWorld} \to (\text{State\# RealWorld} \to (\text{State\# RealWorld} \to (\text{State\# RealWorld} \to (\text{State\# RealWorld} \to (\text{State\# RealWorld} \to (\text{State\# RealWorld} \to (\text{Int} \to \text{List Integer}) \to \text{Int}) \to \text{List Integer}) \to \text{Int}) \to \text{List Integer}) \to \text{Int}) \to \text{List Integer}) \to \text{Int}) \to \text{List Integer}) \to \text{Int}) \to \text{List Integer}) \to \text{Int}) \to \text{List Integer}) \to \text{Int}) \to \text{List Integer})\)
\end{verbatim}
Notice, that we failed to transform `nfoldl'` so the intermediate list built by `witerate` remains, but all the others, the one between the first `map` and the second disappeared.

Running the original program gives:

```
angel 167 (haskell/andreas): unopt +RTS -Stsderr
unopt +RTS -Stsderr
unopt +RTS -Stsderr
Alloc Collect Live GC GC TUT TUT Page Flts
bytes bytes bytes user elap user elap
1000
143028 0.00 0.00

143,028 bytes allocated in the heap
0 bytes copied during GC
0 collections in generation 0 ( 0.00s)
0 collections in generation 1 ( 0.00s)
1Mb total memory in use

INIT time 0.01s ( 0.00s elapsed)
MUT time 0.00s ( 0.01s elapsed)
GC time 0.00s ( 0.00s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 0.01s ( 0.01s elapsed)

%GC time 0.0% (0.0% elapsed)

Alloc rate 14,302,800 bytes per MUT second

Productivity 0.0% of total user, 0.0% of total elapsed
```

The same program with warm fusion gives:
6.4. A DETAILED EXAMPLE

angel 168 (haskell/andreas): opt +RTS -Sstderr

<table>
<thead>
<tr>
<th>Alloc</th>
<th>Collect</th>
<th>Live</th>
<th>GC</th>
<th>GC</th>
<th>TUT</th>
<th>TUT</th>
<th>Page Flts</th>
</tr>
</thead>
<tbody>
<tr>
<td>bytes</td>
<td>bytes</td>
<td>bytes</td>
<td>user</td>
<td>elap</td>
<td>user</td>
<td>elap</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>106912</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

106,912 bytes allocated in the heap
0 bytes copied during GC

0 collections in generation 0 (0.00s)
0 collections in generation 1 (0.00s)

1 Mb total memory in use

INIT time 0.00s (0.00s elapsed)
MUT time 0.00s (0.00s elapsed)
GC time 0.00s (0.00s elapsed)
EXIT time 0.00s (0.00s elapsed)

%GC time 0.0% (0.0% elapsed)

Alloc rate 1,069,120,000 bytes per MUT second

Productivity 100.0% of total user, 2145388542.0% of total elapsed

The total heap allocation for the unoptimised program is 143,028 bytes, while for the optimised one is 106,912. A good 25% decrease in total allocation.

The importance of static argument transformation can not be stressed enough. Had we not done SAT after buildify, catify would have given:

wnmap :: (forall a b.List a -> (a -> b) -> __u - (forall t_s2C4. t_s2C4 -> __u - ((b -> t_s2C4 -> t_s2C4) -> t_s2C4)))

AL 4

wnmap
= \ @ a @ b ->
cata_list
  \ @ a
  @ ((a -> b) -> __u - (forall t_s2C4. t_s2C4 -> __u - ((b -> t_s2C4 -> t_s2C4) -> t_s2C4)))
  \ x_s2Bx :: (a -> b)
  @ t_s2C4
  cl_s2C5 OneShot :: t_s2C4
  c2_s2C6 OneShot :: (b -> t_s2C4 -> t_s2C4) ->
  c1_s2C5)
  \ r_s2CC :: a
  r_s2CE :: ((a -> b) -> __u - (forall t_s2C4. t_s2C4 -> __u - ((b -> t_s2C4 -> t_s2C4) -> t_s2C4)))
6.4. A DETAILED EXAMPLE

\[
x_{s2Bx} :: (a \to b)
\]
\[
\emptyset \ t_{s2C4}
\]
\[
c_{s2C5} OneShot :: t_{s2C4}
\]
\[
c_{s2C6} OneShot :: (b \to t_{s2C4} \to t_{s2C4}) \to
c_{s2C6} (x_{s2Bx} \ r_{s2CE} x_{s2Bx} \emptyset \ t_{s2C4} c_{s2C5} c_{s2C6})
\]

\[
\text{wntake} :: (\forall a.\text{List}\ a \to \text{Int})
\]
\[
\rightarrow ___u = (\forall all t_{s2C0c}.t_{s2C0c} \rightarrow ___u = ((a \to t_{s2C0c} \to t_{s2C0c}) \to t_{s2C0c}))
\]
\[
\text{wntake}
\]
\[
= \ \emptyset \ a \rightarrow
\]
\[
cata_{\text{List}}
\]
\[
\emptyset \ a
\]
\[
\emptyset \ (\text{Int})
\]
\[
\rightarrow ___u = (\forall all t_{s2C0c}.t_{s2C0c} \rightarrow ___u = ((a \to t_{s2C0c} \to t_{s2C0c}) \to t_{s2C0c}))
\]
\[
\left(\ x_{s2B1} :: \text{Int}\right.
\]
\[
\emptyset \ t_{s2C0c}
\]
\[
c_{s2C5} OneShot :: t_{s2C0c}
\]
\[
c_{s2C6} OneShot :: (a \to t_{s2C0c} \to t_{s2C0c}) \to
\]
\[
\text{case } x_{s2B1} \text{ of wild}_B1 \{ \text{I# ds}_d2nc \to
\]
\[
\text{case ds}_d2nc \text{ of ds}_X2nc \{ 0 \to c_{s2C0d} ; \text{__DEFAULT} \to c_{s2C0d} \}
\]
\[
\right)
\]
\[
\left(\ r_{s2D9} :: a\right.
\]
\[
r_{s2D0} :: (\text{Int})
\]
\[
\rightarrow ___u = (\forall all t_{s2C0c}.t_{s2C0c} \rightarrow ___u = ((a \to t_{s2C0c} \to t_{s2C0c}) \to t_{s2C0c}))
\]
\[
x_{s2B1} :: \text{Int}
\]
\[
\emptyset \ t_{s2C0c}
\]
\[
c_{s2C5} OneShot :: t_{s2C0c}
\]
\[
c_{s2C6} OneShot :: (a \to t_{s2C0c} \to t_{s2C0c}) \to
\]
\[
\text{case } x_{s2B1} \text{ of wild}_B1 \{ \text{I# ds}_d2nc \to
\]
\[
\text{case ds}_d2nc \text{ of ds}_X2nc \{ 0 \to c_{s2C0d} ; \text{__DEFAULT} \to
\]
\[
\left(\ 
\right)
\]
\[
Contrast these with the previously given definitions! The only difference is that now all the arguments to \text{wntake} and \text{wnmap} are passed around in the recursive call. \text{main} changes
6.4. A DETAILED EXAMPLE

accordingly to:

```haskell
main :: (State# RealWorld) -> (# State# RealWorld, ()#))
\wmain
  = \w :: (State# RealWorld)
    case nfoldl'
      @ Int
      @ Integer
      (witerate
        @ Integer
        (\s7 :: Int# Integer \+@1 s7 lit)
        lit)
      @ (Int \to __u \to __u \to ((Integer \to t \to t) \to t))
      (\x :: Int \ot c1 :: t \ot c2 :: (Integer \to t \to t)
        case x of wild {I# ds -> c1})
      @ (Int)
      (r1 :: (Int
        \to __u \to ((forall t. t \to __u \to ((Integer \to t \to t) \to t)))
      x :: Int
      \ot t
      c1 :: t
      c2 :: (Integer \to t \to t)
      case x of wild {I# ds ->
        case ds of ds1 {I# 0 -> c1;
          __DEFAULT ->
            case ># ds1 0 of wild1 {
              True ->
                c2 r
                (let {s7 :: Int#
                  s7 = -# ds1 1}
                  in r1 ($wI# s7) \ot c1 c2);
                False -> __coerce t (error @ (List Integer) lvl1))})
      )
      )
      )
      )
    (nfoldl'
      (\s7 :: Integer \to PrelNum.*1 s7 lvl)
      )
      (\wmain 1000)
    @ ((Integer \to Integer)
      \to __u \to ((forall t. t \to __u \to ((Integer \to t \to t) \to t)))
    (\m3 \ot Integer \ot Integer)
    (\m2 \ot Integer \ot Integer)
    (\n7 :: Integer \to PrelNum.*1 s7 lvl)
    @ ((Integer \to Integer)
      \to __u \to ((forall t. t \to __u \to ((Integer \to t \to t) \to t)))
    (\m3 \ot Integer \ot Integer)
    (\m2 \ot Integer \ot Integer)
    (\n7 :: Integer \to PrelNum.*1 s7 lit)
    @ (List Integer)
    (\wNil \ot Integer)
    (\w^: \ot Integer))
```
6.4. A DETAILED EXAMPLE

\[
\begin{align*}
\text{(n :: Int} & \text{ ds :: Integer} \rightarrow \\
\text{case n of wild (I# x1} & \rightarrow \\
\text{let} \{ \\
\text{s7 :: Int#} & \text{s7} \\
\text{= +# x1 1} & \text{in } $wI# s7 \\
\text{)} \text{ in } $wI# s7 \\
\text{)}} \text{ of w1 (I# ww} & \rightarrow \\
\text{case PrelIO.$whPutStr} & \text{ PrelHandle.stdout (PrelShow.$wshowSignedInt 0 ww ($v[\text{ Char}] @)) v} \\
\text{of wild (} & \text{(# new_s, a1)} \rightarrow \\
\text{case PrelIO.$whPutChar PrelHandle.stdout }' & \text{ new_s} \\
\text{of wildl (} & \text{(# new_s1, a11)} \rightarrow \\
\text{(} & \text{ new_s1, $u() } #) \\
\text{)} & \\
\text{)} & \\
\end{align*}
\]

Fusion still takes place (GHC also reports the same number of applications of the cata-build rule): this can be seen from that nfoldl' is applied to witerate, so no intermediate list exists in between the two functions. But now look at the total allocations:

angel 170 (haskell/andreas): a.out +RTS -Sstderr
a.out +RTS -Sstderr
a.out +RTS -Sstderr

Alloc Collect Live GC GC TOT TOT Page Flts
bytes bytes bytes user elap user elap
1000
207544 0.00 0.00

207,544 bytes allocated in the heap
0 bytes copied during GC

0 collections in generation 0 ( 0.00s)
0 collections in generation 1 ( 0.00s)

1 Mb total memory in use

INIT time 0.00s ( 0.00s elapsed)
MUT time 0.00s ( 0.00s elapsed)
GC time 0.00s ( 0.00s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 0.00s ( 0.00s elapsed)

%GC time 0.0% (0.0% elapsed)

Alloc rate 2,075,440,000 bytes per MUT second

Productivity 100.0% of total user, 307018953.7% of total elapsed
Total allocation almost doubled compared to the run when we used static argument transformation and increased by 50% compared to the unoptimised program. The cata-build rule has been applied the very same number of times: 7 (the compiler’s output is not shown). It is applied four times to buildify the definitions of map, take, iterate, and the derived map_list, and three times to eliminate the intermediate lists from the original program between, iterate and take, take and the first map, and the first and the second map. The explanation for this phenomenon is in wmain: witerate is now applied to about 20 arguments, which are higher-order functions. It is reasonable to conclude that the STG machine is not particularly efficient when executing higher-order code.

A remark concerning the Standard Prelude definition of length is not inappropriate here. It is defined, for efficiency reasons, in terms of foldl', which is the strict version of folding from the left. Because it is folding from the left, we failed to turn it to a catamorphism, therefore the intermediate list between witerate and length remained. Had it been defined with a foldr, we would have the following result (only wmain is shown, as length’s definition is trivial):

```
wmain :: (State# RealWorld) -> case PrelIO.$whPutStr
(PrelHandle.stdout
(PrelNum.showSignedInteger
(PrelBase.zeroInt
(witerate
@g Integer
(\ s3 :: Integer -> PrelNum.*1 s3 lit)
lit
@g (Int -> Integer)
(\ x :: Int -> case x of wild { I# ds -> c1 })
(\ r :: Integer r1 :: (Int -> Integer) x :: Int ->
case x of wild { I# ds ->
case ds of ds1 {
0 -> c1;
DEFAULT ->
case ># ds1 0 of wild1 {
True ->
let {
s3 :: Int#
s3 = # ds1 1
} in PrelNum.*1 lit (r1 ($wI# s3));
False -> __coerce Integer (error @ (List Integer) lvl)
}
}
)}))
($wI# 1000))
```
6.4. A DETAILED EXAMPLE

\[
(w[] @ Char)
\]

\[
\begin{align*}
&\text{of wild } (\text{# new\_s, a1 #}) -> \\
&\text{case Prelude.$whPutChar Prelude.Handle.stdout 'new\_s'} \\
&\text{of wild1 } (\text{# new\_s1, a11 #}) -> \\
&\text{(# new\_s1, $w() #)} \\
&\}
\end{align*}
\]

Not a single list constructor remains: we managed to eliminate all the intermediate data structures. This is because \textit{length} is now a catamorphism (GHC also reports that the \texttt{cata-build} rule has been applied 8 times), the intermediate list between \texttt{witerate} and \texttt{length} also disappeared.

angel 57 (haskell/andreas): a.\texttt{out} +RTS -Sstderr
a.\texttt{out} +RTS -Sstderr
a.\texttt{out} +RTS -Sstderr

\begin{table}
\begin{tabular}{lrrrrrr}
\hline
Alloc & Collect & Live & GC & GC & TUT & TUT Page Flts
\hline
bytes & bytes & bytes & user & elap & user & elap
\hline
1000 & & & & & &
\hline
71028 & & & & & & 0.00 0.00
\hline
\end{tabular}
\end{table}

71,028 bytes allocated in the heap
0 bytes copied during GC

0 collections in generation 0 ( 0.00s)
0 collections in generation 1 ( 0.00s)

1 Mb total memory in use

\begin{tabular}{lrr}
\hline
INIT time & 0.00s ( 0.00s elapsed) \\
MUT time & 0.01s ( 0.00s elapsed) \\
GC time & 0.00s ( 0.00s elapsed) \\
EXIT time & 0.00s ( 0.00s elapsed) \\
Total time & 0.01s ( 0.00s elapsed) \\
\hline
%GC time & 0.0% ( 0.0% elapsed) \\
Alloc rate & 7,102,800 bytes per MUT second \\
Productivity & 100.0% of total user, 36756475700.0% of total elapsed \\
\hline
\end{tabular}

The total allocation is half of that the original, unoptimised (-O2, without warm fusion) program. It appears that the presence of the warm fusion optimisation affects how functions should be defined: with warm fusion, manually introducing strictness leads to decreased performance, while without warm fusion strict versions of functions are sometimes more efficient. This substantiates the saying: more haste, less speed.
### 6.5. THE BENCHMARKS

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp3</td>
<td>Calculate $3^n$ using Naturals</td>
</tr>
<tr>
<td>gen_regexps</td>
<td>Generate all the expansions of a generalised regular expression</td>
</tr>
<tr>
<td>paraffins</td>
<td>Generation of radicals</td>
</tr>
<tr>
<td>primes</td>
<td>Generate the first 1500 prime numbers</td>
</tr>
<tr>
<td>queens</td>
<td>Count the the number of solutions to the ”n queens” problem</td>
</tr>
<tr>
<td>rfib</td>
<td>nfib 30 with Doubles</td>
</tr>
<tr>
<td>tak</td>
<td>Calculate tak 24 16 8</td>
</tr>
<tr>
<td>x2n1</td>
<td>Calculate a root to the equation $x^n = 1$ using complex numbers</td>
</tr>
</tbody>
</table>

**Table 6.1** Programs of the imaginary subset

### 6.5 The benchmarks

To allow comparison with similar work we follow Gill [Gil96] and use the nofib benchmark suite. The nofib suite is divided into three subsets:

- the *imaginary* or toy subset: trivial few-liners like queens and fib. Mostly used in the literature to demonstrate the usefulness of optimisations which usually remain unsubstantiated afterwards.

- the *spectral* subset: somewhat bigger programs. Following Gill [Gil96] we include Hartel’s [HL93, Har94] benchmarks.

- the *real* subset: programs that are written to get a job done.

The programs with brief description and their original authors are listed in Tables 6.1, 6.2, 6.3 and 6.4. Data is gathered from the nofib suite directly (i.e. from the source) or when the code is completely unannotated from Gill [Gil96].

### 6.6 A short analysis of the benchmarks

Before we give endless pages of numbers of several different runs of the compiler we would like to ‘guess’ what our numbers could be. We make this guess based on the limitations of the implementation and our expectations.

1. **The Haskell Prelude is not put through the optimisation**\(^1\). The difficulty with optimising the Standard Prelude is that a number of definitions, types, and

\(^1\)It may be surprising to the uninitiated but the binary of the Glasgow Haskell Compiler, until very recently — GHC-4.06 is *not* an exception — is compiled without -O, i.e. warm fusion would not be attempted anyway.
functions in the Prelude are also hard-wired into the compiler itself and in some cases these hard-wired entities silently take precedence over the text of the files which define these datatypes and functions. In particular, the most commonly used *List* datatype is affected by this. Attempting fusion for the built-in *List* datatype is further complicated by the new RULES mechanism in GHC. The RULES mechanism is used to implement cheap deforestation ([Gil96]) — amongst other transformations, which can be described by an appropriately typed one-step rewrite rule — but it does not attempt to turn arbitrary functions into build-cata form.

In order to reap the benefits of warm fusion, we also use the aforementioned mechanism, but for historic reasons the function which we call cata in this thesis is called *foldr* in Haskell with a slightly different type: \(\forall \alpha \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta\) while the cata — as derived by the methods described in this thesis — would have

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>awards</td>
<td>Public awards scheme</td>
<td>Kevin Hammond</td>
</tr>
<tr>
<td>banner</td>
<td>Simple banner program</td>
<td>Mark P Jones</td>
</tr>
<tr>
<td>boyer</td>
<td>Boyer benchmark</td>
<td>Denis Howe</td>
</tr>
<tr>
<td>boyer2</td>
<td>Gabriel benchmark 'Boyer'</td>
<td></td>
</tr>
<tr>
<td>calendar</td>
<td>Calendar program</td>
<td>Mark P Jones</td>
</tr>
<tr>
<td>cichelli</td>
<td>Perfect hashing function</td>
<td>Iain Checkland</td>
</tr>
<tr>
<td>circsim</td>
<td>Circuit simulator</td>
<td>David King</td>
</tr>
<tr>
<td>clausify</td>
<td>Reducing propositions to clausal form</td>
<td>Colin Runciman</td>
</tr>
<tr>
<td>cse</td>
<td>Common subexpression elimination</td>
<td>Mark P Jones</td>
</tr>
<tr>
<td>eliza</td>
<td>Pseudo-psychoanalyst</td>
<td>Mark P Jones</td>
</tr>
<tr>
<td>expert</td>
<td>Minimal expers system</td>
<td>Ian Holyer</td>
</tr>
<tr>
<td>fibheaps</td>
<td>Fibonacci heaps</td>
<td>Chris Okasaki</td>
</tr>
<tr>
<td>fish</td>
<td>Knights tour</td>
<td>Jonathan Hill</td>
</tr>
<tr>
<td>life</td>
<td>Game of life</td>
<td>John Launchbury</td>
</tr>
<tr>
<td>mandel</td>
<td>Mandelbrot set generator</td>
<td>Jonathan Hill</td>
</tr>
<tr>
<td>mandel2</td>
<td>Mandelbrot set generator</td>
<td>David Hanley</td>
</tr>
<tr>
<td>minimax</td>
<td>Tic-tac-toe</td>
<td>Iain Checkland</td>
</tr>
<tr>
<td>multiplier</td>
<td>Binary multiplier</td>
<td>John T O'Donnell</td>
</tr>
<tr>
<td>pretty</td>
<td>Pretty printer</td>
<td></td>
</tr>
<tr>
<td>primetest</td>
<td>Probabilistic primality testing</td>
<td>David Lester</td>
</tr>
<tr>
<td>rewrite</td>
<td>Rewriting system</td>
<td>Mike Spivey</td>
</tr>
<tr>
<td>scc</td>
<td>Strongly connected components of a graph</td>
<td>John Launchbury</td>
</tr>
<tr>
<td>simple</td>
<td>Standard Id benchmark</td>
<td></td>
</tr>
<tr>
<td>sorting</td>
<td>Sorting algorithms</td>
<td>Will Partain</td>
</tr>
<tr>
<td>sphere</td>
<td>Ray tracer for spheres</td>
<td>David King</td>
</tr>
</tbody>
</table>

Table 6.2 Programs of the spectral subset
### 6.6. A SHORT ANALYSIS OF THE BENCHMARKS

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp_lab_zift</td>
<td>Image processing application</td>
<td></td>
</tr>
<tr>
<td>event</td>
<td>Event driven simulation</td>
<td></td>
</tr>
<tr>
<td>fft</td>
<td>Two Fast Fourier Transforms</td>
<td></td>
</tr>
<tr>
<td>genfft</td>
<td>Generation of synthetic FFT programs</td>
<td></td>
</tr>
<tr>
<td>ida</td>
<td>Solution of a particular configuration of the n-puzzle</td>
<td></td>
</tr>
<tr>
<td>listcompr</td>
<td>Compilation of list comprehensions</td>
<td></td>
</tr>
<tr>
<td>listcopy</td>
<td>Compilation of list comprehensions</td>
<td></td>
</tr>
<tr>
<td>parstof</td>
<td>Wadler’s method for lexing and parsing</td>
<td></td>
</tr>
<tr>
<td>sched</td>
<td>Calculation of an optimum schedule of parallel jobs</td>
<td></td>
</tr>
<tr>
<td>solid</td>
<td>Point membership classification algorithm</td>
<td></td>
</tr>
<tr>
<td>transform</td>
<td>Transformation of a number of programs represented as synchronous process networks into master-slave style parallel programs</td>
<td></td>
</tr>
<tr>
<td>typecheck</td>
<td>Polymorphic typechecking of a set of function definitions</td>
<td></td>
</tr>
<tr>
<td>wlang</td>
<td>Wang’s algorithm for solving a system of linear equations</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.3** Programs of the spectral subset: the Hartel Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>Strictness analyser</td>
<td>Iain Checkland</td>
</tr>
<tr>
<td>bspt</td>
<td>BSP tree modeller</td>
<td>Paul Sanders</td>
</tr>
<tr>
<td>compress</td>
<td>Text compression</td>
<td></td>
</tr>
<tr>
<td>ebnf2ps</td>
<td>Syntax diagram generator</td>
<td>Peter Thiemann</td>
</tr>
<tr>
<td>fluid</td>
<td>Fluid dynamics program</td>
<td>Xiaoming Zhang</td>
</tr>
<tr>
<td>fulsom</td>
<td>Solid modeling</td>
<td>Duncan Sinclair</td>
</tr>
<tr>
<td>gamteb</td>
<td>Monte Carlo photon transport</td>
<td>Pat Fasel</td>
</tr>
<tr>
<td>gg</td>
<td>Graphs from GRIP statistics</td>
<td>Iain Checkland</td>
</tr>
<tr>
<td>grep</td>
<td>Grep program</td>
<td></td>
</tr>
<tr>
<td>hpg</td>
<td>Haskell program generator</td>
<td>Nick North</td>
</tr>
<tr>
<td>infer</td>
<td>Hindley-Milner type inference</td>
<td>Phil Wadler</td>
</tr>
<tr>
<td>lift</td>
<td>Fully-lazy lambda lifter</td>
<td>David Lester &amp; Simon Peyton Jones</td>
</tr>
<tr>
<td>maillist</td>
<td>Mailing list generator</td>
<td>Paul Hudak</td>
</tr>
<tr>
<td>mkhprog</td>
<td>Command line parser generator</td>
<td>N D North</td>
</tr>
<tr>
<td>parser</td>
<td>Partial Haskell parser</td>
<td>Julian Seward</td>
</tr>
<tr>
<td>prolog</td>
<td>mini-Prolog interpreter</td>
<td>Mark P Jones</td>
</tr>
<tr>
<td>reptile</td>
<td>Escher tiling program</td>
<td>Sandra Foubister</td>
</tr>
<tr>
<td>rsa</td>
<td>RSA encryption</td>
<td>John Launchbury</td>
</tr>
<tr>
<td>symalg</td>
<td>Command line evaluator</td>
<td></td>
</tr>
<tr>
<td>veritas</td>
<td>Theorem prover</td>
<td>Gareth Howells</td>
</tr>
</tbody>
</table>

**Table 6.4** Programs of the real subset
6.7. SUMMARY

In this section we have a look at the numbers our transformations produce and attempt an analysis of the sometimes surprising results.

6.7.1 The control run

Compilation times and run times are reported in seconds, while binary size, total allocation and heap residency are shown in bytes. There are no surprises in Tables 6.5, 6.7, 6.6, or 6.8. Maximum heap residency is sometimes 0, but that only means that the program is small, so no sample of the heap contents is available. This is typically true for programs which allocate less than 300K in total.

It is intriguing to compare binary sizes to those reported in [Gil96]. It appears from this comparison that the programs generated by GHC-4.06 are approximately half the size that of the ones compiled by GHC-0.26.
### 6.7. SUMMARY

#### Table 6.5 Control run: imaginary subset

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap</th>
<th>Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>10.96</td>
<td>182618</td>
<td>2.28</td>
<td>54794652</td>
<td>2944284</td>
<td>2944284</td>
</tr>
<tr>
<td>fft</td>
<td>14.71</td>
<td>404394</td>
<td>0.41</td>
<td>10804340</td>
<td>83084</td>
<td>83084</td>
</tr>
<tr>
<td>genfft</td>
<td>13.31</td>
<td>188602</td>
<td>0.46</td>
<td>26356304</td>
<td>3372</td>
<td>3372</td>
</tr>
<tr>
<td>listcompr</td>
<td>12.83</td>
<td>188010</td>
<td>2.13</td>
<td>129027180</td>
<td>2372</td>
<td>2372</td>
</tr>
<tr>
<td>listcopy</td>
<td>12.9</td>
<td>188282</td>
<td>2.37</td>
<td>144099284</td>
<td>2152</td>
<td>2152</td>
</tr>
<tr>
<td>nucleic2</td>
<td>92.4</td>
<td>479130</td>
<td>1.18</td>
<td>47629352</td>
<td>5632</td>
<td>5632</td>
</tr>
<tr>
<td>parstof</td>
<td>93.07</td>
<td>327290</td>
<td>0.16</td>
<td>1350940</td>
<td>41204</td>
<td>41204</td>
</tr>
<tr>
<td>sched</td>
<td>11.27</td>
<td>180074</td>
<td>0.45</td>
<td>17368564</td>
<td>3176</td>
<td>3176</td>
</tr>
<tr>
<td>solid</td>
<td>40.63</td>
<td>424890</td>
<td>1.84</td>
<td>86154404</td>
<td>295332</td>
<td>295332</td>
</tr>
<tr>
<td>transform</td>
<td>56.28</td>
<td>298602</td>
<td>6.51</td>
<td>307153768</td>
<td>27600</td>
<td>27600</td>
</tr>
<tr>
<td>typecheck</td>
<td>18.19</td>
<td>201770</td>
<td>4.44</td>
<td>146058248</td>
<td>3552</td>
<td>3552</td>
</tr>
<tr>
<td>wang</td>
<td>12.65</td>
<td>397674</td>
<td>1.51</td>
<td>44075624</td>
<td>3922944</td>
<td>3922944</td>
</tr>
<tr>
<td>wave4main</td>
<td>16.31</td>
<td>205450</td>
<td>5.39</td>
<td>48742552</td>
<td>1835352</td>
<td>1835352</td>
</tr>
</tbody>
</table>

#### Table 6.6 Control run: the Hartel Benchmarks

#### 6.7.2 Normalised run

One thing to note here: because of the implementation, the normalised run needs the results of the cata, build, map derivation phase (see Sections 4.5.1, 4.5.2), in other words the increased code size and increased compilation times are partly due to those. There is no change in total allocation, which is what we expect. This means that all the normalised wrappers get inlined and there is no penalty for rearranging the arguments to functions.

It appears that there is a slight increase in runtimes for most programs, while parstof and sched, amongst others, improves. The improvement is likely to be due to the extra run
<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>6.69</td>
<td>389866</td>
<td>5.55</td>
<td>181389016</td>
<td>3172</td>
</tr>
<tr>
<td>awards</td>
<td>6.67</td>
<td>204922</td>
<td>0.12</td>
<td>105220</td>
<td>0</td>
</tr>
<tr>
<td>banner</td>
<td>9.38</td>
<td>203379</td>
<td>0.12</td>
<td>83584</td>
<td>0</td>
</tr>
<tr>
<td>boyer</td>
<td>13.34</td>
<td>199226</td>
<td>0.62</td>
<td>27229080</td>
<td>31376</td>
</tr>
<tr>
<td>boyer2</td>
<td>20.54</td>
<td>231610</td>
<td>0.23</td>
<td>1730252</td>
<td>31832</td>
</tr>
<tr>
<td>calendar</td>
<td>6.69</td>
<td>180858</td>
<td>0.10</td>
<td>346404</td>
<td>2336</td>
</tr>
<tr>
<td>cichelli</td>
<td>18.31</td>
<td>225235</td>
<td>1.56</td>
<td>34323100</td>
<td>21680</td>
</tr>
<tr>
<td>circsim</td>
<td>25.24</td>
<td>284522</td>
<td>16.13</td>
<td>684021348</td>
<td>12754416</td>
</tr>
<tr>
<td>clausify</td>
<td>7.97</td>
<td>190746</td>
<td>0.66</td>
<td>19524212</td>
<td>3432</td>
</tr>
<tr>
<td>constraints</td>
<td>12.32</td>
<td>197386</td>
<td>64.27</td>
<td>965843528</td>
<td>20793288</td>
</tr>
<tr>
<td>cryptarithm1</td>
<td>3.57</td>
<td>173082</td>
<td>16.86</td>
<td>926684208</td>
<td>2228</td>
</tr>
<tr>
<td>cryptarithm2</td>
<td>13.51</td>
<td>211482</td>
<td>0.49</td>
<td>19785960</td>
<td>4248</td>
</tr>
<tr>
<td>cse</td>
<td>11.58</td>
<td>209562</td>
<td>0.11</td>
<td>529672</td>
<td>3608</td>
</tr>
<tr>
<td>eliza</td>
<td>13.89</td>
<td>228259</td>
<td>0.13</td>
<td>262428</td>
<td>0</td>
</tr>
<tr>
<td>expert</td>
<td>25.01</td>
<td>241539</td>
<td>0.15</td>
<td>111236</td>
<td>0</td>
</tr>
<tr>
<td>fibheaps</td>
<td>9.97</td>
<td>234250</td>
<td>0.56</td>
<td>25572344</td>
<td>154980</td>
</tr>
<tr>
<td>fish</td>
<td>10.34</td>
<td>185658</td>
<td>0.24</td>
<td>7152788</td>
<td>2024</td>
</tr>
<tr>
<td>gcd</td>
<td>3.8</td>
<td>178618</td>
<td>0.59</td>
<td>25989928</td>
<td>5428</td>
</tr>
<tr>
<td>life</td>
<td>5.24</td>
<td>182730</td>
<td>4.61</td>
<td>222927164</td>
<td>12512</td>
</tr>
<tr>
<td>knights</td>
<td>32.39</td>
<td>235930</td>
<td>0.16</td>
<td>968360</td>
<td>16540</td>
</tr>
<tr>
<td>mandel1</td>
<td>13.34</td>
<td>466627</td>
<td>3.38</td>
<td>103856092</td>
<td>19332</td>
</tr>
<tr>
<td>mandel2</td>
<td>6.55</td>
<td>186634</td>
<td>0.24</td>
<td>4512064</td>
<td>2352</td>
</tr>
<tr>
<td>minimax</td>
<td>21.38</td>
<td>212499</td>
<td>0.13</td>
<td>2684428</td>
<td>2992</td>
</tr>
<tr>
<td>multiplier</td>
<td>9.27</td>
<td>188634</td>
<td>1.99</td>
<td>95067900</td>
<td>18616</td>
</tr>
<tr>
<td>para</td>
<td>9.74</td>
<td>227779</td>
<td>9.65</td>
<td>350640388</td>
<td>95228</td>
</tr>
<tr>
<td>primetest</td>
<td>13.82</td>
<td>251267</td>
<td>7.77</td>
<td>89433208</td>
<td>62176</td>
</tr>
<tr>
<td>puzzle</td>
<td>10.8</td>
<td>191578</td>
<td>2.75</td>
<td>58604352</td>
<td>2140</td>
</tr>
<tr>
<td>rewrite</td>
<td>21.52</td>
<td>220842</td>
<td>0.53</td>
<td>14027148</td>
<td>8048</td>
</tr>
<tr>
<td>scc</td>
<td>4.81</td>
<td>171962</td>
<td>0.10</td>
<td>6284</td>
<td>0</td>
</tr>
<tr>
<td>sorting</td>
<td>9.62</td>
<td>200467</td>
<td>0.15</td>
<td>524164</td>
<td>59448</td>
</tr>
<tr>
<td>sphere</td>
<td>20.72</td>
<td>466010</td>
<td>2.15</td>
<td>55503740</td>
<td>3000</td>
</tr>
</tbody>
</table>

**Table 6.7** Control run: spectral subset

of the simplifier after normalise. This was verified for parstof with normalise and derive switched off but allowing for the extra run of the simplifier.

In clausify the maximum heap residency increased dramatically, which appears to be due to the fact that there is only one sample.
### Table 6.8 Control run: the real subset

<table>
<thead>
<tr>
<th>Program</th>
<th>Time</th>
<th>Binary size</th>
<th>Time</th>
<th>Total allocation</th>
<th>Max</th>
<th>Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>compile to</td>
<td></td>
<td>run</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exp3_8</td>
<td>1.21</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gen_regexps</td>
<td>1.01</td>
<td>1.00</td>
<td>1.22</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>paraffins</td>
<td>1.09</td>
<td>1.01</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primes</td>
<td>1.03</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>queens</td>
<td>1.02</td>
<td>1.00</td>
<td>1.13</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rfib</td>
<td>1.00</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>tak</td>
<td>1.04</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>x2n1</td>
<td>1.06</td>
<td>1.00</td>
<td>1.16</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wheel-sieve1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wheel-sieve2</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>integrate</td>
<td>1.01</td>
<td>1.00</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.00</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.21</td>
<td>1.01</td>
<td>1.22</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.04</td>
<td>1.00</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

### 6.7.3 Buildify only

In general, we can expect buildify to increase compilation times, because of the worker-wrapper split, which gives rise to further inlining. Binary sizes should only be affected to
the extent normalise affects it, as buildify and normalise works very much the same way.

Two programs, `clausify` and `fibheaps`, are highly problematic: total allocation increases tenfold for `clausify`, while `fibheaps` runs out of heap. Examination revealed that in the case of `clausify` this is due to the highly successful transformation on the datatype shown in Table 6.13. Every single function defined in the module is successfully transformed to explicit build form leading to increased allocation. There is nothing to worry about yet, as no `cata-build` reductions take place in this run. It just shows how bad the result of buildify can get.

The problem with `fibheaps` is likely to be the same as it also uses a datatype on which most of its functions can be buildified.

Total allocation in most programs are not affected, which is a consequence of not doing fusion for the built-in datatype `List`.

### 6.7.4 Catify only

We expect catify to result in increased runtimes as compared to buildify. The reason for this is the worker-wrapper split. Here we split a single function to as many as data constructors the fusible — the functions first argument — datatype has.

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>1.11</td>
<td>1.01</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fft</td>
<td>1.07</td>
<td>1.00</td>
<td>1.12</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>genfft</td>
<td>1.05</td>
<td>1.00</td>
<td>1.09</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcompr</td>
<td>1.07</td>
<td>1.00</td>
<td>1.06</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcopy</td>
<td>1.07</td>
<td>1.00</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>nucleic2</td>
<td>1.12</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>parstof</td>
<td>1.09</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sched</td>
<td>1.17</td>
<td>1.02</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>solid</td>
<td>1.12</td>
<td>1.01</td>
<td>1.09</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>transform</td>
<td>1.07</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>typecheck</td>
<td>1.09</td>
<td>1.00</td>
<td>1.07</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wang</td>
<td>1.06</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wave4main</td>
<td>1.08</td>
<td>1.00</td>
<td>1.06</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.05</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.12</td>
<td>1.02</td>
<td>1.12</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.08</td>
<td>1.01</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 6.10 Normalise: the Hartel Benchmarks
6.7. SUMMARY

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>1.01</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>awards</td>
<td>1.01</td>
<td>1.00</td>
<td>0.67</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>banner</td>
<td>1.04</td>
<td>1.00</td>
<td>0.83</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>boyer</td>
<td>1.23</td>
<td>1.01</td>
<td>0.90</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>boyer2</td>
<td>1.02</td>
<td>1.00</td>
<td>1.26</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>calendar</td>
<td>1.11</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cichelli</td>
<td>1.07</td>
<td>1.00</td>
<td>1.07</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>circsim</td>
<td>1.14</td>
<td>1.01</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>clausify</td>
<td>1.27</td>
<td>1.02</td>
<td>1.12</td>
<td>1.00</td>
<td>25.68²</td>
</tr>
<tr>
<td>constraints</td>
<td>1.13</td>
<td>1.01</td>
<td>1.06</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm1</td>
<td>1.11</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm2</td>
<td>1.11</td>
<td>1.00</td>
<td>0.88</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cse</td>
<td>1.08</td>
<td>1.00</td>
<td>1.09</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>eliza</td>
<td>1.09</td>
<td>1.00</td>
<td>0.92</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>expert</td>
<td>1.12</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>fibheaps</td>
<td>1.17</td>
<td>1.01</td>
<td>1.07</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fish</td>
<td>1.12</td>
<td>1.00</td>
<td>0.92</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gcd</td>
<td>1.09</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>life</td>
<td>1.06</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>knights</td>
<td>1.08</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mandel</td>
<td>1.08</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mandel2</td>
<td>1.11</td>
<td>1.01</td>
<td>1.21</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>minimax</td>
<td>1.09</td>
<td>1.00</td>
<td>0.85</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>multiplier</td>
<td>1.07</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>para</td>
<td>1.06</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primetest</td>
<td>1.06</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>puzzle</td>
<td>1.13</td>
<td>1.01</td>
<td>1.07</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rewrite</td>
<td>1.12</td>
<td>1.01</td>
<td>1.09</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>scc</td>
<td>1.09</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>sorting</td>
<td>1.16</td>
<td>1.01</td>
<td>0.80</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sphere</td>
<td>1.11</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.01</td>
<td>1.00</td>
<td>0.67</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.27</td>
<td>1.02</td>
<td>1.26</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.09</td>
<td>1.01</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 6.11  Normalise: spectral subset

Tables 6.22, 6.23, 6.24, and 6.25 complete the data we gathered to demonstrate the efficiency of warm fusion. We conclude our measurements with a few random comments:

- It appears that the nofib suite has not kept up with the constantly improving micro-processors, larger and larger amounts of memory and improvements to GHC. Run-
### Table 6.12 Normalise: the real subset

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>1.06</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>compress</td>
<td>1.06</td>
<td>1.01</td>
<td>1.10</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ebnf2ps</td>
<td>1.07</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fem</td>
<td>1.09</td>
<td>1.01</td>
<td>0.84</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fulsom</td>
<td>1.13</td>
<td>1.01</td>
<td>1.27</td>
<td>1.10</td>
<td>0.75</td>
</tr>
<tr>
<td>gamteb</td>
<td>1.08</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gg</td>
<td>1.08</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>grep</td>
<td>1.01</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>hpg</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>infer</td>
<td>1.01</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>lift</td>
<td>1.08</td>
<td>1.00</td>
<td>0.92</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>maillist</td>
<td>1.09</td>
<td>1.00</td>
<td>1.27</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mkhprog</td>
<td>1.08</td>
<td>1.00</td>
<td>0.69</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>parser</td>
<td>1.11</td>
<td>1.01</td>
<td>1.16</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pic</td>
<td>1.07</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>prolog</td>
<td>1.06</td>
<td>1.00</td>
<td>0.92</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>reptile</td>
<td>1.08</td>
<td>1.00</td>
<td>1.26</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rsa</td>
<td>1.07</td>
<td>1.00</td>
<td>1.09</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>veritas</td>
<td>1.04</td>
<td>1.01</td>
<td>1.09</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Minimum</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
<td><strong>0.69</strong></td>
<td><strong>1.00</strong></td>
<td>0.75</td>
</tr>
<tr>
<td><strong>Maximum</strong></td>
<td><strong>1.13</strong></td>
<td><strong>1.01</strong></td>
<td><strong>1.27</strong></td>
<td><strong>1.00</strong></td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Geometric mean</strong></td>
<td><strong>1.06</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.03</strong></td>
<td><strong>1.00</strong></td>
<td>0.98</td>
</tr>
</tbody>
</table>

**data**

<table>
<thead>
<tr>
<th>Formula</th>
<th>Sym</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Formula</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dis Formula</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Con Formula</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imp Formula</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eqv Formula</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.13** A datatype making buildify too successful

times, with and without warm fusion, are generally under 10s. In order to make a fair comparison with for example the cheap deforestation work [Gil96], one would have to re-run the benchmarks. Unfortunately, this is not possible anymore as the compiler versions used do not build any longer.

- It is interesting to examine the result of warm fusion on larger programs: in general the transformation has the effect of producing a lot of higher-order functions. The
Table 6.14 Buildify only: imaginary subset

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>1.06</td>
<td>1.01</td>
<td>1.05</td>
<td>1.01</td>
<td>1.04</td>
</tr>
<tr>
<td>fft</td>
<td>1.03</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>genfft</td>
<td>1.01</td>
<td>1.00</td>
<td>0.93</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcompr</td>
<td>1.02</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcopy</td>
<td>1.02</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>nucleic2</td>
<td>1.10</td>
<td>1.01</td>
<td>0.98</td>
<td>1.01</td>
<td>1.03</td>
</tr>
<tr>
<td>parstof</td>
<td>1.04</td>
<td>1.00</td>
<td>0.88</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sched</td>
<td>1.15</td>
<td>1.02</td>
<td>0.84</td>
<td>1.19</td>
<td>0.93</td>
</tr>
<tr>
<td>solid</td>
<td>1.11</td>
<td>1.02</td>
<td>1.81</td>
<td>1.97</td>
<td>1.00</td>
</tr>
<tr>
<td>transform</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>typecheck</td>
<td>1.10</td>
<td>1.01</td>
<td>1.01</td>
<td>1.07</td>
<td>0.92</td>
</tr>
<tr>
<td>wang</td>
<td>1.01</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wave4main</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.01</td>
<td>1.00</td>
<td>0.88</td>
<td>1.00</td>
<td>0.92</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.15</td>
<td>1.02</td>
<td>1.81</td>
<td>1.97</td>
<td>1.04</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.05</td>
<td>1.01</td>
<td>1.01</td>
<td>1.07</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 6.15 Buildify only: the Hartel Benchmarks

somewhat disappointing results are most probably due to the fact that buildify and catify makes programs run much slower and the STG machine is not well-suited to run programs containing a lot of higher-order functions.

- As a result of the transformations some programs break: most of them run out of
<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>1.03</td>
<td>1.00</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>awards</td>
<td>1.03</td>
<td>1.00</td>
<td>0.83</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>banner</td>
<td>1.06</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>boyer</td>
<td>1.24</td>
<td>1.01</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>boyer2</td>
<td>1.01</td>
<td>1.00</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>calendar</td>
<td>1.06</td>
<td>1.00</td>
<td>1.40</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cichelli</td>
<td>1.04</td>
<td>1.00</td>
<td>1.01</td>
<td>1.08</td>
<td>0.97</td>
</tr>
<tr>
<td>circsim</td>
<td>1.13</td>
<td>1.01</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>clausify</td>
<td>1.43</td>
<td>1.05</td>
<td>23.94</td>
<td>11.72</td>
<td>3234.33</td>
</tr>
<tr>
<td>constraints</td>
<td>1.07</td>
<td>1.01</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm1</td>
<td>1.05</td>
<td>1.00</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm2</td>
<td>1.05</td>
<td>1.00</td>
<td>0.92</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cse</td>
<td>1.05</td>
<td>1.00</td>
<td>1.27</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>eliza</td>
<td>1.05</td>
<td>1.00</td>
<td>0.85</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>expert</td>
<td>1.12</td>
<td>1.02</td>
<td>0.80</td>
<td>1.01</td>
<td>NA</td>
</tr>
<tr>
<td>fibheaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fish</td>
<td>1.07</td>
<td>1.00</td>
<td>0.79</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gcd</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>life</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>knights</td>
<td>1.04</td>
<td>1.01</td>
<td>1.25</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mandel</td>
<td>1.02</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mandel2</td>
<td>1.05</td>
<td>1.01</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>minimax</td>
<td>1.06</td>
<td>1.01</td>
<td>1.00</td>
<td>1.01</td>
<td>1.14</td>
</tr>
<tr>
<td>multiplier</td>
<td>1.03</td>
<td>1.00</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>para</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primetest</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>puzzle</td>
<td>1.12</td>
<td>1.01</td>
<td>1.08</td>
<td>1.20</td>
<td>1.18</td>
</tr>
<tr>
<td>rewrite</td>
<td>1.12</td>
<td>1.01</td>
<td>0.89</td>
<td>1.00</td>
<td>1.02</td>
</tr>
<tr>
<td>scc</td>
<td>1.03</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>sorting</td>
<td>1.22</td>
<td>1.02</td>
<td>0.53</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sphere</td>
<td>1.07</td>
<td>1.00</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.01</td>
<td>1.00</td>
<td>0.79</td>
<td>1.00</td>
<td>0.97</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.43</td>
<td>1.05</td>
<td>23.49</td>
<td>11.72</td>
<td>3234.23</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.06</td>
<td>1.01</td>
<td>1.08</td>
<td>1.09</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 6.16 Buildify only: spectral subset

stack space. These are omitted from the benchmarks without further notice.
6.8. CONCLUSIONS

We base our summary on two sources: the detailed example in Section 6.4 and the nofib suite. Most of the programs in the nofib suite are not affected by the transformations (Section 6.6) therefore the detailed example is the more important source.

1. Transforming programs to build-cata form results in a considerable increase in total allocation. See Tables 6.14 through 6.17, and Tables 6.18 through 6.21. This suggests that in an industrial strength implementation care must be taken to verify if the cata-build rule is applied enough times, and if not the transformations need to be reversed.

2. Static argument transformation (SAT) is absolutely essential to get improvements (Page 114). It appears that the STG machine is not well-equipped to execute heavy higher-order code.

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>1.05</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>compress</td>
<td>1.09</td>
<td>1.02</td>
<td>1.03</td>
<td>1.01</td>
<td>1.47</td>
</tr>
<tr>
<td>ebnf2ps</td>
<td>1.02</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fem</td>
<td>1.04</td>
<td>1.01</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fulsom</td>
<td>1.13</td>
<td>1.03</td>
<td>1.04</td>
<td>1.15</td>
<td>0.96</td>
</tr>
<tr>
<td>gamteb</td>
<td>1.01</td>
<td>1.00</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gg</td>
<td>1.05</td>
<td>1.01</td>
<td>1.11</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>grep</td>
<td>1.02</td>
<td>1.00</td>
<td>1.20</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>hpg</td>
<td>1.02</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>infer</td>
<td>1.05</td>
<td>1.01</td>
<td>1.03</td>
<td>0.99</td>
<td>1.05</td>
</tr>
<tr>
<td>lift</td>
<td>1.03</td>
<td>1.00</td>
<td>0.69</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>maillist</td>
<td>1.02</td>
<td>1.00</td>
<td>1.14</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mkhprog</td>
<td>1.03</td>
<td>1.00</td>
<td>0.77</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>parser</td>
<td>1.10</td>
<td>1.01</td>
<td>0.80</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pic</td>
<td>1.01</td>
<td>1.00</td>
<td>0.92</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>prolog</td>
<td>1.03</td>
<td>1.00</td>
<td>0.77</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>reptile</td>
<td>1.02</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rsa</td>
<td>0.99</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>veritas</td>
<td>1.05</td>
<td>1.01</td>
<td>1.27</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.99</td>
<td>1.00</td>
<td>0.69</td>
<td>0.99</td>
<td>0.96</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.13</td>
<td>1.03</td>
<td>1.27</td>
<td>1.15</td>
<td>1.47</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.03</td>
<td>1.01</td>
<td>0.96</td>
<td>1.00</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table 6.17  Buildify only: the real subset

6.8 Conclusions
### 6.8. CONCLUSIONS

**Table 6.18** Catify only: imaginary subset

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp3_8</td>
<td>1.20</td>
<td>1.01</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gen_regexps</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>paraffins</td>
<td>1.10</td>
<td>1.01</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primes</td>
<td>1.04</td>
<td>1.00</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>queens</td>
<td>1.03</td>
<td>1.00</td>
<td>1.13</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rfib</td>
<td>1.05</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>tak</td>
<td>0.99</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>x2n1</td>
<td>1.06</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wheel-sieve1</td>
<td>0.99</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wheel-sieve2</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>integrate</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>0.99</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.20</td>
<td>1.01</td>
<td>1.13</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.04</td>
<td>1.00</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

3. The Standard Prelude is biased towards a compiler which does not use fusion (Section 6.4), which limits the applicability of fusion.

4. Binary sizes are practically unaffected by warm fusion, as the wrappers are always inlined. The only exception is the wrappers for exported functions, which are required in other modules.
<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>1.03</td>
<td>1.00</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>awards</td>
<td>1.03</td>
<td>1.00</td>
<td>0.83</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>banner</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>boyer</td>
<td>1.74</td>
<td>1.08</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>boyer2</td>
<td>1.02</td>
<td>1.00</td>
<td>0.91</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>calendar</td>
<td>1.03</td>
<td>1.00</td>
<td>1.20</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cichelli</td>
<td>1.05</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>circsim</td>
<td>1.11</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>clausify</td>
<td>1.20</td>
<td>1.02</td>
<td>1.11</td>
<td>1.00</td>
<td>25.68</td>
</tr>
<tr>
<td>constraints</td>
<td>1.08</td>
<td>1.01</td>
<td>0.99</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm1</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm2</td>
<td>1.03</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cse</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>eliza</td>
<td>1.06</td>
<td>1.00</td>
<td>0.62</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>fish</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gcd</td>
<td>1.03</td>
<td>1.00</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>life</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>knights</td>
<td>1.10</td>
<td>1.03</td>
<td>1.06</td>
<td>0.97</td>
<td>1.05</td>
</tr>
<tr>
<td>mandel</td>
<td>1.01</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mandel2</td>
<td>1.06</td>
<td>1.01</td>
<td>0.88</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>minimax</td>
<td>1.05</td>
<td>1.00</td>
<td>1.38</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>multiplier</td>
<td>1.01</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>para</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primetest</td>
<td>1.01</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>puzzle</td>
<td>1.47</td>
<td>1.07</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>scc</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>sorting</td>
<td>1.11</td>
<td>1.01</td>
<td>0.73</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sphere</td>
<td>1.06</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.01</td>
<td>1.00</td>
<td>0.62</td>
<td>0.97</td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.74</td>
<td>1.08</td>
<td>1.38</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.08</td>
<td>1.02</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 6.19 Cotify only: spectral subset
### Table 6.20  Catify only: the Hartel Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>1.06</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fft</td>
<td>1.03</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>genfft</td>
<td>1.00</td>
<td>1.00</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcompr</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcopy</td>
<td>1.02</td>
<td>1.00</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>nucleic2</td>
<td>1.10</td>
<td>1.01</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>parstof</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sched</td>
<td>1.11</td>
<td>1.02</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>transform</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>typecheck</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wang</td>
<td>1.01</td>
<td>1.00</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wave4main</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.00</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.11</td>
<td>1.02</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.04</td>
<td>1.01</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

### Table 6.21  Catify only: the real subset

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebnf2ps</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fem</td>
<td>1.05</td>
<td>1.01</td>
<td>0.80</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gamteb</td>
<td>1.02</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>grep</td>
<td>1.02</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>hpg</td>
<td>1.03</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>infer</td>
<td>1.02</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>0.98</td>
</tr>
<tr>
<td>lift</td>
<td>1.03</td>
<td>1.00</td>
<td>0.69</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>maillist</td>
<td>1.01</td>
<td>1.00</td>
<td>1.14</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mkhprog</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>parser</td>
<td>1.28</td>
<td>1.04</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pic</td>
<td>1.01</td>
<td>1.00</td>
<td>1.12</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>prolog</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>reptile</td>
<td>1.02</td>
<td>1.00</td>
<td>1.16</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rsa</td>
<td>1.01</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>veritas</td>
<td>1.08</td>
<td>1.03</td>
<td>0.73</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.01</td>
<td>1.00</td>
<td>0.69</td>
<td>1.00</td>
<td>0.98</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.28</td>
<td>1.04</td>
<td>1.16</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.03</td>
<td>1.01</td>
<td>0.97</td>
<td>1.00</td>
<td>0.99</td>
</tr>
</tbody>
</table>
### Table 6.22
Buildify, catify and the `cata-build` rule: imaginary subset

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp3_8</td>
<td>1.39</td>
<td>1.02</td>
<td>1.90</td>
<td>1.80</td>
<td>0.35</td>
</tr>
<tr>
<td>gen_regexps</td>
<td>1.02</td>
<td>1.00</td>
<td>1.33</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>paraffins</td>
<td>1.10</td>
<td>1.01</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primes</td>
<td>1.07</td>
<td>1.00</td>
<td>1.12</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>queens</td>
<td>1.03</td>
<td>1.00</td>
<td>1.32</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rfib</td>
<td>1.00</td>
<td>1.00</td>
<td>0.95</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>tak</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>x2n1</td>
<td>1.07</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wheel-sieve1</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wheel-sieve2</td>
<td>1.04</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>integrate</td>
<td>1.03</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Minimum: 1.00 1.00 0.95 1.00 0.35
Maximum: 1.39 1.02 1.90 1.80 1.00
Geometric mean: 1.06 1.01 1.12 1.05 0.90
### 6.8. CONCLUSIONS

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>1.02</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>awards</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>banner</td>
<td>1.08</td>
<td>1.00</td>
<td>0.83</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>boyer</td>
<td>1.78</td>
<td>1.08</td>
<td>1.06</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>boyer2</td>
<td>1.02</td>
<td>1.00</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>calendar</td>
<td>1.06</td>
<td>1.00</td>
<td>1.20</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cichelli</td>
<td>1.09</td>
<td>1.01</td>
<td>1.01</td>
<td>1.08</td>
<td>0.97</td>
</tr>
<tr>
<td>circsim</td>
<td>1.19</td>
<td>1.02</td>
<td>1.04</td>
<td>1.03</td>
<td>0.98</td>
</tr>
<tr>
<td>clausify</td>
<td>1.46</td>
<td>1.05</td>
<td>23.35</td>
<td>11.72</td>
<td>3234.33</td>
</tr>
<tr>
<td>constraints</td>
<td>1.08</td>
<td>1.01</td>
<td>0.99</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm1</td>
<td>1.06</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cryptarithm2</td>
<td>1.04</td>
<td>1.00</td>
<td>0.88</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>cse</td>
<td>1.06</td>
<td>1.00</td>
<td>0.73</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>eliza</td>
<td>1.05</td>
<td>1.00</td>
<td>0.77</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>fish</td>
<td>1.07</td>
<td>1.00</td>
<td>0.75</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gcd</td>
<td>1.03</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>life</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>knights</td>
<td>1.11</td>
<td>1.03</td>
<td>0.88</td>
<td>0.97</td>
<td>1.05</td>
</tr>
<tr>
<td>mandel</td>
<td>1.03</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mandel2</td>
<td>1.10</td>
<td>1.01</td>
<td>0.88</td>
<td>1.04</td>
<td>1.03</td>
</tr>
<tr>
<td>minimax</td>
<td>1.09</td>
<td>1.01</td>
<td>0.92</td>
<td>1.01</td>
<td>1.14</td>
</tr>
<tr>
<td>multiplier</td>
<td>1.03</td>
<td>1.00</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>para</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>primetest</td>
<td>1.02</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>puzzle</td>
<td>1.41</td>
<td>1.05</td>
<td>1.08</td>
<td>1.23</td>
<td>1.08</td>
</tr>
<tr>
<td>scc</td>
<td>1.05</td>
<td>1.00</td>
<td>1.10</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>sorting</td>
<td>1.26</td>
<td>1.03</td>
<td>0.73</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sphere</td>
<td>1.06</td>
<td>1.00</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.02</td>
<td>1.00</td>
<td>0.73</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.78</td>
<td>1.08</td>
<td>23.35</td>
<td>11.72</td>
<td>3234.33</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.10</td>
<td>1.01</td>
<td>1.06</td>
<td>1.10</td>
<td>1.34</td>
</tr>
</tbody>
</table>

Table 6.23  Buildify, catify and the cata-build rule: spectral subset
## 6.8. CONCLUSIONS

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>wave4main</td>
<td>1.06</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>wang</td>
<td>1.01</td>
<td>1.00</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>typecheck</td>
<td>1.11</td>
<td>1.01</td>
<td>1.00</td>
<td>1.07</td>
<td>0.92</td>
</tr>
<tr>
<td>transform</td>
<td>1.04</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>sched</td>
<td>1.17</td>
<td>1.02</td>
<td>1.00</td>
<td>1.18</td>
<td>1.05</td>
</tr>
<tr>
<td>nucleic2</td>
<td>1.16</td>
<td>1.01</td>
<td>1.00</td>
<td>1.01</td>
<td>1.03</td>
</tr>
<tr>
<td>parstof</td>
<td>1.05</td>
<td>1.00</td>
<td>1.06</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcopy</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>listcompr</td>
<td>1.03</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>genfft</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fft</td>
<td>1.04</td>
<td>1.00</td>
<td>1.05</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>event</td>
<td>1.08</td>
<td>1.01</td>
<td>1.09</td>
<td>1.01</td>
<td>1.04</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.92</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.17</td>
<td>1.02</td>
<td>1.06</td>
<td>1.18</td>
<td>1.05</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.06</td>
<td>1.01</td>
<td>1.02</td>
<td>1.02</td>
<td>1.00</td>
</tr>
</tbody>
</table>

**Table 6.24** Buildify, catify and the cata-build rule: the Hartel Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Time to compile</th>
<th>Binary size</th>
<th>Time to run</th>
<th>Total allocation</th>
<th>Max Heap Residency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebnf2ps</td>
<td>1.03</td>
<td>1.00</td>
<td>0.72</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>fem</td>
<td>1.07</td>
<td>1.01</td>
<td>0.85</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gamteb</td>
<td>1.03</td>
<td>1.00</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>grep</td>
<td>1.01</td>
<td>1.00</td>
<td>0.90</td>
<td>1.00</td>
<td>NA</td>
</tr>
<tr>
<td>hpg</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>infer</td>
<td>1.05</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00</td>
<td>0.98</td>
</tr>
<tr>
<td>lift</td>
<td>1.04</td>
<td>1.00</td>
<td>1.08</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>maillist</td>
<td>1.01</td>
<td>1.00</td>
<td>1.14</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>mkhprog</td>
<td>1.04</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>parser</td>
<td>1.32</td>
<td>1.04</td>
<td>0.94</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>pic</td>
<td>1.01</td>
<td>1.00</td>
<td>0.84</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>prolog</td>
<td>1.03</td>
<td>1.00</td>
<td>0.92</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>reptile</td>
<td>1.03</td>
<td>1.00</td>
<td>0.79</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>rsa</td>
<td>1.01</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>veritas</td>
<td>1.10</td>
<td>1.03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Minimum</td>
<td>1.01</td>
<td>1.00</td>
<td>0.72</td>
<td>1.00</td>
<td>0.98</td>
</tr>
<tr>
<td>Maximum</td>
<td>1.32</td>
<td>1.04</td>
<td>1.14</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Geometric mean</td>
<td>1.05</td>
<td>1.01</td>
<td>0.94</td>
<td>1.00</td>
<td>0.99</td>
</tr>
</tbody>
</table>

**Table 6.25** Buildify, catify and the cata-build rule: the real subset
Chapter 7

Conclusions and Further Work

In this thesis, we have demonstrated that warm fusion is a practical approach for the removal of intermediate data structures within a real, production quality compiler for Haskell. We also have seen that the techniques required to implement warm fusion are a higher level — higher complexity — of transformations compared to most of those reported, for example, in Santos thesis [San95]: some bits are conditional, sometimes other transformations are needed to find out that warm fusion cannot proceed further. Contrasting this with those in Santos thesis, it is clear that his transformations are unconditional and almost always result in a benefit: decreased heap allocation or runtime improvement. The transformations of the warm fusion method are not always beneficial, in fact, both buildify and catify has been shown to increase heap allocation and runtime unless the cata-build rule gets applied to the transformed functions.

Through the implementation we discovered that warm fusion, being a higher level transformation often stretches the capabilities of the compiler. Our findings, which can also be considered as suggestions for a new implementation — both for GHC and the warm fusion transformation — are as follows:

- GHC’s inliner cannot cope with the complexity of the conditions required to efficiently implement warm fusion. We were often forced to have many passes of the simplifier instead of one, which leads to increased compilation times.

- GHC’s philosophy is often quite different from what warm fusion requires. In particular, in order to successfully buildify we sometimes need the wrappers of already catified functions. This mismatch is particularly painful with conditional transformations, where the problem of reversal arises.

- Recent work by Chitil [Chi00] demonstrates that build can be dispensed, because his
type system can predict when buildify is successful. A new design incorporating this observation should be somewhat simpler in terms of implementation, as after type inference all the functions which can be buildified would be properly annotated, so the transformation buildify would cease to be conditional.

- The two transformations presented in this thesis are quite complex. Their interaction with other transformations (see Table A.1 and Santos’s thesis [San95]) is even more so. This has two consequences:

  1. Fusion transformation can be quite unpredictable for the user, and sometimes even for the implementor.
  2. It is hard to insert the new transformations into the standard sequence of passes and guarantee that the new sequence always results in better programs.

If the current fusion engine is extended for example to apply to datatypes with embedded functions (Section 7.1.5) or to allow fusion for functions with multiple inductive arguments these interactions may become intractably complex. In this case, the use of some sort of guarantee that the transformations do indeed improve the code, for example improvement theory [San96b, San96a] will be unavoidable.

## 7.1 Further Work

One of the most exciting aspect of the work presented in this thesis is that by putting a lot of theory into practice, it opened up many avenues for further exploration.

### 7.1.1 Automatically deriving code from types

We have shown that in order to transform arbitrary functions to build-cata form we need the definitions of a few functions: `cata`, `build`. Sometimes we also need the appropriate type functor or `map`. These functions exist for a certain class of datatypes. It is known that other functions also exist: for example a `length` kind of function always exist for polynomial datatypes. `zip` style functions between any two types also exist for a large class. Functions whose existence is guaranteed, should be derived by the compiler automatically from the type declaration (data) and made available to the user. This would have several advantages:

- It would simplify the Standard Prelude, since `map`, `foldr` etc would not need to be defined there.
• The derivable functions need not be written by the user.

• The derivable functions would be unique within the compiler, possibly leading to the opportunity of generating better code for them.

• Encourage a style of programming in which simply declaring a type would result in functions over that type. The idea of this style, albeit in a seemingly different context, is not a new one: in the HOL theorem prover [GM93] declaring a type results in theorems about it. For example, the existence of a unique, primitive recursion operator can be asserted for a large class of datatypes from the declaration. The system then efficiently proves these theorems [Mel88], which happens to be almost the same as what we called deriving catamorphisms (see Sections 4.5.2, 5.2.2) in this thesis.

Perhaps this could be the starting point of connecting (a compiler for) Haskell with a theorem prover, thereby increasing the power of transformation methods and increasing the confidence in the correctness of the generated code.

7.1.2 Special abstract machine for fused programs

We noted in Chapter 6 that warm fusion tends to produce lots of higher-order functions in the resulting code, and STG seems to be ill-suited for efficient execution of such code. It would be interesting to see, if other abstract machines used for executing functional languages cope can better.

7.1.3 Transparency of transformations

Warm fusion is not a transparent program transformation, meaning that it is hard for the user to predict if the transformation applies or not. For efficiency conscious programmers this presents a dilemma: they can try to write optimised code — which in some cases has the embarrassing effect of disabling other built-in optimisations — or hope for the best. If we contrast this situation with simpler, traditional, perhaps better understood optimisations or the transparency provided by the MAG system [DMS99] we realise the need to provide feedback not just when warm fusion is successful, but also when and why it fails. How to provide this feedback and what form it should take is currently unknown, but its deeper understanding may lead to wider acceptance of higher level transformations.
7.1.4 More aggressive inlining

In our implementation, applicability of the \texttt{cata-build} rule depends entirely on inlining of the wrapper functions. It is therefore of utmost importance that these functions are inlined at every possible call site. Unfortunately, inlining have two major risks: code duplication and duplication of computations. Duplication of computations can arise when we inline across lambdas. In certain cases a linear type system or usage analysis [TWM95, WPJ99] can ensure that inlining is without this risk. Warm fusion would certainly benefit from these analyses.

Another problem with inlining concerns the Glasgow Haskell Compiler itself. We are forced to have multiple runs of simplification over the module being compiled, because we want one pass of simplification to happen and only then have inlining. Since this cannot currently be expressed in the simplifier we need to have one pass with inlining disabled and a second one to get the effects of inlining.

This only affects compilation time, but finer control over inlining — for example some form of conditional inlining — would make the warm fusion transformation faster and simpler to implement.

7.1.5 Fusion for datatypes with embedded functions

The first theoretical proposal to handle datatypes with embedded functions is the one by Meijer and Hutton [MH95] based on Freyds work [Fre90]. Fegaras and Sheard [FS96] suggested a more implementable way. Their proposal requires three modifications to the work reported in this thesis:

- The deriving mechanism (see Section 4.5.2) needs to be modified:

  1. by adding a fictitious constructor, \texttt{Place }\alpha, acting as a placeholder, to every datatype and catamorphism which uses embedded functions.
  2. within the catamorphism, the action of the constructor which uses the embedded function needs to be slightly altered and a new \texttt{case} alternative needs to be added which deals with the fictitious constructor.

Despite of these modifications, the only change to the type of the catamorphisms is an extra type argument for \(\alpha\). Nothing else changes, apart from the recursive uses of the type being defined, where the extra type argument is needed, since the \texttt{Place} constructor remains hidden from the user.
• The typechecker needs to be modified to restrict the uses of the new constructor.

• The \texttt{cata-build} rule and other rules defining the interaction between catamorphisms and Core needs to be changed to accommodate the extra type argument.

These modifications seem to be quite simple, but interaction with other extensions (Section 5.1 and Section 5.2) needs to be thoroughly investigated.

### 7.1.6 Fegaras style folds

In their 1994 PEPM paper, Fegaras, Sheard and Zhou [FSZ94] suggested a new form of catamorphisms, and the corresponding binary fusion theorem to handle functions which induct on two arguments. Their method can perform fusion on both arguments for example on the well-known \texttt{zip} function, which have been used as a benchmark to compare the relative strengths of different deforestation methods [HIT97]. Their work can, in theory, be easily generalised to functions with an arbitrary number of inductive arguments, but the extension does not fit into our framework. We started the theory chapter, Chapter 3, with a quotation from the bananas paper [MFP91], which is a fundamental assumption of our work. We derive folds and maps, once for all after the desugarer, from the type constructor, while they derive their fold operators on a per-function basis. In other words, in the current framework all functions consuming arguments of type list use the same fold operator, while in their framework, a function which consumes a single list (e.g. \texttt{filter}) would use the familiar fold operator, while another function (e.g. \texttt{zip}, or structural equality) would use a different one, and could only be fused with the use of a different fusion law!

Incorporating their fusion method into GHC would certainly result in serious penalty regarding compilation times.

### 7.1.7 Monadic maps, folds and fusion

Catamorphisms are control structures that exactly match the datatypes they belong to, in other words, folding structures functions by the way they consume their arguments. An alternative is to structure computations by the way they compute their results, by using monads [Mog91, Wad92, WPJ93, Wad95]. It is possible to combine these two approaches, as it was shown by Fokkinga [Fok94] and later by Meijer and Jeuring [MJ95]. The usefulness of their approach is amply demonstrated in the later paper.

Incorporating a monadic fusion engine into GHC raises several problems:
1. Many simple functions are hard to express in terms of a monadic fold, that is the recursive patterns captured by monadic folds are often to specific to be useful.

2. The deriving mechanism (see Section 4.5.2) can be extended to automatically derive monadic maps and folds, but the existence of these functions for a given datatype depends on a side condition [Fok94, paragraph 5.1] on the monad. Verifying this condition seems to be rather hard in general — may even require a theorem prover — and it is known not to hold for several monads, for example the state monad.

3. In the desugaring phase (see page 142) of the Glasgow Haskell Compiler, the monadic structure of the original program is lost, because the definitions of the two functions, which constitute a monad — together with the given type constructor — often called bind and result, are inlined for efficiency. For reasons we explained in Section 4.4.2, maps and folds are derived after the desugarer. Since we need the monadic structure to be able apply the monadic fusion law, we would need to modify the desugarer not to inline bind and result. This requires a major rethinking, restructuring of the compiler and may have a far reaching consequences on compilation time and the efficiency of generated code.

7.1.8 Warmer fusion

Catamorphisms represent structural induction over datatypes. Together with tupling and currying, they are capable of representing primitive recursive functions. A more natural framework to deal with primitive recursive functions could be based on Meertens work [Mee90], since paramorphisms directly correspond to primitive recursive functions. Most of the techniques, for example transforming an arbitrary function to catamorphic form by composing it with the identity catamorphism, carries over to paramorphisms, which may lead to a simpler design for a transformation system centred around the concept of paramorphisms or it may lead to a more powerful transformation engine.
Appendix A

The Framework

In this chapter we give a short introduction to the Glasgow Haskell Compiler (GHC 3.03), on which the design and the first implementation is based. The definitive, though rather outdated, description is Santos’ thesis [San95]. Newer accounts are [PJS96, PJ96]. Section A.1 details the main passes of the compiler before the incorporation of the fusion engine. Section A.3 summarises the changes as the result of this thesis. The rationale for these changes are given in Chapter 4.

A.1 The compiler (pre-warm fusion)

The compiler has a modular design. The compilation process consists of a series of correctness-preserving transformations, which are shown in Figure A.1. The main passes, which follow one another in the order given are:

- **reader**
  
  Written in Lex and Yacc.

- **renamer**
  
  Resolves scoping and naming issues and makes identifiers unique.

- **type inference**
  
  Annotates the program with type information.

- **desugarer**
  
  Transforms the high level constructs of Haskell (like pattern matching, and list comprehensions) into 2\textsuperscript{nd}-order lambda calculus, which in GHC terminology is called the
A.1. THE COMPILER (PRE-WARM FUSION)

Core language. Its abstract syntax is given in Figure A.2.

- **core-simplifier**
  A series of transformation passes over Core that aim at improving the efficiency of the code.

- **core-to-stg**
  Translator from Core to the Shared Term Graph STG [PJ92] language.

- **stg-transformations**
  A few more transformations, now on STG language.

- **code-generator**
  A pass which converts STG language to Abstract C, or generates assembly code directly.

We will be mostly concerned with the core-simplifier, which also consists of many passes over Core programs. Note that core-simplifier passes are functions from Core to Core, they can be performed any number of times and in any order. The sequence of these passes is governed by a Perl (gasp) script; ordering does matter and picking the right ordering — which gives the best performance — can best be described as a Black Art. The most important ones are, in the order they are performed in GHC 3.03:

- **simplify**
  Performs local transformations (see Table A.1): beta-reduction, inlining, case elimination, case merging, eta expansion etc.

- **specialise**
  Eliminates overloading.

- **simplify**
  Performs local transformations (see Table A.1): beta-reduction, inlining, case elimination, case merging, eta expansion etc.

- **float-out**
  Full laziness transformation.

- **float-in**
  The opposite of full laziness.
A.2. THE SIMPLIFIER

- **simplify**
  Performs local transformations: beta-reduction, inlining, case elimination, case merging, eta expansion etc.

- **strictness analysis**
  This annotates identifiers with their strictness properties.

- **simplify**
  Performs local transformations: beta-reduction, inlining, case elimination, case merging, eta expansion etc.

- **float-in**
  The opposite of full laziness.

- **simplify**
  Performs local transformations: beta-reduction, inlining, case elimination, case merging, eta expansion etc. This is the final clean up simplification.

Santos [San95] devotes a whole chapter of his thesis to the discussion of the constraints, which a good sequence should satisfy and presents the one shown above. One would like to see this process of simplification formulated as a rewrite system and to see the proofs of a few desirable (confluence, termination) properties. Unfortunately, neither confluence nor termination holds.

### A.2 The simplifier

At the very heart of the compiler, there is the simplifier. It implements a set of local transformations and its primary aims are twofold:

- some transformations *remove* Core constructs: \(\beta\)-reduction, let elimination, case elimination;

- some transformations *move* Core constructs: let-floating, case floating.

The simplifier is also used to 'clean up' mess after transformations. Sometimes, it is just too inconvenient/hard/complex to write code (within the compiler) which produces the best possible code. For example, when pieces of code become 'dead' one would have to combine
### Table A.1 Local transformations

<table>
<thead>
<tr>
<th>Rule</th>
<th>Before</th>
<th>After</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>beta reduction</td>
<td>((\lambda , v. , e) , x)</td>
<td>(e[x/v])</td>
<td></td>
</tr>
<tr>
<td>typed beta reduction</td>
<td>((\Lambda , \tau. , e) , \sigma)</td>
<td>(e[\sigma/\tau])</td>
<td></td>
</tr>
<tr>
<td>dead code removal</td>
<td>(\text{let } _v = e_v \text{ in } e)</td>
<td>(e)</td>
<td>(v) doesn’t occur free in (e)</td>
</tr>
<tr>
<td>inlining</td>
<td>(\text{let } _v = e_v \text{ in } e)</td>
<td>(\text{let } _v = e_v \text{ in } e[e_v/v])</td>
<td>several see Santos’s thesis [San95]</td>
</tr>
<tr>
<td>case of known constructor</td>
<td>\text{case } C_i , v_1 \ldots , v_n \text{ of } C_1 \ldots \rightarrow e_1</td>
<td>(e_i[v_1/w_1 \ldots v_n/w_n])</td>
<td></td>
</tr>
<tr>
<td>case of error</td>
<td>\text{case error } E \text{ of } :</td>
<td>\text{error } E</td>
<td></td>
</tr>
<tr>
<td>case elimination</td>
<td>\text{case } v_i \text{ of } e</td>
<td>(e[v_i/v_j])</td>
<td></td>
</tr>
<tr>
<td>let to case</td>
<td>(\text{let } _v = e_v \text{ in } e)</td>
<td>\text{case } e_v \text{ of } _v \rightarrow e</td>
<td>(e) is strict in (v) and (e_v) is not in weak head normal form</td>
</tr>
</tbody>
</table>

The given transformation with dead-code elimination, which would introduce unnecessary complications.

We give a list of rewrite rules, which are needed for warm fusion to work in Table A.1. Santos [San95] calls these rules local transformations. These will be referred to in the body of the thesis by their names without further discussion. The interested reader is again referred to Santos’ thesis [San95] for a thorough discussion of these rules.

The main points to be noted about Core are:

- **Explicit type abstraction and type application.**
- **Atomic arguments.** The arguments of an application or constructor are atomic (variables, literals or types).
- **Applications of constructors and primitive operations are saturated.**
- **Core programs have a direct operational interpretation.**

1. All heap allocation is represented by `let`s.
2. evaluation is always denoted by case.

This means that the case construct of Haskell is not the same as the case construct of Core. In this thesis, all case constructs are considered to be strict, that is they are of the Core variety.

A.3 The compiler (post-warm fusion)

Adding the fusion engine to GHC 3.03 does not result in deep structural changes in the compiler. A new pass (derive) is added to the main compilation process.

- **reader**
  Written in Lex and Yacc.

- **renamer**
  Resolves scoping and naming issues and makes identifiers unique.

- **type inference**
  Annotates the program with type information.

- **desugarer**
  Transforms the high level constructs of Haskell (like pattern matching, and list comprehensions) into 2\textsuperscript{nd}-order lambda calculus, which in GHC terminology is called the Core language. Its abstract syntax is given in Figure A.2.

- **derive**
  The existence of certain functions is guaranteed by their types. The existence is explained in Chapter 3 and the deriving process is explained at length in Section 4.5.2.

- **core-simplifier**
  A series of transformation passes over Core that aim at improving the efficiency of the code.

- **core-to-stg**
  Translator from Core to the Shared Term Graph STG [PJ92] language.

- **stg-transformations**
  A few more transformations, now on STG language.
• **code-generator**
  A pass which converts STG language to Abstract C, or generates assembly code directly.

The core-simplifier is the pass which is most affected by the fusion transformation. The new passes normalise, warm fusion (which consists of many simpler passes), static argument transformation are detailed in Chapter 4.

• **simplify**
  Performs local transformations (see Table A.1): beta-reduction, inlining, case elimination, case merging, eta expansion etc.

• **specialise**
  Eliminates overloading.

• **normalise**
  Rearranges the arguments of functions to a 'standard' order. This is explained in Section 5.4.

• **simplify**
  Performs local transformations (see Table A.1): beta-reduction, inlining, case elimination, case merging, eta expansion etc.

• **float-out**
  Full laziness transformation.

• **warm fusion**
  What this thesis is about. It consists of two transformations: buildify (see Sections 4.5.4, 5.1.3, and 5.2.4) and catify (Sections 4.5.5, 5.1.4, and 5.2.5). Between buildify and catify, there is a simplify pass and in some cases a static argument transformation (Section 5.1.6).

• **float-in**
  The opposite of full laziness.

• **simplify**
  Performs local transformations: beta-reduction, inlining, case elimination, case merging, eta expansion etc.
A.3. THE COMPILER (POST-WARM FUSION)

- **strictness analysis**
  This annotates identifiers with their strictness properties.

- **simplify**
  Performs local transformations: beta-reduction, inlining, case elimination, case merging, eta expansion etc.

- **float-in**
  The opposite of full laziness.

- **simplify**
  Performs local transformations: beta-reduction, inlining, case elimination, case merging, eta expansion etc. This is the final clean up simplification.

There is an additional set of rules, which describe how the newly introduced constructs (cata, build) interact with the rest of Core. These are described in the chapter dealing with the practice of warm fusion.
A.3. THE COMPILER (POST-WARM FUSION)

Figure A.1 Glasgow Haskell Compiler passes
<table>
<thead>
<tr>
<th>Program</th>
<th>( \text{Prog} ::= \text{TopDecl}_1 ; \ldots ; \text{TopDecl}_n \quad n \geq 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations</td>
<td>( \text{TopDecl} ::= \text{Binding} \mid \text{TypeDecl} )</td>
</tr>
<tr>
<td>Declaration</td>
<td>( \text{TypeDecl} ::= \text{data}\ \text{Con} \ \vec{\alpha} = {C_i \ \vec{\tau}<em>i}</em>{i=1}^n )</td>
</tr>
<tr>
<td>Types</td>
<td>( \tau ::= \text{TyCon}[\tau] \quad \text{Constructor application} )</td>
</tr>
<tr>
<td></td>
<td>( \tau \rightarrow \tau' \quad \text{Function space} )</td>
</tr>
<tr>
<td></td>
<td>( \forall \alpha.\tau \quad \text{Universal quantification} )</td>
</tr>
<tr>
<td></td>
<td>( \alpha \quad \text{Type variable} )</td>
</tr>
<tr>
<td>Bindings</td>
<td>( \text{Binding} ::= \text{Bind} \mid \text{rec}\ \text{Bind}_1 \ldots \text{Bind}_n )</td>
</tr>
<tr>
<td></td>
<td>( \text{Bind} ::= \text{var} :: \tau = \text{Expr} )</td>
</tr>
<tr>
<td>Expression</td>
<td>( \text{Expr} ::= \text{Expr}\ \text{Atom} \quad \text{Application} )</td>
</tr>
<tr>
<td></td>
<td>( \text{Expr}\ \tau \quad \text{Type application} )</td>
</tr>
<tr>
<td></td>
<td>( \lambda\ \text{var}_1 :: \tau_1 \ldots \text{var}_n :: \tau_n.\text{Expr} \quad \text{Lambda abstraction} )</td>
</tr>
<tr>
<td></td>
<td>( \Lambda\ \text{ty} .\ \text{Expr} \quad \text{Type abstraction} )</td>
</tr>
<tr>
<td></td>
<td>( \text{case}\ \text{Expr}\ \text{of}\ \text{Alts} \quad \text{Case expression} )</td>
</tr>
<tr>
<td></td>
<td>( \text{let}\ \text{Binding}\ \text{in}\ \text{Expr} \quad \text{Local definition} )</td>
</tr>
<tr>
<td></td>
<td>( \text{con}\ \text{var}_1 \ldots \text{var}_n \quad \text{Constructor} \ n \geq 0 )</td>
</tr>
<tr>
<td></td>
<td>( \text{prim}\ \text{var}_1 \ldots \text{var}_n \quad \text{Primitive} \ n \geq 0 )</td>
</tr>
<tr>
<td></td>
<td>( \text{Atom} )</td>
</tr>
<tr>
<td>Atoms</td>
<td>( \text{Atom} ::= \text{var} :: \tau \quad \text{Variable} )</td>
</tr>
<tr>
<td></td>
<td>( \text{Literal} \quad \text{Unboxed Object} )</td>
</tr>
<tr>
<td>Literal values</td>
<td>( \text{Literal} ::= \text{integer} \mid \text{float} \mid \ldots )</td>
</tr>
<tr>
<td>Alternatives</td>
<td>( \text{Alts} ::= \text{Calt}_1; \ldots ; \text{Calt}_n;\ \text{Default} \quad n \geq 0 )</td>
</tr>
<tr>
<td></td>
<td>( \text{Lalt}_1; \ldots ; \text{Lalt}_n;\ \text{Default} \quad n \geq 0 )</td>
</tr>
<tr>
<td>Constr. alt</td>
<td>( \text{Calt} ::= \text{Con}\ \text{var}_1 \ldots \text{var}_n \rightarrow \text{Expr} \quad n \geq 0 )</td>
</tr>
<tr>
<td>Literal alt</td>
<td>( \text{Lalt} ::= \text{Literal} \rightarrow \text{Expr} )</td>
</tr>
<tr>
<td>Default alt</td>
<td>( \text{Default} ::= \text{NoDefault} \mid \text{var} \rightarrow \text{Expr} )</td>
</tr>
</tbody>
</table>

**Figure A.2** Syntax of the Core language
Bibliography


[Wad85a] Philip Wadler. How to replace failure by a list of successes. In Jouannaud [Jou85], pages 113–128.


[Wad95] Philip Wadler. Monads for functional programming. In Jeuring and Meijer [JM95].


Index

abstract datatype, 91
accumulating argument, 73
Acid Rain theorem, 12, 24, 42
ADT, 91
Alg(F), category of algebras, 19
algebraic replacement rule, 9
algorithm for
  buildify, 55, 69
catify, 60, 70, 81
beta reduction, 32, 145
build-cata form, 13
buildify, 55, 69
case elimination, 145
case of error, 145
case of known constructor, 83, 145
cata fusion law, 43
cata of case rule, 57, 79
cata of error rule, 79
cata of known constructor rule, 20, 57, 79, 103
cata-build rule, 12, 13, 25, 37, 38, 44, 55, 79
catamorphism, 12, 20
evaluation rule, 20
fusion law, 21
reflection law, 20
catify, 60, 70, 81
compiler passes
  core simplifier, 143, 146
derive, 146
desugarer, 64, 142, 146
reader, 142, 146
renamer, 142, 146
type inference, 142, 146
copy function, 20, 43, 141
core simplifier, 44, 56, 143, 146
float in, 147
full laziness, 147
normalise, 147
simplify, 147
specialise, 147
strictness analysis, 148
warm fusion, 147
cut elimination, 11
dead code removal, 145
definition of
  E, 57, 78
  M, 50, 77
catamorphism, 20
map, 50
polynomial datatype, 37
polynomial functor, 19
regular datatype, 37
rewrite system, 84
the functor E, 52, 78
the functor M, 50, 77
type functor, 50
definition rule, 9
deforestation, 7, 10, 12
desugarer, 64, 142, 146
dynamic rewrite system, 35

E, 52, 57
evaluation rule for catamorphism, 20
evaluation rule for catamorphism, 20
example
catamorphism
cataTree α, 67
first-order catamorphism
append, 65
for buildify
mapMaybe, 58
mapRose, 59
downTo, 56
level, 68
repAnswer, 30
for catify
length for List, 62
level, 69
map[], 72
mutually recursive maps, 82
sum, 33
for deriving
map[], 51
for deriving a catamorphism
List cata[α], 52
Rose tree cataRose α, 53
higher-order catamorphism
append, 66
level, 69
map[], 73
optimal translation of list comprehensions, 96
standardising argument ordering, 87
existence of
catamorphisms, 20
existential quantification, 91
F-algebra, 19
F-homomorphism, 19
float in, 147
float out, 147
folding rule, 9
tofoldl, 8
tofoldr, 12, 25, 53
tofoldr/build rule, 12, 25
full laziness, 48, 147
fusion law
for catamorphism, 21, 70
monadic, 141
generate, 8
good consumer, 32, 39
good producer, 29, 39, 41, 56
HOL, 138
identity catamorphism, 43, 141
improvement theory, 137
inlining, 9, 145
instantiation rule, 9
interaction of catas and Core
cata-build rule, 55
cata of case rule, 55
cata of error rule, 55
cata of known constructor rule, 55
let to case, 145
list comprehension
optimal translation of, 92
local transformations
beta reduction, 145
case elimination, 145
case of error, 145
case of known constructor, 145
dead code removal, 145
inlining, 145
let to case, 145
type beta reduction, 145
M, 50
map, 73
map for list, 8
monoConstrs, 52, 55, 80
natural transformation, 15
normalise, 87, 147
notStatic, 74
optimal translation, 92
parametricity theorem, 14
partial evaluation, 33
polynomial datatype, 37
polynomial functor, 19
program calculation, 5
program derivation, 5
program transformation
rules and strategies approach, 5
schemata approach, 5
promotion theorem, 43, 70
reader, 142, 146
reflection law for catamorphisms, 20
regular datatype, 37
renamer, 142, 146
rewrite system, 35, 84
Rose tree, 53, 59
RULES, 118
rules
cata-build rule, 79
case of known constructor, 83
cata of case, 55, 57, 79
cata of error, 55, 79
cata of known constructor, 20, 55, 57, 79, 103
rules and strategies approach, 5
SAT, 74
schemata approach, 5
second-order fusion, 48
second-order fusion theorem, 73
separate compilation, 89
simplify, 147
snoc list, 8
sourceTypeOf, 50, 77, 78
specialise, 147
Squiggol, 10, 12
Standard Prelude, 12, 53, 67, 117
static argument transformation, 74
strictness, 30, 43, 60
strictness analysis, 48, 148
structural recursion, 40
targetTypeOf, 50, 52, 78
theorem prover, 138
transparency of transformations, 11, 17, 138
TyConOf, 77
type beta reduction, 145
type functor, 137
type inference, 142, 146
typeOf, 50, 77, 78
tyVarsOf, 50, 77
unfolding rule, 9
warm fusion, 13, 147
where abstraction rule, 9
worker-wrapper split, 31, 61