UNIVERSITY
of
GLASGOW

Department of Computing Science

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

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

List of figures

List of tables

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 Weeden, 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 *sum*, *map*, and *upto*) to compute the sum of the squares from 1 to n:

$$foo \;=\; sum \;.\; map\ square \;.\; upto\ 1$$

In this case the intermediate list $[1, 2, \ldots, n]$ connects the functions *upto* and *map square*. Another intermediate list $[1, 4, \ldots, n^2]$ connects *map square* with *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:

$$\begin{array}{lll} foo & = & bar\ 1 \\ bar\ l\ u & = & \textbf{case}\ l \ \leq\ u\ \textbf{of} \\ & & \quad True \ \rightarrow\ l^2\ +\ bar\ (l+1)\ u \\ & & \quad False \ \rightarrow\ 0 \end{array}$$

Intermediate data structure removal algorithms attempt to automatically turn the former defintion 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.

- 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

$$
\begin{tikzcd}
P_0 \arrow[r] \arrow[dr] & P_1 \arrow[r] \arrow[d] & \cdots \arrow[r] & P_n \arrow[dll] \\
 & V & &
\end{tikzcd}
$$

Figure 2.1 The idea of program transformation

buildify and catify is closer in spirit to the rules and strategies approach.

The motivation for deriving a new program, P_2, from and old one, P_1, is typically that we want to improve some aspect of P_1 while preserving its semantics: we want $Sem(P_1) = Sem(P_2)$ for some given semantic function. More precisely, we want the equivalence induced by 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, $Sem(P_0) = Sem(P_n)$ for some given semantic function Sem. This is often done in more than one step, by constructing a sequence $< P_0, \ldots, P_n >$ of programs such that $Sem(P_i) = Sem(P_{i+1})$ 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.

Method	Data types	Language	Condition	Implementation
Schema deforestation [Wad81]	list	higher order	syntactic condition: only for expression written in terms of the basic combinators: *map*, *foldl*, *generate*	NA
Listless transformer [Wad86]	list	first order	preorder traversal	no
Deforestation [Wad90]	polynomial	first order (+ non-recursive, higher order macros)	functions in treeless form	prototype
Chin's extensions [Chi92b]	polynomial	higher order	accepts all functions but deforestation does not take place for non-treeless terms. Higher-order functions are removed	??
Higher order deforestation [Mar95]	polynomial	higher order	??	yes, GHC
Supercompilation [Tur86]	polynomial	higher order	??	NA
Cheap deforestation [GLPJ93]	list	higher order	fixed set of functions (from the Standard Prelude)	yes, GHC
Warm Fusion [LS95] and this thesis	polynomial	higher order	syntactic condition	yes, GHC
Hylo Fusion [Hu96, OHIT97]	polynomial	higher order	structural hylomorphisms	in progress, GHC

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*.

$$
\begin{array}{ll}
\mathit{map}\, f\ (\mathit{map}\ g\ \mathit{xs}) & = \mathit{map}\ (f\ .\ g)\ \mathit{xs} \\
\mathit{foldl}\, f\ a\ (\mathit{map}\ g\ \mathit{xs}) & = \mathit{foldl}\ h\ a\ \mathit{xs} \\
\quad \textbf{where} & \\
\qquad h\ a'\ x\ =\ f\ a'\ (g\ x) & \\
\mathit{map}\, f\ (\mathit{generate}\ p\ g1\ g2\ x) & = \mathit{generate}\ p\ h\ g2\ x \\
\quad \textbf{where} & \\
\qquad h\ b'\ =\ f\ (g1\ b') & \\
\mathit{foldl}\, f\ a\ (\mathit{generate}\ p\ g1\ g2\ x) & = h\ a\ x \\
\quad \textbf{where} & \\
\qquad h\ a'\ b' \mid p\ b' \qquad\qquad = a' & \\
\qquad h\ a'\ b' \mid \textbf{otherwise}\ = h\ (f\ a'\ (g1\ b'))\ (g2\ b') &
\end{array}
$$

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

$$
\begin{array}{ll}
\mathit{map} & :: \forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\
\mathit{map}\, f\ [] & = [] \\
\mathit{map}\, f\ (x : \mathit{xs}) & = f\ x\ :\ \mathit{map}\, f\ \mathit{xs}
\end{array}
$$

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

$$
\begin{array}{ll}
\mathit{foldl} & :: \forall\alpha\beta.(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta \\
\mathit{foldl}\, f\ z\ [] & = z \\
\mathit{foldl}\, f\ z\ (x : \mathit{xs}) & = \mathit{foldl}\, f\ (f\ z\ x)\ \mathit{xs}
\end{array}
$$

Finally, *generate* is used to express list production:

$$
\begin{array}{ll}
\mathit{generate} & :: \forall\ \alpha\beta.(\alpha \rightarrow \mathit{Bool}) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \\
\mathit{generate}\ p\ f\ g\ n \mid p\ n & = [] \\
\mathit{generate}\ p\ f\ g\ n & = f\ n\ :\ \mathit{generate}\ p\ f\ g\ (g\ n)
\end{array}
$$

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, Par90, Chi93, CK93, HIT96b, HITT97], 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 $f\ .\ g$, 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 *map f . 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 *foldr* list data type **data** $[\alpha] = [] \mid \alpha : [\alpha]$ the *foldr* function is defined in the following way:

$$\begin{array}{ll} \mathit{foldr} & :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\ \mathit{foldr}\ c\ n\ [] & = [] \\ \mathit{foldr}\ c\ n\ (x : xs) & = c\ x\ (\mathit{foldr}\ c\ n\ xs) \end{array}$$

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

To standardise the *production* of lists, they introduced a function *build* with type and definition:

$$\begin{array}{ll} \mathit{build} & :: \forall\,\alpha.(\forall\,\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha] \\ \mathit{build}\ g & = g\ (:)\ [] \end{array}$$

and the following property:

$$\mathit{foldr}\ c\ n\ (\mathit{build}\ g) = g\ c\ n \tag{2.1}$$

which appears in the literature under the names `foldr/build` rule, `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 *build*, and the result is consumed by *foldr* then the intermediate list need not be built, the result can be constructed by passing *foldr* first two argument directly to g. Each application of the foldr-build rule can be seen as a canned application of unfold/simplify/fold in the traditional deforestation framework. Unfortunately, at that time *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 *foldr* 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 Kieburtz [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 *fold* (or catamorphism), production is expressed using an explicit *build*. It generalises the cheap deforestation work by extending the `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 `build-cata` form with an explicit *fold* and *build*, which, when successful, allows the application of the one-step fusion rule, the `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 *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 *zip*. Later he abandoned the fixed pattern of recursion idea, (*fold*), returned to the basics and suggested the direct use of the *parametricity theorem* [Rey83, Wad89] to fusion [Feg96].

2.7 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, Chitil [Chi99] uses the type system to predict when the transformation to explicit *build* form can be successful. More recently [Chi00], he managed to dispense *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 `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.

first. On the other hand we say `cata-build` *rule* because that describes the application of a *cata* to a *build* (also read from left to right).

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: $[\![\varphi, \psi]\!] = (\!|\varphi|\!) \cdot [\!(\psi)\!]$) and its associated fusion laws. An extension of the basic system is suggested in [HIT97], 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.

$$\begin{array}{lll} f & :: & B \to A \\ f & \Rightarrow & [\![\varphi, \eta, \psi]\!]_{G,F}, \textbf{where } \varphi :: (\mathsf{G}A \to A),\ \eta :: (\mathsf{F} \dot{\to} G),\ \psi :: (B \to \mathsf{F}B) \end{array} \tag{2.2}$$

Note, that η 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:

$$\text{Cata-Hylo Fusion} \tag{2.3}$$

$$\tau :: \forall A.(\mathsf{F}A \to A) \to \mathsf{F}A \to A \quad \Rightarrow \quad [\![\varphi, \eta_1, out_F]\!]_{G,F} \cdot [\![\tau in_F, \eta_2, \psi]\!]_{F,L} = [\![\tau(\varphi \cdot \eta_1), \eta_2, \psi]\!]_{F,L}$$

$$\text{Hylo-Ana Fusion} \tag{2.4}$$

$$\sigma :: \forall A.(A \to \mathsf{F}A) \to A \to \mathsf{F}A \quad \Rightarrow \quad [\![\varphi, \eta_1, \sigma out_F]\!]_{G,F} \cdot [\![in_F, \eta_2, \psi]\!]_{F,L} = [\![\varphi, \eta_1, \sigma(\eta_2 \cdot \psi)]\!]_{G,F}$$

Note, that fusion can only take place (in the Cata-Hylo case for example), if the hylomorphism on the left ($[\![\varphi, \eta_1, out_F]\!]_{G,F}$) is really a catamorphism and the hylomorphism on the right is of a specific form $[\![\tau in_F, \eta_2, \psi]\!]_{F,L}$. 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

$$[\![\varphi, \eta, \psi]\!]_{G,F} = [\![\varphi \cdot \eta, id, \psi]\!]_{F,F} = [\![\varphi, id, \eta \cdot \psi]\!]_{G,G} \tag{2.5}$$

and appropriate τ, σ 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 removal. 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 ($Int, Bool, [Int]$), 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 $\mathcal{CPO}$, 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* $\mathbf{C}$*. An* F-algebra *is an arrow of type* $\mathsf{F}A \to A$*, the object* A *is called the* carrier *of the algebra.*

Definition 3.2 *An* F-homomorphism *to an algebra* $f : \mathsf{F}A \to A$ *from an algebra* $g : \mathsf{F}B \to B$ *is an arrow* $h : B \to A$ *such that* $h \cdot g = f \cdot \mathsf{F}h$*.*

Definition 3.3 *The objects of the category* $\mathbf{Alg}(\mathsf{F})$ *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 id and the constant functors* K_A *are polynomial;*
- *If* F *and* G *are polynomial, then so are their composition* FG*, their pointwise sum* $\mathsf{F}+\mathsf{G}$ *and their pointwise product* $\mathsf{F} \times \mathsf{G}$*. These pointwise functors are defined by*

$$
\begin{array}{rcl}
(\mathsf{F}+\mathsf{G})h & = & \mathsf{F}h + \mathsf{G}h \\
(\mathsf{F}\times\mathsf{G})h & = & \mathsf{F}h \times \mathsf{G}h
\end{array}
$$

- *If* F *is polynomial, then so is the type functor for* F

$$\mathsf{T}f = (\!| in \cdot \mathsf{F}(f, id) |\!) \tag{3.1}$$

Type functors only appear in Sections 4.5.1 and 5.2.1 in connection with datatypes like $T\ \alpha\ =\ T_1[T\ \alpha] \mid \ldots$ 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* **Alg**(F) *has an initial object and it will be called the* initial algebra*. It will be denoted* $in : \mathsf{F}T \rightarrow T$*. (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 $f : \mathsf{F}A \rightarrow A$, 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* $f : \mathsf{F}A \rightarrow A$ *is called a* catamorphism*. We will denote this homomorphism by* $(\!| f |\!)$*.* $(\!| f |\!) : T \rightarrow A$ *is characterised by the universal property*

$$h = (\!| f |\!) \equiv h \cdot in = f \cdot \mathsf{F}h$$

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

$$(\!| in |\!) = id \tag{3.2}$$

The *evaluation rule* for catamorphisms states how to evaluate an application of $(\!| f |\!)$ to an arbitary element of F (returned by in):

$$(\!| f |\!) \cdot in = f \cdot \mathsf{F}(\!| f |\!) \tag{3.3}$$

apply $(\!|f|\!)$ recursively to the argument of 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]

$$\frac{f \cdot \bot = g \cdot \bot \qquad (\forall x, y. f \cdot x = g \cdot y \Rightarrow f \cdot \varphi \cdot \mathsf{F}x = g \cdot \psi \cdot \mathsf{F}y)}{f \cdot (\!|\varphi|\!) = g \cdot (\!|\psi|\!)} \tag{CATAIND}$$

follows from the fixed point induction rule

$$\frac{P(\bot) \qquad (\forall a \in A. P(a) \Rightarrow P(f\ a))}{P(\mu f)} \tag{μ-IND}$$

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

Then there is the very useful *fusion law*

$$h \cdot (\!|f|\!) = (\!|g|\!) \Leftarrow h \cdot \bot = (\!|g|\!) \cdot \bot \wedge h \cdot f = g \cdot \mathsf{F}h \tag{3.4}$$

The fusion law states that the compostion 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 $(\!|f|\!)$ is applied to and builds another one, which is then traversed by h. The right-hand side however combines h and $(\!|f|\!)$ 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

$$
\begin{array}{ll}
 & \langle \text{base case} \rangle \\
 & h \cdot \bot = i \cdot \bot \\
 & \langle i = id \rangle \\
= & h \cdot \bot = \bot \\
 & \langle \text{induction step: assuming } h \cdot x = i \cdot y \rangle \\
= & h \cdot f \cdot \mathsf{F}x = i \cdot g \cdot \mathsf{F}y \\
 & \langle i = id \rangle \\
= & h \cdot f \cdot \mathsf{F}x = g \cdot \mathsf{F}y \\
 & \langle \text{hypothesis} \rangle \\
= & h \cdot f \cdot \mathsf{F}x = g \cdot \mathsf{F}(h \cdot x) \\
 & \langle \text{functor calculus} \rangle \\
= & h \cdot f \cdot \mathsf{F}x = g \cdot \mathsf{F}h \cdot \mathsf{F}x
\end{array}
$$

$$\begin{aligned}
&\langle \text{assume } h \cdot f = g \cdot \mathsf{F}h \rangle \\
= \; & \mathit{True}
\end{aligned}$$

A more useful variation on the fusion law is to replace the condition $h \cdot \perp = (\!|g|\!) \cdot \perp$ by $h \cdot \perp = \perp$, i.e. h is strict.

$$h \cdot (\!|f|\!) = (\!|g|\!) \Leftarrow h \text{ strict} \wedge \; h \cdot f = g \cdot \mathsf{F}h \tag{3.5}$$

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.

$$\begin{aligned}
& \mathit{cata}^{[]} :: \forall\, \alpha\, \rho . \rho \to (\alpha \to \rho \to \rho) \to [\alpha] \to \rho \\
& \quad \{ \text{ parametricity } \} \\
& (\mathit{cata}^{[]},\ \mathit{cata}^{[]}) \in \forall\, \mathcal{A}\, \mathcal{X} . \mathcal{X} \to (\mathcal{A} \to \mathcal{X} \to \mathcal{X}) \to [\mathcal{A}] \to \mathcal{X} \\
& \quad \{ \forall \text{ on relations twice } \} \\
= \; & \forall\, \mathcal{A} : A \Leftrightarrow A',\ \mathcal{B} : B \Leftrightarrow B'. \\
& (\mathit{cata}^{[]}_{AB},\ \mathit{cata}^{[]}_{A'B'}) \in \mathcal{B} \to (\mathcal{A} \to \mathcal{B} \to \mathcal{B}) \to [\mathcal{A}] \to \mathcal{B} \\
& \quad \{ \to \text{ three times } \} \\
= \; & \forall\, \mathcal{A} : A \Leftrightarrow A',\ \mathcal{B} : B \Leftrightarrow B'. \\
& \forall\, (n,\ n') \in \mathcal{B},\ (c,\ c') \in (\mathcal{A} \to \mathcal{B} \to \mathcal{B}),\ (xs,\ xs') \in [\mathcal{A}]. \\
& (\mathit{cata}^{[]}_{AB}\ n\ c\ xs,\ \mathit{cata}^{[]}_{A'B'}\ n'\ c'\ xs') \in \mathcal{B} \\
& \quad \{ \forall (n, n') \in \mathcal{B}, (c, c') \in (\mathcal{A} \to \mathcal{B} \to \mathcal{B}), (xs, xs') \in [\mathcal{A}] \text{ gives three conditions } \}
\end{aligned}$$

Case 1.$\langle \forall (n, n') \in \mathcal{B} \rangle$

$$b\ n = n'$$

Case 2.$\langle \forall (c, c') \in (\mathcal{A} \to \mathcal{B} \to \mathcal{B}) \rangle$

if $a\ z = z' \wedge b\ zs = zs'$

then

$$b\ (c\ z\ zs) = c'\ (a\ z)\ (b\ zs)$$

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

$$\begin{aligned}
& \mathit{map}^{[]}\ a\ xs = xs' \\
& \quad \{ \text{ this gives } \} \\
= \; & \forall\, a : A \to A',\ b : B \to B'.
\end{aligned}$$

$$
\begin{array}{ll}
 & \textbf{if}\ \forall\, z\ :\ A,\ zs\ :\ B. b\ n\ =\ n' \wedge\ b\ (c\ z\ zs)\ =\ c'\ (a\ z)\ (b\ zs)\ \wedge\ map^{[]}\ a\ xs\ =\ xs' \\
 & \textbf{then} \\
 & \quad b\ (cata^{[]}_{AB}\ n\ c\ xs)\ =\ cata^{[]}_{A'B'}\ n'\ c'\ .\ map^{[]}\ a\ xs \\
 & \quad \{\ \text{or slightly rewritten, in point-free style}\ \} \\
= & \forall\, a : A \to A',\ b : B \to B'. \\
 & \textbf{if}\ \forall\, z\ :\ A,\ zs\ :\ B. b\ n\ =\ n' \wedge\ b\ (c\ z\ zs)\ =\ c'\ (a\ z)\ (b\ zs) \\
 & \textbf{then} \\
 & \quad b\ .\ cata^{[]}_{AB}\ n\ c\ =\ cata^{[]}_{A'B'}\ n'\ c'\ .\ map^{[]}\ a
\end{array}
\qquad (3.6)
$$

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 $\lambda\, z'\ zs'. b\ (c\ z\ zs)$, simplify and apply the substitutions $[z\ :=\ z',\ b\ zs\ :=\ zs']$, which are the premisses of Case 2.

To turn an arbitrary function into a catamorphism:

$$
\begin{array}{ll}
 & \quad \{\ \text{Take}\ n\ =\ [],\ c\ =\ (:),\ a\ =\ id\ \} \\
= & \forall\, b : B \to B'. b\ \mathit{strict} \\
 & \textbf{if}\ \forall\, z\ :\ A,\ zs\ :\ B. b\ []\ =\ n' \wedge\ b\ ((:)\ z\ zs)\ =\ c'\ z\ (b\ zs) \\
 & \textbf{then} \\
 & \quad b\ .\ cata^{[]}_{AB}\ []\ (:)\ =\ cata^{[]}_{A'B'}\ n'\ c' \\
 & \quad \{\ \text{Use the conditions as definition of}\ n'\ \text{and}\ c'\ \} \\
= & \forall\, b : B \to B'. b\ \mathit{strict} \\
 & b\ .\ cata^{[]}_{AB}\ []\ (:)\ =\ cata^{[]}_{A'B'}\ n'\ c' \\
 & \textbf{where} \\
 & \quad n'\ =\ b\ [] \\
 & \quad c'\ =\ \lambda\, z'\ zs'. b\ ((:)\ z\ zs)\ [z\ :=\ z',\ b\ zs\ :=\ zs'] \\
 & \quad \{\ cata^{[]}\ []\ (:)\ =\ id\ \} \\
= & \forall\, b : B \to B'. b\ \mathit{strict} \\
 & b\ =\ cata^{[]}_{A'B'}\ n'\ c' \qquad (3.7) \\
 & \textbf{where} \\
 & \quad n'\ =\ b\ [] \\
 & \quad c'\ =\ \lambda\, z'\ zs'. b\ ((:)\ z\ zs)\ [z\ :=\ z',\ b\ zs\ :=\ zs']
\end{array}
$$

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 $b\ ((:)\ z\ zs)$.

Lets see an example in detail! This time we turn the definition of *map* for lists to a catamorphism.

$$
\begin{array}{l}
\quad\quad \langle\text{definition of } map^{[]}\rangle \\
map^{[]} \ :: \forall\, \alpha\, \beta.[\alpha] \ \to\ (\alpha \ \to\ \beta) \ \to\ [\beta] \\
map^{[]} \ = \Lambda\, \alpha\, \beta.\lambda\, xs\, f.\, \mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\quad \to [] \, \beta \\
\qquad\qquad\qquad\qquad (:)\, x\, xs \ \to (:)\, \beta\, (f\ x)\, (map\ \alpha\, \beta\, xs\, f) \\
\quad\quad \langle\text{Equation 3.7}\rangle \\
map^{[]} \ = \Lambda\, \alpha\, \beta.\lambda\, xs\, f.cata\ \beta\ [\beta]\ n'\ c'\ xs\ f \\
\quad \mathbf{where} \\
\qquad n' \ = \ map^{[]}\ \alpha\, \beta\, ([]\ \alpha) \\
\qquad c' \ = \ \lambda\, z'\ zs'.map^{[]}\ \alpha\, \beta\, ((:)\ \alpha\ z\ zs) \\
\quad\quad \langle\text{simplifies to}\rangle \\
map^{[]} \ = \Lambda\, \alpha\, \beta.\lambda\, xs\, f.cata\ \beta\ [\beta]\ n'\ c'\ xs\ f \\
\quad \mathbf{where} \\
\qquad n' \ = \ \lambda\, f.[]\ \beta \\
\qquad c' \ = \ \lambda\, z'\ zs'.(:)\ \beta\, (f\ z)\, (map^{[]}\ \alpha\, \beta\, zs\, f) \\
\quad\quad \langle\text{apply the substitutions } [z := z', map^{[]}\ \alpha\ \beta\ zs := zs']\rangle \\
map^{[]} \ = \Lambda\, \alpha\, \beta.\lambda\, xs\, f.cata\ \beta\ [\beta]\ n'\ c'\ xs\ f \\
\quad \mathbf{where} \\
\qquad n' \ = \ \lambda\, f.[]\ \beta \\
\qquad c' \ = \ \lambda\, z'\ zs'.(:)\ \beta\, (f\ z')\, (zs'\ f)
\end{array}
$$

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

Theorem 3.2 (Acid Rain for catamorphism)

$$g : \forall A.(\mathsf{F}A \to A) \to B \to A \Rightarrow (\!|\varphi|\!)_F \cdot (g\ in_F) = g\ \varphi$$

Proof by parametricity:

$$
\begin{array}{l}
\quad\quad \{\ \text{wish}\ \} \\
= (\!|\ \varphi\ |\!)_F\ (build^F\ g) \ = \ g\, \varphi \\
\quad\quad \{\ \text{definition: } build^F\ g \ = \ g\ in_F\ \} \\
\quad (\!|\ \varphi\ |\!)_F\ (g\ in_F) \ = \ g\, \varphi
\end{array}
$$

$$
\begin{aligned}
& \quad \{ \text{ the free theorem for } g\text{'s type } \} \\
& f\ (g\ \psi) \ = \ g\ \varphi \ \Leftarrow \ f \cdot \psi = \varphi \cdot \mathsf{F}\, f \\
& \quad \{ \text{ take } f := (\!|\, \varphi \,|\!)_F,\ \psi := in_F \ \} \\
= \ & (\!|\, \varphi \,|\!)_F\ (g\ in_F) \ = \ g\ \varphi \ \Leftarrow \ (\!|\, \varphi \,|\!)_F \cdot in_F \ = \ \varphi \cdot \mathsf{F}\, (\!|\, \varphi \,|\!)_F \\
& \quad \{ \text{ premise trivially holds } \} \\
= \ & \mathit{True}
\end{aligned}
$$

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 $build^F\ g$ for $g\ in_F$ the Acid Rain theorem can be restated as follows (provided the left-hand side is well-typed):

$$(\!|\varphi|\!)_F \cdot (build^F\ g) = g\ \varphi \tag{3.8}$$

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]):

$$foldr\ k\ z\ (build\ g) = g\ k\ z \tag{3.9}$$

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.

$$
\begin{aligned}
& f \\
& \quad \{ \ \mathit{build} \text{ introduction splits } f \text{ into two } \} \\
= \ & build^F\ f' \\
f' = \ & \lambda\, \varphi . (\!|\, \varphi \,|\!)_F \cdot f
\end{aligned}
$$

$$
\begin{aligned}
&\quad \{ \text{ definition of } f' \ \} \\
&= \mathit{build}^F \ (\lambda\, \varphi . (\!| \varphi \,|\!)_F \ \cdot \ f) \\
&\quad \{ \text{ definition of } \mathit{build} \ \} \\
&= (\lambda\, \varphi . (\!| \ \varphi \ |\!)_F \ \cdot \ f) \ \mathit{in}_F \\
&\quad \{ \text{ beta reduction } \} \\
&= (\!| \ \mathit{in}_F \ |\!)_F \ \cdot \ f \\
&\quad \{ \ (\!| \ \mathit{in}_F \ |\!)_F \ = \ \mathit{id} \ \} \\
&= f
\end{aligned}
$$

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

$$\mathbf{data}\ List\ a\ =\ Nil\ |\ Cons\ a\ (List\ a)$$

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:

Nil	⟨The empty list⟩
$Cons\ 42\ Nil$	⟨The list containing one element: 42⟩
$Cons\ 42\ (Cons\ 69\ Nil)$	⟨The list containing two elements: 42 and 69⟩
...	⟨There are many more lists⟩

An important function which can naturally be associated with this type is called *cata* (from catamorphism). The defining property of *cata* is that when it is applied to a list it uses its arguments (*nil* and *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 $cata\ 0\ (+)$, where $(+)$ is the infix addition operator, to the empty list *Nil* results in 0, since the catamorphism replaces *Nil* with the first argument to the *cata*, which is 0. The result of applying the *cata* above to our second example:

$$\begin{aligned} & cata\ 0\ (+)\ (Cons\ 42\ Nil) \\ \rightarrow\ & (+)\ 42\ (cata\ 0\ (+)\ Nil) \\ \rightarrow\ & (+)\ 42\ 0 \\ \rightarrow\ & 42 \end{aligned}$$

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

$$\begin{aligned} cata\ n\ c\ Nil &= n \\ cata\ n\ c\ (Cons\ x\ xs) &= c\ x\ (cata\ n\ c\ xs) \end{aligned}$$

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

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

$$build\ g\ =\ g\ Nil\ Cons$$

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 *Nil* and *Cons*. For example,

$$\begin{array}{lcl} map & = & build\ (\lambda\, f\ xs\ n\ c.\ \mathbf{case}\ xs\ \mathbf{of} \\ & & \qquad\qquad\qquad \begin{array}{lcl} Nil & \rightarrow & n \\ Cons\ x\ xs & \rightarrow & c\ (f\ x)\ (map\ f\ xs\ n\ c)) \end{array} \end{array}$$

is a valid use of build, while

$$\begin{array}{lcl} map & = & build\ (\lambda\, f\ xs\ n\ c.\ \mathbf{case}\ xs\ \mathbf{of} \\ & & \qquad\qquad\qquad \begin{array}{lcl} Nil & \rightarrow & Nil \\ Cons\ x\ xs & \rightarrow & Cons\ (f\ x)\ (map\ f\ xs\ n\ c)) \end{array} \end{array}$$

is not, because *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 `cata-build` rule:

$$cata\ nil\ cons\ (build\ g)\ =\ g\ nil\ cons$$

The theorem says: if a list is built with *build* and consumed by a *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 `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 exists: the transformation which introduces a build is called *buildify*, the other one which introduces a catamorphism is called *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 *build* in it. The reason it is called buildify is that the transformation makes it explicit that the function *produces* its result in a certain way. Functions which can be transformed are often called *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:

$$\begin{aligned} repAnswer \;=\; & \lambda\, n.\, \textbf{case}\ n\ \textbf{of} \\ & \qquad\qquad 0 \rightarrow Nil \\ & \qquad\qquad \_ \rightarrow Cons\ 42\ (repAnswer\ (n-1)) \end{aligned}$$

One — wrong — way to do this transformation is to simply slap a *build* around the definition of *repAnswer*:

$$\begin{aligned} & \text{— The two new lambdas are needed because build's argument must be} \\ & \text{— a function which takes the two constructors as arguments} \\ repAnswer \;=\; & build\ (\lambda\ nil\ cons.\lambda\ n.\, \textbf{case}\ n\ \textbf{of} \\ & \qquad\qquad 0 \rightarrow Nil \\ & \qquad\qquad \_ \rightarrow Cons\ 42\ (repAnswer\ (n-1))) \end{aligned}$$

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

$$\begin{aligned} & \text{— First correct definition of the transformation} \\ repAnswer \;=\; & build\ (\lambda\ nil\ cons\ n.cata\ nil\ cons\ (\ \textbf{case}\ n\ \textbf{of} \\ & \qquad\qquad 0 \rightarrow Nil \\ & \qquad\qquad \_ \rightarrow Cons\ 42\ (repAnswer\ (n-1)))) \end{aligned}$$

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

$$\begin{aligned} repAnswer \;=\; & build\ (\lambda\ nil\ cons\ n.\, \textbf{case}\ n\ \textbf{of} \\ & \qquad\qquad 0 \rightarrow cata\ nil\ con\ Nil \\ & \qquad\qquad \_ \rightarrow cata\ nil\ cons\ (Cons\ 42\ (repAnswer\ (n-1)))) \end{aligned}$$

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

[1]Notice, that *Cons* is the constructor while *cons* is its abstraction. We use the same name, lowercased, to help the reader.

$$
\begin{aligned}
repAnswer \;=\; build\,(\lambda\, nil\ cons\ n.\ \mathbf{case}\ n\ \mathbf{of}\\
&\quad 0 \rightarrow nil\\
&\quad \_ \rightarrow cons\ 42\ (cata\ nil\ cons\ (repAnswer\ (n-1))))
\end{aligned}
$$

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:

$$
\begin{aligned}
repAnswer &= \lambda\, n.build\,(repAnswer'\ n)\\
repAnswer' &= \lambda\, n\ nil\ cons.cata\ nil\ cons\ (\ \mathbf{case}\ n\ \mathbf{of}\\
&\qquad 0 \rightarrow Nil\\
&\qquad \_ \rightarrow Cons\ 42\ (repAnswer\ (n-1)))
\end{aligned}
$$

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 *Nil* and *Cons*. This gives:

$$
\begin{aligned}
repAnswer &= \lambda\, n.build\,(repAnswer'\ n)\\
repAnswer' &= \lambda\, n\ nil\ cons.\ \mathbf{case}\ n\ \mathbf{of}\\
&\qquad 0 \rightarrow nil\\
&\qquad \_ \rightarrow cons\ 42\ (cata\ nil\ cons\ (repAnswer\ (n-1)))
\end{aligned}
$$

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.

$$
\begin{array}{lcl}
repAnswer' & = & \lambda\, n\; nil\; cons.\; \textbf{case}\; n\; \textbf{of} \\
& & \quad 0 \rightarrow nil \\
& & \quad \_ \rightarrow cons\; 42\; (cata\; nil \\
& & \qquad\qquad cons \\
& & \qquad\qquad ((\lambda\, n'.build\; (repAnswer'\; n'))\; (n-1)))
\end{array}
$$

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

$$
\begin{array}{lcl}
repAnswer & = & \lambda\, n.build\; (repAnswer'\; n) \\
repAnswer' & = & \lambda\, n\; nil\; cons.\; \textbf{case}\; n\; \textbf{of} \\
& & \quad 0 \rightarrow nil \\
& & \quad \_ \rightarrow cons\; 42\; (cata\; nil\; cons\; (build\; (repAnswer'\; (n-1))))
\end{array}
$$

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

$$
\begin{array}{lcl}
repAnswer & = & \lambda\, n.build\; (repAnswer'\; n) \\
repAnswer' & = & \lambda\, n\; nil\; cons.\; \textbf{case}\; n\; \textbf{of} \\
& & \quad 0 \rightarrow nil \\
& & \quad \_ \rightarrow cons\; 42\; (repAnswer'\; (n-1)\; nil\; cons)
\end{array}
$$

We managed to transform a function into another one with a *build* in it without paying any penalty for the extra traversal by a remaining *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 `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 *consumes* its argument in a certain way. Accordingly, successfully transformed functions are often called *good consumers*, and to denote this property we stick a *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 *sum* function:

$$
\begin{aligned}
sum \;=\; \lambda\, xs.\, &\mathbf{case}\ xs\ \mathbf{of} \\
&\quad Nil \qquad\quad \rightarrow\ 0 \\
&\quad Cons\ a\ as\ \rightarrow\ a\ +\ sum\ as
\end{aligned}
$$

The game plan is to somehow change this definition to have a *cata* in it. There are several places one could put a *cata* into the right-hand side of *sum*, but if we recall what a *cata* does we might just find the right place. We already discussed, that a *cata* is a special form of recursion (*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 *Cons* cells are replaced by the second argument, and *Nil* is replaced by the first. In other words, the first argument to *cata* must be equivalent what *sum* does if it finds a *Nil* constructor and the second argument must do the same what *sum* does when it finds a *Cons*. But how do we find out what *sum* does in each case? We partially evaluate it!

$$sum \;=\; \lambda\, xs.cata\ sum_{Nil}\ sum_{Cons}\ xs$$

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

$$
\begin{aligned}
sum_{Nil} &= sum\ Nil \\
sum_{Cons} &= sum\ (Cons\ t\ ts)
\end{aligned}
$$

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

$$
\begin{aligned}
sum_{Nil} &= (\lambda\, xs.\, \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad Nil \qquad\quad \rightarrow\ 0 \\
&\qquad\qquad Cons\ a\ as\ \rightarrow\ a\ +\ sum\ as)\ Nil \\
sum_{Cons} &= (\lambda\, xs.\, \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad Nil \qquad\quad \rightarrow\ 0 \\
&\qquad\qquad Cons\ a\ as\ \rightarrow\ a\ +\ sum\ as)\ (Cons\ t\ ts)
\end{aligned}
$$

Beta-reduction both on sum_{Nil} and sum_{Cons} gives:

$$\begin{aligned}
sum_{Nil} &= \textbf{case}\ Nil\ \textbf{of} \\
&\qquad Nil \qquad\quad \rightarrow\ 0 \\
&\qquad Cons\ a\ as\ \rightarrow\ a\ +\ sum\ as \\
sum_{Cons} &= \textbf{case}\ Cons\ t\ ts\ \textbf{of} \\
&\qquad Nil \qquad\quad \rightarrow\ 0 \\
&\qquad Cons\ a\ as\ \rightarrow\ a\ +\ sum\ as
\end{aligned}$$

This exposes further opportunities for simplification because the scrutinee of the case is known i.e. in the first case of sum_{Nil} the scrutinee is *Nil* therefore it cannot possibly be a *Cons* so we simplify, get rid of the entire **case** expression and get:

$$\begin{aligned}
sum_{Nil} &= 0 \\
sum_{Cons} &= t\ +\ sum\ ts
\end{aligned}$$

This is almost perfect. The only problem left is the call to *sum* in the right-hand side of sum_{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 *sum* in our case) with calls to the appropriate catamorphism. So we need to replace *sum ts* with something which does not mention the function *sum*. But unfortunately, there does not seem to be anything to replace *sum ts* with.

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

$$\begin{aligned}
sum_{Nil} &= 0 \\
sum_{Cons} &= \lambda\, z\ zs.\ t\ +\ sum\ ts
\end{aligned}$$

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 sum_{Cons} and t and ts are free. We can solve the two problems (there was nothing to replace *sum ts* with and t and ts being free in the body of sum_{Cons}) if we replace t with z and *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. *List Int* in our case) then replace the application of the original function to this argument (*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. *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 *sum* is transformed, the expression *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.

Lets see what happens if we replace variables according to the rules above:

$$\begin{aligned}
sum &= \lambda\, xs.cata\ sum_{Nil}\ sum_{Cons}\ xs \\
sum_{Nil} &= 0 \\
sum_{Cons} &= \lambda\, z\ zs.\ z\ +\ zs
\end{aligned}$$

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

— We renamed *xs* in the body of *cata* to avoid a name clash

$$\begin{aligned}
sum &= \lambda\, xs.(\lambda\, nil\ cons\ xs'.\ \textbf{case}\ xs'\ \textbf{of} \\
&\qquad\qquad\qquad Nil \to nil \\
&\qquad\qquad\qquad Cons\ a\ as \to cons\ a\ (cata\ nil\ cons\ as)) \\
&\qquad sum_{Nil}\ sum_{Cons}\ xs \\
sum_{Nil} &= 0 \\
sum_{Cons} &= \lambda\, z\ zs.\ z\ +\ zs
\end{aligned}$$

We can now inline sum_{Nil} and sum_{Cons} into the body of *sum* and do three beta-reductions. This gives:

$$\begin{aligned}
sum &= \lambda\, xs.\ \textbf{case}\ xs\ \textbf{of} \\
&\qquad Nil \to 0 \\
&\qquad Cons\ a\ as \to a\ +\ (cata\ sum_{Nil}\ sum_{Cons}\ as)
\end{aligned}$$

$$\begin{aligned} sum_{Nil} &= 0 \\ sum_{Cons} &= \lambda\, z\; zs.\; z\; +\; zs \end{aligned}$$

Earlier on we made the claim that *sum* is equivalent to $cata\ sum_{Nil}\ sum_{Cons}$. If we assume that this is in fact the case, we can replace *sum* by $cata\ sum_{Nil}\ sum_{Cons}$ or vice versa. We note that $cata\ sum_{Nil}\ sum_{Cons}$ does occur in the body and replacing it by *sum* gives:

$$\begin{aligned} sum &= \lambda\, xs.\; \mathbf{case}\; xs\; \mathbf{of} \\ &\qquad\qquad\quad Nil \qquad\quad \rightarrow\; 0 \\ &\qquad\qquad\quad Cons\; a\; as\; \rightarrow\; a\; +\; sum\; as \\ sum_{Nil} &= 0 \\ sum_{Cons} &= \lambda\, z\; zs.\; z\; +\; zs \end{aligned}$$

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 *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]):

$$\begin{aligned} &\mathbf{data}\; cx\; \Rightarrow\; T_1\; tv_1\; \ldots\; tv_m\; = K_{11}\; ty_{11}\; \ldots\; ty_{1k} \mid \ldots \mid K_{1n}\; ty_{n1}\; \ldots\; ty_{nk} \\ &\vdots \\ &\mathbf{data}\; cx\; \Rightarrow\; T_i\; tv_1\; \ldots\; tv_m\; = K_{i1}\; ty_{11}\; \ldots\; ty_{1k} \mid \ldots \mid K_{in}\; ty_{n1}\; \ldots\; ty_{nk} \end{aligned} \tag{4.1}$$

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:

$$K_{ij}\; ::\; \forall\; tv_1\; \ldots\; tv_m.\; ty_{i1}\; \rightarrow\; \cdots\; \rightarrow\; ty_{ik_i}\; \rightarrow\; (T_i\; tv_1\; \ldots\; tv_m)$$

Polynomial datatypes are properly defined in Definition 4.1, here we give a purely syntactic definition.

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* ($\rightarrow$) *nor quantifiers* ($\forall$) *appear in* $ty_{i1}, \ldots, ty_{ik_i}$ *for all* i, k.

An example of non-polynomial datatype is:

$$\textbf{data}\ T\,\alpha\,\beta\ =\ T1\,(\alpha\ \rightarrow\ \beta)\mid\ \ldots$$

because of the function space constructor in $T1\,(\alpha\ \rightarrow\ \beta)$.

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:

$$\textbf{data}\ \mathit{Twist}\,\alpha\,\beta\ =\ \mathit{Twist}\,\alpha\,(\mathit{Twist}\,\beta\,\alpha)\mid\ \ldots$$

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

$$\textbf{data}\ \mathit{Nest}\,\alpha\ =\ N1\,(\mathit{Nest}\,[\alpha])\mid\ \ldots$$

is also non-regular, because in the recursive use of the datatype being defined (the first argument $\mathit{Nest}\,[\alpha]$ to the constructor $N1$) Nest's argument is $[\alpha]$, while in the head of the definition is α. 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 `cata-build` rule with the help of inlining wrappers. Neither the wrappers nor the `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 *already transformed* functions to allow for more fusion.

4.3.1 The preprocessing stage

The *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

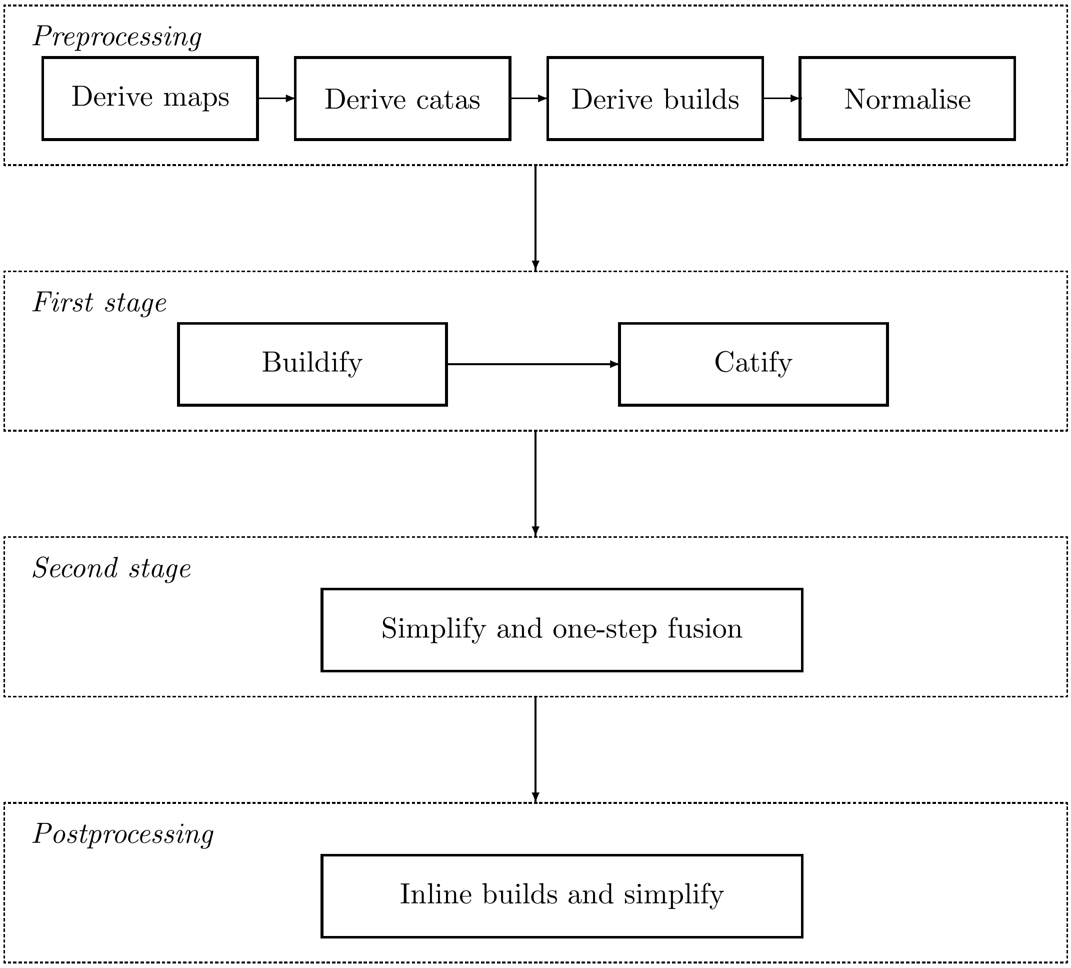

Figure 4.1 Overview of the fusion transformation

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 $map^{\square} :: \forall\,\alpha\beta.[\alpha] \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta]$ as opposed to the usual $map^{\square} :: \forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\beta])$.

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.

4.3.3 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:

$$\begin{array}{l}\langle\text{Pseudo code}\rangle\\ f \;=\; \lambda\,\bar{v}.e\\ \Longrightarrow\\ f \;=\; \lambda\,\bar{v}.\mathit{build}\;(\lambda\;c_1\;\ldots\;c_n.\;e)\end{array}$$

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 $\mathit{cata}^T\;c_1\;\ldots\;c_n$ 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:

$$\begin{array}{l}\langle\text{Pseudo code}\rangle\\ f \;=\; \lambda\,\bar{v}.e\\ \Longrightarrow\\ f \;=\; \lambda\,\bar{v}.\mathit{build}\;(\lambda\;c_1\;\ldots\;c_n.\mathit{cata}\;c_1\;\ldots\;c_n\;e)\end{array}$$

For example, in the case of lists, build has type $(\forall\alpha\;\rho.\rho \to\;(\alpha\;\to\;\rho\;\to\;\rho)\;\to\;\rho)\;\to\;[\alpha]$. By using the parametricity theorem [Rey83, Wad89], one can show that if g has the given type, it must work for *any* ρ. The intuitive explanation of this result is that g is provided with no other operations of type ρ 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 `cata-build` rule is that a catamorphism is applied to an appropriately typed function. For example, in the case of lists:

$$cata^{[\alpha]}\ \rho\ n\ c\ g \Rightarrow g\ \rho\ n\ c,\ \textbf{if}\ g :: \forall\beta.\beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \tag{4.2}$$

Of course, if we dispense build it would be somewhat meaningless to call Equation 4.2 the `cata-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[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 *good consumer:* it consumes its argument in a disciplined manner i.e. with a fixed pattern of recursion.

[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] 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

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 [Fok92b] — are well known in functional programming. The usual reading of the type of map for lists, $map^{[]} :: \forall\alpha\beta.(\alpha \to \beta) \to ([\alpha] \to [\beta])$ is that 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 α.

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

$$\begin{aligned} map^T = \;& \Lambda\, \bar\alpha\, \bar\beta. \lambda\, f_1 \ldots f_m . \lambda\, t. \\ & \quad \mathbf{case}\; t\; \mathbf{of} \\ & \quad\quad \{ T_i\; \bar v \;\to\; T_i\; \bar\beta\, (M^T\, f_1 \ldots f_m\; (map^T\; \bar\alpha\, \bar\beta\, f_1 \ldots f_m)\; \bar v) \}_{i=1}^{n} \end{aligned} \tag{4.3}$$

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.

$$\begin{aligned} & M^T\, f_1 \ldots f_m\; g\; v \;=\; \mathcal{M}^T\, f_1 \ldots f_m\; g\; (\texttt{typeOf}\; v)\; v \\ & \quad \mathbf{where} \\ & \quad\quad \mathcal{M}^T\, f_1 \ldots f_m\; g\; [\![\, \mathit{primitive} \,]\!] = \lambda\, x.x \\ & \quad\quad \mathcal{M}^T\, f_1 \ldots f_m\; g\; [\![\alpha]\!] = \lambda\, x. \{ f_i\; x \mid \texttt{sourceTypeOf}\, f_i \;=\; \alpha \;\wedge\; i \;\in\; \{1 \ldots\, n\} \} \\ & \quad\quad \mathcal{M}^T\, f_1 \ldots f_m\; g\; [\![T\; \bar\alpha]\!] = \lambda\, x.g\; x \\ & \quad\quad \mathcal{M}^T\, f_1 \ldots f_m\; g\; [\![K\; \bar\tau]\!] = \lambda\, x.map^K\; (\texttt{tyVarsOf}(\texttt{sourceTypeOf}\; g)) \\ & \quad\quad\quad (\texttt{tyVarsOf}(\texttt{targetTypeOf}\; g)) \\ & \quad\quad\quad (\mathcal{M}^T\, f_1 \ldots f_m\; g\; [\![\, \bar\tau \,]\!]) \\ & \quad\quad\quad x \end{aligned} \tag{4.4}$$

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

Lets see what M does! The first case deals with primitive types, for example the built-in 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 approriate f which rewrites the given type variable. Two questions arise: can we be sure that we find *at least one* f such that `sourceTypeOf` f is equal to the given type variable and can we be sure that we find *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:

$$\begin{array}{l}
\langle \text{Equation 4.3} \rangle \\
\mathit{map}^{[]} \;=\; \Lambda\,\alpha\,\beta.\lambda\,f\;t.\,\textbf{case}\;t\;\textbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow [\,]\,\beta \\
\qquad\qquad\qquad\qquad (:)\;a\;as \rightarrow (:)\,\beta\,(M^{[]}\,f\,(\mathit{map}[]\,\alpha\,\beta\,f)\;a) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (M^{[]}\,f\,(\mathit{map}[]\,\alpha\,\beta\,f)\;as) \\
\langle \text{Equation 4.4} \rangle \\
\mathit{map}^{[]} \;=\; \Lambda\,\alpha\,\beta.\lambda\,f\;t.\,\textbf{case}\;t\;\textbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow [\,]\,\beta \\
\qquad\qquad\qquad\qquad (:)\;a\;as \rightarrow (:)\,\beta\,(\,\mathcal{M}^{[]}\,f\,(\mathit{map}[]\,\alpha\,\beta\,f)\;\alpha\;as) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\,\mathcal{M}^{[]}\,f\,(\mathit{map}[]\,\alpha\,\beta\,f)\;[\,]\;as) \\
\langle \text{second and third clause of } \mathcal{M} \rangle \\
\mathit{map}^{[]} \;=\; \Lambda\,\alpha\,\beta.\lambda\,f\;t.\,\textbf{case}\;t\;\textbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow [\,]\,\beta \\
\qquad\qquad\qquad\qquad (:)\;a\;as \rightarrow (:)\,\beta\,(\lambda\,x.f\;x)\;a \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\lambda\,x.\mathit{map}^{[]}\,\alpha\,\beta\,f\;x)\;as \\
\langle \text{beta reductions} \rangle \\
\mathit{map}^{[]} \;=\; \Lambda\,\alpha\,\beta.\lambda\,f\;t.\,\textbf{case}\;t\;\textbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow [\,]\,\beta \\
\qquad\qquad\qquad\qquad (:)\;a\;as \rightarrow (:)\,\beta\,(f\;a)\,(\mathit{map}^{[]}\,\alpha\,\beta\,f\;as)
\end{array}$$

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:

$$\begin{array}{l}
\mathit{cata}^{T} \;= \Lambda\,\bar{\alpha}\,\rho.\lambda\bar{c}.\lambda\,t. \\
\qquad\quad \textbf{case}\;t\;\textbf{of} \\
\qquad\qquad \{T_i\;\bar{v}\;\rightarrow\;c_i\,(E^{T}\,(\mathit{cata}^{T}\;\bar{\alpha}\,\rho\,\bar{c})\;\bar{v})\}_{i=1}^{n}
\end{array} \tag{4.5}$$

In the equation above, n is the number of constructors the datatype $T\bar{\alpha}$ has, ρ 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 ρ. If $\texttt{monoConstrs}(T\,\bar{\alpha})$ denotes the list of constructors (with their forall(s) stripped off), the substitution $[\rho/T\,\bar{\alpha}]$ — substitute ρ for $T\,\bar{\alpha}$ — will give the right types.

For example, for lists

$$\textbf{data}\,[]\,\alpha \;=\; []\mid\alpha\;:\;[\alpha]$$

$\texttt{monoConstrs}([\alpha])$ 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 ρ and $\alpha \to \rho \to \rho$. Equipped with this notation, it is easy to give a type to $cata^T$.

$$cata^T \;::\; \forall\bar{\alpha}.\forall\rho.\texttt{monoConstrs}(T\bar{\alpha}) \to\; T\bar{\alpha} \;\to\; \rho$$

In the running example of lists we get

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

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

$$
\begin{array}{l}
E^T\;g\;v \;=\; \mathcal{E}^T\;g\;(\texttt{typeOf}\;v)\;v \\
\quad\textbf{where} \\
\qquad \begin{array}{lcl}
\mathcal{E}^T\;g\;[\![\,\textit{primitive type}]\!] & = & \lambda\,x.x \\
\mathcal{E}^T\;g\;[\![\alpha]\!] & = & \lambda\,x.x \\
\mathcal{E}^T\;g\;[\![T\;\bar{\alpha}]\!] & = & \lambda\,x.g\;x \\
\mathcal{E}^T\;g\;[\![K\;\bar{\tau}]\!] & = & \lambda\,x.map^K\;(\texttt{sourceTypeOf}\;g) \\
 & & (\texttt{targetTypeOf}\;g) \\
 & & (\mathcal{E}^T\;g\;[\![\,\bar{\tau}\,]\!]) \\
 & & x
\end{array}
\end{array}
\tag{4.6}
$$

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

$\langle$Equation 4.5$\rangle$

$$
\begin{array}{l}
cata^{[]} \;=\; \Lambda\, \alpha\, \rho. \lambda\, nil\; cons. \lambda\, t.\, \mathbf{case}\; t\; \mathbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow nil \\
\qquad\qquad\qquad\qquad (:)\; y\; ys \rightarrow cons\; (E^{[]}\; (cata^{[]}\; \alpha\, \rho\; nil\; cons)\; y) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (E^{[]}\; (cata^{[]}\; \alpha\, \rho\; nil\; cons)\; ys) \\
\langle \text{Equation 4.6} \rangle \\
cata^{[]} \;=\; \Lambda\, \alpha\, \rho. \lambda\, nil\; cons. \lambda\, t.\, \mathbf{case}\; t\; \mathbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow nil \\
\qquad\qquad\qquad\qquad (:)\; y\; ys \rightarrow cons\; (\, \mathcal{E}^{[]}\; (cata^{[]}\; \alpha\, \rho\; nil\; cons)\; \alpha\; y) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\, \mathcal{E}^{[]}\; (cata^{[]}\; \alpha\, \rho\; nil\; cons)\; []\; ys) \\
\langle \text{second and third clause of } \mathcal{E} \rangle \\
cata^{[]} \;=\; \Lambda\, \alpha\, \rho. \lambda\, nil\; cons. \lambda\, t.\, \mathbf{case}\; t\; \mathbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow nil \\
\qquad\qquad\qquad\qquad (:)\; y\; ys \rightarrow cons\; (\lambda\, x.x)\; y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda\, x.cata^{[]}\; \alpha\, \rho\; nil\; cons\; x)\; ys \\
\langle \text{beta reductions} \rangle \\
cata^{[]} \;=\; \Lambda\, \alpha\, \rho. \lambda\, nil\; cons. \lambda\, t.\, \mathbf{case}\; t\; \mathbf{of} \\
\qquad\qquad\qquad\qquad [] \qquad\;\; \rightarrow nil \\
\qquad\qquad\qquad\qquad (:)\; y\; ys \rightarrow cons\; y\; (cata^{[]}\; \alpha\, \rho\; nil\; cons\; ys)
\end{array}
$$

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 $\mathcal{E}$, is the derivation of the cata for Rose trees.

$$
\begin{array}{l}
\mathbf{data}\; Rose\; \alpha \;=\; Fork\; \alpha\; [Rose\; \alpha] \\
\langle \text{definition} \rangle \\
cata^{Rose} \;::\; \forall \alpha. \forall \rho. (\alpha \rightarrow [\rho] \rightarrow \rho) \rightarrow\; Rose\; \alpha \rightarrow \rho \\
cata^{Rose} \;=\; \Lambda \alpha. \Lambda \rho. \lambda\, fork. \lambda\, t. \\
\qquad\qquad \mathbf{case}\; t\; \mathbf{of} \\
\qquad\qquad\quad Fork\; (a :: \alpha) \\
\qquad\qquad\qquad\quad (lt :: [Rose\; \alpha]) \;\rightarrow\; fork\; (E^{Rose}\; (cata^{Rose}\; \alpha\, \rho\, fork)\; [a,\; lt]) \\
\langle \text{definition of E twice} \rangle \\
cata^{Rose} \;=\; \Lambda \alpha. \Lambda \rho. \lambda\, fork. \lambda\, t. \\
\qquad\qquad \mathbf{case}\; t\; \mathbf{of} \\
\qquad\qquad\quad Fork\; (a :: \alpha) \\
\qquad\qquad\qquad\quad (lt :: [Rose\; \alpha]) \;\rightarrow\; fork\; (\, \mathcal{E}^{Rose}\; (cata^{Rose}\; \alpha\, \rho\, fork)\; a) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\, \mathcal{E}^{Rose}\; (cata^{Rose}\; \alpha\, \rho\, fork)\; lt) \\
\langle \mathcal{E} \text{ applied to a type variable} \rangle
\end{array}
$$

$$
\begin{array}{lll}
cata^{Rose} & = & \Lambda\alpha.\Lambda\rho.\lambda\, fork.\lambda\, t. \\
& & \quad \mathbf{case}\ t\ \mathbf{of} \\
& & \qquad Fork\ (a :: \alpha) \\
& & \qquad\qquad (lt :: [Rose\ \alpha]) \ \rightarrow\ fork\ a \\
& & \qquad\qquad\qquad\qquad\qquad\qquad (\ \mathcal{E}^{Rose}\ (cata^{Rose}\ \alpha\, \rho\, fork)\ lt)
\end{array}
$$

$\langle\mathcal{E}$ applied to a type constructor different from the one being defined$\rangle$

$$
\begin{array}{lll}
cata^{Rose} & = & \Lambda\alpha.\Lambda\rho.\lambda\, fork.\lambda\, t. \\
& & \quad \mathbf{case}\ t\ \mathbf{of} \\
& & \qquad Fork\ (a :: \alpha) \\
& & \qquad\qquad (lt :: [Rose\ \alpha]) \ \rightarrow\ fork\ a \\
& & \qquad\qquad\qquad\qquad\qquad\qquad ((\lambda\, x.map^{[]}\ (Rose\ \alpha) \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rho \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (cata^{Rose}\ \alpha\, \rho\, fork) \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x)\ lt)
\end{array}
$$

$\langle\beta$-reduction$\rangle$

$$
\begin{array}{lll}
cata^{Rose} & = & \Lambda\alpha.\Lambda\rho.\lambda\, fork.\lambda\, t. \\
& & \quad \mathbf{case}\ t\ \mathbf{of} \\
& & \qquad Fork\ (a :: \alpha) \\
& & \qquad\qquad (lt :: [Rose\ \alpha]) \ \rightarrow\ fork\ a \\
& & \qquad\qquad\qquad\qquad\qquad\qquad (map^{[]}\ (Rose\ \alpha) \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \rho \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad (cata^{Rose}\ \alpha\, \rho\, fork) \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\qquad lt)
\end{array}
$$

Notice, $map^{[]}$ in the definition! This is a call to the familiar *map* function for lists. We have already shown how to derive the *map* function for arbitrary datatypes in Section 4.5.1. It is easy to verify that $cata^{Rose}$ is well-typed: $map^{[]}$ takes two type arguments $Rose\ \alpha$ and ρ, and a function from $Rose\ \alpha$ to ρ. $cata^{Rose}\ \alpha\, \rho\, fork$ does indeed have that type. $map^{[]}\ (Rose\ \alpha)\ \rho\ (cata^{Rose}\ \alpha\, \rho\, fork)\ lt$ has type $[Rose\ \alpha] \rightarrow [\rho]$ and *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 `cata-build` rule is proved correct on Page 24.

$$
\begin{array}{ll}
\langle \text{cata of case rule} \rangle & \\
\mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c}\ (\ \mathbf{case}\ \mathit{Expr}\ \mathbf{of}\ \overline{\{C\ \bar{v}\ \rightarrow\ e\}}) & \rightarrow\ \mathbf{case}\ \mathit{Expr}\ \mathbf{of}\ \overline{\{C\ \bar{v}\ \rightarrow\ \mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c}\ e\}} \\
\langle \text{cata of known constructor rule} \rangle & \\
\mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c}\ (C_i\ v_1 \ldots\ v_n) & \rightarrow\ c_i\ (E^T\ (\mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c})\ v_1) \\
 & \qquad \vdots \\
 & \qquad (E^T\ (\mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c})\ v_n) \\
\langle \text{cata-build rule} \rangle & \\
\mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c}\ (\mathit{build}^T\ \rho\, f) & \rightarrow\ f\ \rho\ \bar{c} \\
\langle \text{cata-of-error rule} \rangle & \\
\mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c}\ \mathit{error} & \rightarrow\ \mathit{error}
\end{array}
$$

Figure 4.2 Rules for the interaction of catamorphisms and Core

The local transformations (Table A.1) are still in effect. The notation *lhs* $\rightarrow$ *rhs* has its standard meaning: *lhs* reduces to *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 *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

$$
\begin{array}{ll}
f & :: \forall \bar{\alpha}.\bar{\sigma} \rightarrow\ T\bar{\tau} \\
f & = \Lambda \bar{\alpha}.\lambda \bar{v}.e \\
\Longrightarrow & \\
f & :: \forall \bar{\alpha}.\bar{\sigma} \rightarrow\ T\bar{\tau} \\
f & = \Lambda \bar{\alpha}.\lambda \bar{v}.\mathit{build}^T\ (f'\ \bar{\alpha}\ \bar{v}) \\
f' & :: \forall \bar{\alpha}.\bar{\sigma} \rightarrow (\forall \rho.\, \texttt{monoConstrs}(T\bar{\tau})[\rho/T\bar{\tau}] \rightarrow \rho) \\
f' & = \Lambda \bar{\alpha}.\lambda \bar{v}.\Lambda \rho.\lambda \bar{c}.\ \mathit{cata}^T\ \bar{\tau}\ \rho\ \bar{c}\ e
\end{array}
$$

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 ρ 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 $cata^T$ 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.

$$
\begin{aligned}
&\mathit{downTo} \;::\; \mathit{Int} \rightarrow [\mathit{Int}] \\
&\mathit{downTo} \;=\; \lambda\, n.\, \mathbf{case}\; n > 0\; \mathbf{of} \\
&\qquad\qquad\qquad\quad \mathit{True} \;\rightarrow\; (:)\; \mathit{Int}\; n\; (\mathit{downTo}\; (n-1)) \\
&\qquad\qquad\qquad\quad \mathit{False} \;\rightarrow\; []\; \mathit{Int}
\end{aligned}
$$

We rewrite this binding according to Step 1 and get:

$$
\begin{aligned}
&\mathit{downTo} \;::\; \mathit{Int} \rightarrow [\mathit{Int}] \\
&\mathit{downTo} \;=\; \lambda\, n.\mathit{build}^{[]}\; (\mathit{downTo}'\; n) \\
&\mathit{downTo}' \;::\; \mathit{Int} \rightarrow (\forall \rho.\rho \rightarrow (\mathit{Int} \rightarrow \rho \rightarrow \rho) \rightarrow \rho) \\
&\mathit{downTo}' \;=\; \lambda\, n.\Lambda\rho.\lambda\; \mathit{nil}.\lambda\; \mathit{cons}.
\end{aligned}
$$

$$\begin{array}{l} \mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons}\ (\ \mathbf{case}\ n > 0\ \mathbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{True} \ \rightarrow\ (:)\ \mathit{Int}\ n\ (\mathit{downTo}\ (n-1)) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{False} \ \rightarrow\ []\ \mathit{Int}) \end{array}$$

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 *downTo* apart from getting inlined, therefore we omit it. We also omit the type of downTo' since it does not change.

$$\begin{array}{l} \langle \text{cata of case} \rangle \\ \mathit{downTo}' \ = \ \lambda\, n.\Lambda\, \rho.\lambda\, \mathit{nil}.\lambda\, \mathit{cons}. \\ \qquad\qquad \mathbf{case}\ n > 0\ \mathbf{of} \\ \qquad\qquad\quad \mathit{True} \ \rightarrow\ \mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons}\ ((:)\ \mathit{Int}\ n\ (\mathit{downTo}\ (n-1))) \\ \qquad\qquad\quad \mathit{False} \ \rightarrow\ \mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons}\ ([]\ \mathit{Int}) \\ \langle \text{case of known constructor twice} \rangle \\ \mathit{downTo}' \ = \ \lambda\, n.\Lambda\, \rho.\lambda\, \mathit{nil}.\lambda\, \mathit{cons}. \\ \qquad\qquad \mathbf{case}\ n > 0\ \mathbf{of} \\ \qquad\qquad\quad \mathit{True} \ \rightarrow\ \mathit{cons}\ (E^{[]}\ (\mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons})\ n) \\ \qquad\qquad\qquad\qquad\quad (E^{[]}\ (\mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons})\ (\mathit{downTo}\ (n-1))) \\ \qquad\qquad\quad \mathit{False} \ \rightarrow\ \mathit{nil} \\ \langle \text{definition of } E \text{ twice} \rangle \\ \mathit{downTo}' \ = \ \lambda\, n.\Lambda\, \rho.\lambda\, \mathit{nil}.\lambda\, \mathit{cons}. \\ \qquad\qquad \mathbf{case}\ n > 0\ \mathbf{of} \\ \qquad\qquad\quad \mathit{True} \ \rightarrow\ \mathit{cons}\ (\ \mathcal{E}^{[]}\ (\mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons})\ \mathit{Int}\ n) \\ \qquad\qquad\qquad\qquad\quad (\ \mathcal{E}^{[]}\ (\mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons})\ [\mathit{Int}]\ (\mathit{downTo}\ (n-1))) \\ \qquad\qquad\quad \mathit{False} \ \rightarrow\ \mathit{nil} \\ \langle \mathcal{E} \text{ applied to a primitive type and } \mathcal{E} \text{ applied to the recursive use of } [\alpha] \rangle \\ \mathit{downTo}' \ = \ \lambda\, n.\Lambda\, \rho.\lambda\, \mathit{nil}.\lambda\, \mathit{cons}. \\ \qquad\qquad \mathbf{case}\ n > 0\ \mathbf{of} \\ \qquad\qquad\quad \mathit{True} \ \rightarrow\ \mathit{cons}\ ((\lambda\, x.x)\ n) \\ \qquad\qquad\qquad\qquad\quad ((\lambda\, x.\mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons}\ x)\ (\mathit{downTo}\ (n-1))) \\ \qquad\qquad\quad \mathit{False} \ \rightarrow\ \mathit{nil} \\ \langle \beta \text{ reductions} \rangle \\ \mathit{downTo}' \ = \ \lambda\, n.\Lambda\, \rho.\lambda\, \mathit{nil}.\lambda\, \mathit{cons}. \\ \qquad\qquad \mathbf{case}\ n > 0\ \mathbf{of} \\ \qquad\qquad\quad \mathit{True} \ \rightarrow\ \mathit{cons}\ n\ (\mathit{cata}^{[]}\ \mathit{Int}\ \rho\ \mathit{nil}\ \mathit{cons}\ (\mathit{downTo}\ (n-1))) \\ \qquad\qquad\quad \mathit{False} \ \rightarrow\ \mathit{nil} \\ \langle \mathit{downTo} \text{ gets inlined} \rangle \end{array}$$

$$
\begin{array}{l}
downTo' \;=\; \lambda\, n.\Lambda\, \rho.\lambda\, nil.\lambda\, cons.\\
\qquad\qquad \textbf{case}\; n > 0\; \textbf{of}\\
\qquad\qquad\quad True \;\rightarrow\; cons\; n\\
\qquad\qquad\qquad\qquad\qquad (cata^{[]}\; Int\; \rho\; nil\; cons\\
\qquad\qquad\qquad\qquad\qquad\qquad ((\lambda\, n'.build^{[]}\; (downTo'\; n'))\; (n-1)))\\
\qquad\qquad\quad False \;\rightarrow\; nil
\end{array}
$$

$\langle\beta$ reduction$\rangle$

$$
\begin{array}{l}
downTo' \;=\; \lambda\, n.\Lambda\, \rho.\lambda\, nil.\lambda\, cons.\\
\qquad\qquad \textbf{case}\; n > 0\; \textbf{of}\\
\qquad\qquad\quad True \;\rightarrow\; cons\; n\; (cata^{[]}\; Int\; \rho\; nil\; cons\; (build^{[]}\; (downTo'\; (n-1))))\\
\qquad\qquad\quad False \;\rightarrow\; nil
\end{array}
$$

⟨`cata-build` rule⟩

$$
\begin{array}{l}
downTo' \;=\; \lambda\, n.\Lambda\, \rho.\lambda\, nil.\lambda\, cons.\\
\qquad\qquad \textbf{case}\; n > 0\; \textbf{of}\\
\qquad\qquad\quad True \;\rightarrow\; cons\; n\; (downTo'\; (n-1)\; \rho\; nil\; cons)\\
\qquad\qquad\quad False \;\rightarrow\; nil
\end{array}
$$

⟨The Core Simplifier finished⟩

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

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 *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.

$$\textbf{data}\; Maybe\; \alpha \;=\; Nothing \mid Just\; \alpha$$

$$
\begin{array}{l}
map^{Maybe} :: \forall\alpha\beta.Maybe\; \alpha \rightarrow\; (\alpha \rightarrow \beta) \rightarrow\; Maybe\; \beta\\
map^{Maybe} = \Lambda\; \alpha\; \beta.\lambda\; m\; f.\; \textbf{case}\; m\; \textbf{of}\\
\qquad\qquad\qquad\qquad Nothing \rightarrow Nothing\\
\qquad\qquad\qquad\qquad Just\; a \quad\; \rightarrow Just\; (f\;\; a)
\end{array}
$$

$$map^{Maybe} \quad :: \forall\alpha\beta.Maybe\; \alpha \rightarrow\; (\alpha \rightarrow \beta) \rightarrow\; Maybe\; \beta$$

$$
\begin{array}{ll}
map^{Maybe} & = \Lambda\, \alpha\, \beta . \lambda\, m\, f .\ build^{Maybe}\ (map\#^{Maybe}\ \alpha\, \beta\, m\, f) \\
map\#^{Maybe} & :: \forall \alpha\beta . Maybe\ \alpha \rightarrow\ (\alpha \rightarrow \beta) \rightarrow\ (\forall\, \rho . \rho \rightarrow (\alpha \rightarrow \rho) \rightarrow \rho) \\
map\#^{Maybe} & = \Lambda\, \alpha\, \beta . \lambda\, m\, f . \Lambda\, \rho . \lambda\ nothing\ just . \\
& \quad \mathbf{let} \\
& \quad\quad c\ =\ \mathbf{case}\ m\ \mathbf{of} \\
& \quad\quad\quad\quad Nothing\ \rightarrow\ Nothing \\
& \quad\quad\quad\quad Just\ a\quad \rightarrow\ Just\ (f\ \ a) \\
& \quad \mathbf{in} \\
& \quad cata^{Maybe}\ \beta\, \rho\ nothing\ just\ c
\end{array}
$$

$$
\begin{array}{ll}
map^{Maybe} & :: \forall \alpha\beta . Maybe\ \alpha \rightarrow\ (\alpha \rightarrow \beta) \rightarrow\ Maybe\ \beta \\
map^{Maybe} & = \Lambda\, \alpha\, \beta . \lambda\, m\, f .\ build^{Maybe}\ (map\#^{Maybe}\ \alpha\, \beta\, m\, f) \\
map\#^{Maybe} & :: \forall \alpha\beta . Maybe\ \alpha \rightarrow\ (\alpha \rightarrow \beta) \rightarrow\ (\forall\, \rho . \rho \rightarrow (\alpha \rightarrow \rho) \rightarrow \rho) \\
map\#^{Maybe} & = \Lambda\, \alpha\, \beta . \lambda\, m\, f . \Lambda\, \rho . \lambda\ nothing\ just . \\
& \quad \mathbf{case}\ m\ \mathbf{of} \\
& \quad\quad Nothing\ \rightarrow\ nothing \\
& \quad\quad Just\ a\quad \rightarrow\ just\ (f\ \ a)
\end{array}
$$

Our final example shows when the third clause of $\mathcal{E}$ 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:

$$\mathbf{data}\ Rose\ a\ =\ Fork\ a\ [Rose\ a]$$

$$
\begin{array}{ll}
map^{Rose} & :: \forall \alpha\beta .\ Rose\ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow\ Rose\ \beta \\
map^{Rose} & = \Lambda \alpha\beta . \lambda\, r\, f .\ \mathbf{case}\ r\ \mathbf{of} \\
& \quad\quad Fork\ a\ rs\ \rightarrow\ \mathbf{let} \\
& \quad\quad\quad\quad g\ =\ \lambda\, r' . map^{Rose}\ \alpha\, \beta\, r'\, f \\
& \quad\quad\quad \mathbf{in} \\
& \quad\quad\quad Fork\ \beta\ (f\ \ a)\ (map^{[]}\ (Rose\ \alpha)\ (Rose\ \beta)\ g\ rs)
\end{array}
$$

After performing Steps 1 to 3 of buildify we get:

$$
\begin{array}{ll}
map^{Rose} & :: \forall \alpha\beta .\ Rose\ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow\ Rose\ \beta \\
map^{Rose} & = \Lambda \alpha\beta . \lambda\, r\, f . build^{Rose}\ (map\#^{Rose}\ \alpha\, \beta\, r\, f) \\
map\#^{Rose} & :: \forall \alpha\beta .\ Rose\ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow\ (\forall \rho . (\alpha \rightarrow [\rho] \rightarrow \rho) \rightarrow \rho) \\
map\#^{Rose} & = \Lambda \alpha\beta . \lambda\, r\, f . \Lambda \rho . \lambda\, fork .
\end{array}
$$

$$
\begin{array}{l}
\textbf{let} \\
\quad g \;=\; \lambda\, r'.map^{Rose}\,\alpha\,\beta\, r'\, f \\
\quad c \;=\; \textbf{case}\; r\; \textbf{of} \\
\qquad\qquad Fork\; a\; rs \;\rightarrow\; Fork\;\beta\;(f\;\; a)\;(map^{[]}\;(Rose\;\alpha)\;(Rose\;\beta)\;g\;rs) \\
\textbf{in} \\
cata^{Rose}\;\beta\,\rho\, fork\;\, c
\end{array}
$$

In the result, we observed the syntactic restriction in Core and **let** bound every argument. Notice, the $map^{[]}$ in the body of $map\#^{Rose}$.

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

$$
\begin{array}{ll}
f & :: \forall\bar{\alpha}.T\;\bar{\tau} \rightarrow \sigma \\
f & = \Lambda\bar{\alpha}.\lambda\; t.e \\
\Longrightarrow & \\
f & :: \forall\bar{\alpha}.T\;\bar{\tau} \rightarrow \sigma \\
f & = \Lambda\bar{\alpha}.\lambda\; t.\; cata^{T}\;\bar{\tau}\;(T\;\bar{\tau})\;(f_{C_1}\;\bar{\alpha})\;\ldots\;(f_{C_n}\;\bar{\alpha})\;t \\
f_{C_1} & = \Lambda\bar{\alpha}.\lambda\overline{z_1}.\;\textbf{let} \\
 & \qquad\quad f'_{C_1} \;=\; \lambda\; t.e \\
 & \qquad \textbf{in} \\
 & \qquad f'_{C_1}\;(C_1\;\bar{\tau}\;\overline{t_1}) \\
\vdots & \\
f_{C_n} & = \Lambda\bar{\alpha}.\lambda\overline{z_n}.\;\textbf{let} \\
 & \qquad\quad f'_{C_n} \;=\; \lambda\; t.e \\
 & \qquad \textbf{in} \\
 & \qquad f'_{C_n}\;(C_n\;\bar{\tau}\;\overline{t_n})
\end{array}
$$

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,

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 t. $\bar{\tau}$ is built up from type variables from $\bar{\alpha}$ and applications of fusible type constructors and primitive types (Int, Bool, etc). σ 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 $g_{[]}$ for the *Nil* constructor and worker $g_{(:)}$ for the *Cons* constructor.

In the rewritten bindings (after the $\Longrightarrow$), $\bar{\alpha}$, $T\bar{\tau}$, σ and e are as above. There are two new sets of variables (in each worker), $\overline{z_n}$ and $\overline{t_n}$. The variables denoted by t are appropriately typed (with respect to the type argument $\bar{\tau}$ to C_n), fresh variables and $\overline{t_n}$ and $\overline{z_n}$ 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 $C_1\,\bar{\tau}\,\overline{t_1}$. The usual way around this restriction is to **let** bind the expression $C_1\bar{\tau}\,\overline{t_1}$ 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.

$$
\begin{aligned}
&\mathit{length} :: \forall\, \alpha.[\alpha] \;\rightarrow\; \mathit{Int}\\
&\mathit{length} = \Lambda\, \alpha.\lambda\, l.\ \mathbf{case}\ l\ \mathbf{of}\\
&\qquad\qquad\qquad [] \;\rightarrow\; 0\\
&\qquad\qquad\qquad (x : xs) \;\rightarrow\; 1 \,+\, \mathit{length}\ \alpha\ xs
\end{aligned}
$$

According to Step 1, we rewrite this definition to:

$$
\begin{aligned}
&\mathit{length} \quad :: \forall\, \alpha.[\alpha] \;\rightarrow\; \mathit{Int}\\
&\mathit{length} \quad = \Lambda\, \alpha.\lambda\, l.\ \mathit{cata}^{[]}\ \alpha\ [\alpha]\ (\mathit{length}_{[]}\ \alpha)\ (\mathit{length}_{(:)}\ \alpha)\ l\\
&\mathit{length}_{[]} \quad = \Lambda\, \alpha.\ \mathbf{let}\\
&\qquad\qquad\qquad \mathit{length}^{*} = \Lambda\, \alpha.\lambda\, l.\ \mathbf{case}\ l\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad [] \;\rightarrow\; 0\\
&\qquad\qquad\qquad\qquad\qquad (x : xs) \;\rightarrow\; 1 \,+\, \mathit{length}\ \alpha\ xs\\
&\qquad\qquad\quad \mathbf{in}\\
&\qquad\qquad\quad \mathit{length}\ \alpha\ ([]\ \alpha)\\
&\mathit{length}_{(:)} \quad = \Lambda\, \alpha.\lambda\, z\ zs.\ \mathbf{let}\\
&\qquad\qquad\qquad\quad \mathit{length}^{*} = \Lambda\, \alpha.\lambda\, l.\ \mathbf{case}\ l\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\quad [] \;\rightarrow\; 0\\
&\qquad\qquad\qquad\qquad\qquad\quad (x : xs) \;\rightarrow\; 1 \,+\, \mathit{length}\ \alpha\ xs\\
&\qquad\qquad\qquad \mathbf{in}\\
&\qquad\qquad\qquad \mathit{length}\ \alpha\ ((:)\ \alpha\ (t :: \alpha)\ (ts :: [\alpha]))
\end{aligned}
$$

According to Step 2, call the Core Simplifier: the definitions marked with $*$ will get inlined, and two β reductions happen (in both bindings).

$$
\begin{array}{ll}
\mathit{length} & :: \forall\, \alpha.[\alpha] \;\rightarrow\; \mathit{Int} \\
\mathit{length} & = \Lambda\, \alpha.\lambda\, l.\ \mathit{cata}^{[]}\ \alpha\ [\alpha]\ (\mathit{length}_{[]}\ \alpha)\ (\mathit{length}_{(:)}\ \alpha)\ l \\
\mathit{length}_{[]} & = \Lambda\, \alpha.0 \\
\mathit{length}_{(:)} & = \Lambda\, \alpha.\lambda\, z\ zs.1 \;+\; \mathit{length}\ \alpha\ ts
\end{array}
$$

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 α and another of type $[\alpha]$.

$$\{t \;\rightarrow\; z,\ \mathit{length}\ \alpha\ ts \;\rightarrow\; zs\}$$

We rewrite the simplified bindings using these rules and get:

$$
\begin{array}{ll}
\mathit{length} & :: \forall\, \alpha.[\alpha] \;\rightarrow\; \mathit{Int} \\
\mathit{length} & = \Lambda\, \alpha.\lambda\, l.\ \mathit{cata}^{[]}\ \alpha\ [\alpha]\ (\mathit{length}_{[]}\ \alpha)\ (\mathit{length}_{(:)}\ \alpha)\ l \\
\mathit{length}_{[]} & = \Lambda\, \alpha.0 \\
\mathit{length}_{(:)} & = \Lambda\, \alpha.\lambda\, z\ zs.1 \;+\; zs
\end{array}
$$

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 $\mathit{length}_{[]}$ 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 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.

5.1.1 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:

$$
\begin{aligned}
&\mathit{append} \;::\; \forall\alpha.[\alpha] \to [\alpha] \to [\alpha]\\
&\mathit{append} \;=\; \Lambda\alpha.\lambda\, \mathit{xs}\ \mathit{ys}.\ \textbf{case}\ \mathit{xs}\ \textbf{of}\\
&\qquad\qquad\qquad\qquad [] \qquad\quad \to \mathit{ys}\\
&\qquad\qquad\qquad\qquad (:)\ x\ \mathit{xs} \to (:)\ \alpha\ x\ (\mathit{append}\ \alpha\ \mathit{xs}\ \mathit{ys})
\end{aligned}
$$

In the body of *append* and in recursive calls to *append* itself, α 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:

$$
\begin{aligned}
&\mathit{append} \;::\; \forall\alpha.[\alpha] \to [\alpha] \to [\alpha]\\
&\mathit{append} \;=\; \Lambda\,\alpha.\ \lambda\, \mathit{xs}\ \mathit{ys}.\ \textbf{let}\\
&\qquad\qquad\quad \mathit{append}' \;::\; [\alpha] \to [\alpha]\\
&\qquad\qquad\quad \mathit{append}' \;= \lambda\, \mathit{xs}.\ \textbf{case}\ t\ \textbf{of}\\
&\qquad\qquad\qquad\qquad [] \to \mathit{ys}\\
&\qquad\qquad\qquad\qquad (x : \mathit{xs}) \to (:)\ \alpha\ x\ (\mathit{append}'\ \mathit{xs})\\
&\qquad\qquad \textbf{in}\\
&\qquad\qquad \mathit{append}'\ \mathit{xs}
\end{aligned}
$$

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:

$$\mathit{append} \;::\; \forall\alpha.[\alpha] \to [\alpha] \to [\alpha]$$

$$
\begin{array}{lll}
\mathit{append} & = \Lambda\,\alpha.\,\lambda\,\mathit{xs}\;\mathit{ys}.\;\textbf{let} & \\
 & \quad \mathit{append}' & :: [\alpha] \;\rightarrow\; [\alpha] \\
 & \quad \mathit{append}' & = \lambda\,\mathit{xs}.\mathit{cata}^{[]}\;\alpha\;[\alpha]\;\mathit{append}'_{[]}\;\mathit{append}'_{(:)}\;\mathit{xs} \\
 & \quad \mathit{append}'_{[]} & :: [\alpha] \\
 & \quad \mathit{append}'_{[]} & = \mathit{ys} \\
 & \quad \mathit{append}'_{(:)} & :: \alpha \;\rightarrow\; [\alpha] \;\rightarrow\; [\alpha] \\
 & \quad \mathit{append}'_{(:)} & = \lambda\,z\;\mathit{zs}.\;(:)\;\alpha\;z\;\mathit{zs} \\
 & \textbf{in} & \\
 & \mathit{append}'\;\mathit{xs} &
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{append} & :: \forall\alpha.[\alpha] \;\rightarrow\; [\alpha] \;\rightarrow\; [\alpha] & \\
\mathit{append} & = \Lambda\,\alpha.\,\lambda\,\mathit{xs}\;\mathit{ys}.\;\textbf{let} & \\
 & \quad \mathit{append}'_{[]} & :: [\alpha] \\
 & \quad \mathit{append}'_{[]} & = \mathit{ys} \\
 & \quad \mathit{append}'_{(:)} & :: \alpha \;\rightarrow\; [\alpha] \;\rightarrow\; [\alpha] \\
 & \quad \mathit{append}'_{(:)} & = \lambda\,z\;\mathit{zs}.\;(:)\;\alpha\;z\;\mathit{zs} \\
 & \textbf{in} & \\
 & \mathit{cata}^{[]}\;\alpha\;[\alpha]\;\mathit{append}'_{[]}\;\mathit{append}'_{(:)}\;\mathit{xs} &
\end{array}
$$

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):

$$
\begin{array}{lll}
\mathit{append} & :: \forall\alpha.[\alpha] \;\rightarrow\; [\alpha] \;\rightarrow\; [\alpha] & \\
\mathit{append} & = \Lambda\,\alpha.\,\lambda\,\mathit{xs}\;\mathit{ys}.\;\textbf{let} & \\
 & \quad \mathit{append}'_{[]} & :: [\alpha] \;\rightarrow\; [\alpha] \\
 & \quad \mathit{append}'_{[]} & = \lambda\,\mathit{ys}.\;\mathit{ys} \\
 & \quad \mathit{append}'_{(:)} & :: \alpha \;\rightarrow\; [\alpha] \;\rightarrow\; [\alpha] \;\rightarrow\; [\alpha] \\
 & \quad \mathit{append}'_{(:)} & = \lambda\,z\;\mathit{zs}\;\mathit{ys}.\;(:)\;\alpha\;z\;(\mathit{zs}\;\mathit{ys}) \\
 & \textbf{in} & \\
 & \mathit{cata}^{[]}\;\alpha\;[\alpha]\;\mathit{append}'_{[]}\;\mathit{append}'_{(:)}\;\mathit{xs}\;\mathit{ys} &
\end{array}
$$

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, *map*, *span*, *break*, *takeWhile*, *filter*, *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:

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

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

5.1.2 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 *level* function for trees. level has type $Tree\ \alpha \rightarrow Int \rightarrow [\alpha]$; 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. *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

$$\textbf{data}\ Tree\ \alpha\ =\ Empty \mid Branch\ \alpha\ (Tree\ \alpha)\ (Tree\ \alpha)$$

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

$$\begin{array}{ll}
cata^{Tree} & :: \forall\, \alpha\, \rho.\rho \rightarrow (\alpha \rightarrow \rho \rightarrow \rho \rightarrow \rho) \rightarrow Tree\ \alpha \rightarrow \rho \\
cata^{Tree} & = \Lambda\, \alpha\, \rho.\lambda\ e\ b\ t.\ \textbf{case}\ t\ \textbf{of} \\
 & \qquad\qquad\qquad \begin{array}{ll} Empty & \rightarrow e \\ Branch\ x\ lt\ rt & \rightarrow b\ x\ (cata^{Tree}\ \alpha\ \rho\ lt)\ (cata^{Tree}\ \alpha\ \rho\ rt) \end{array}
\end{array}$$

the naive definition of level:

$$\begin{array}{ll}
level\ Empty\ n & = [] \\
level\ (Branch\ x\ lt\ rt)\ 0 & = [x] \\
level\ (Branch\ x\ lt\ rt)\ n \mid n\ >\ 0 & = level\ lt\ (n-1)\ +\!\!+\ level\ rt\ (n-1)
\end{array}$$

which, in turn translates to (Desugarer):

$$
\begin{aligned}
&\mathit{level} :: \forall\,\alpha.\mathit{Tree}\ \alpha \ \rightarrow\ \mathit{Int}\ \rightarrow\ [\alpha] \\
&\mathit{level} = \Lambda\,\alpha.\lambda\ t\ i.\ \textbf{case}\ t\ \textbf{of} \\
&\qquad \mathit{Empty} \rightarrow [\,]\ \alpha \\
&\qquad \mathit{Branch}\ x\ \mathit{lt}\ \mathit{rt} \rightarrow \textbf{case}\ i == 0\ \textbf{of} \\
&\qquad\qquad \mathit{True} \rightarrow (:)\ \alpha\ x\ ([\,]\ \alpha) \\
&\qquad\qquad \mathit{False} \rightarrow \textbf{case}\ i > 0\ \textbf{of} \\
&\qquad\qquad\qquad \mathit{True} \rightarrow \mathit{append}\ \alpha \\
&\qquad\qquad\qquad\qquad (\mathit{level}\ \alpha\ \mathit{lt}\ (i-1)) \\
&\qquad\qquad\qquad\qquad (\mathit{level}\ \alpha\ \mathit{rt}\ (i-1)) \\
&\qquad\qquad\qquad \mathit{False} \rightarrow [\,]\ \alpha
\end{aligned}
$$

The reader is invited to verify that buildify succeeds and we get:

$$
\begin{aligned}
&\mathit{level} :: \forall\,\alpha.\mathit{Tree}\ \alpha \ \rightarrow\ \mathit{Int}\ \rightarrow\ [\alpha] \\
&\mathit{level} = \Lambda\,\alpha.\lambda\ t\ i.\mathit{build}^{[\,]}\ (\mathit{level}'\ \alpha\ t\ i) \\
&\mathit{level}' :: \forall\,\alpha.\mathit{Tree}\ \alpha \ \rightarrow\ \mathit{Int}\ \rightarrow\ (\forall\,\rho.\rho\ \rightarrow\ (\alpha \rightarrow \rho \rightarrow \rho)\ \rightarrow\ \rho) \\
&\mathit{level}' = \Lambda\,\alpha.\lambda\ t\ i.\Lambda\,\rho.\lambda\ e\ b. \\
&\quad \textbf{case}\ t\ \textbf{of} \\
&\qquad \mathit{Empty} \rightarrow e \\
&\qquad \mathit{Branch}\ x\ \mathit{lt}\ \mathit{rt} \rightarrow \textbf{case}\ i == 0\ \textbf{of} \\
&\qquad\qquad \mathit{True} \rightarrow b\ x\ e \\
&\qquad\qquad \mathit{False} \rightarrow \textbf{case}\ i > 0\ \textbf{of} \\
&\qquad\qquad\qquad \mathit{True} \rightarrow \mathit{level}'\ \alpha\ \mathit{lt}\ (i-1) \\
&\qquad\qquad\qquad\qquad (\mathit{level}'\ \alpha\ \mathit{rt}\ (i-1)\ e\ b) \\
&\qquad\qquad\qquad\qquad b \\
&\qquad\qquad\qquad \mathit{False} \rightarrow e
\end{aligned}
$$

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 *level′*. It is the traversal of the right branch, which is an artefact of *append* getting inlined. We start to catify this definition of *level*. After rewriting and simplification we have:

$$
\begin{aligned}
&\mathit{level} :: \forall\,\alpha.\mathit{Tree}\ \alpha \ \rightarrow\ \mathit{Int}\ \rightarrow\ [\alpha] \\
&\mathit{level} = \Lambda\,\alpha.\lambda\ t\ i.\mathit{build}^{[\,]}\ (\mathit{level}'\ \alpha\ t\ i) \\
&\mathit{level}' :: \forall\,\alpha.\mathit{Tree}\ \alpha \ \rightarrow\ \mathit{Int}\ \rightarrow\ (\forall\,\rho.\rho\ \rightarrow\ (\alpha \rightarrow \rho \rightarrow \rho)\ \rightarrow\ \rho) \\
&\mathit{level}' = \Lambda\,\alpha.\lambda\ t\ i.\Lambda\,\rho.\lambda\ e\ b.\ \mathit{cata}^{\mathit{Tree}}\ \alpha\ \rho\ (\mathit{level}'_{\mathit{Empty}}\ \alpha)\ (\mathit{level}'_{\mathit{Branch}}\ \alpha)\ t\ i\ e\ b \\
&\mathit{level}'_{\mathit{Empty}} = \Lambda\,\alpha.\lambda\ i\ e\ b.e
\end{aligned}
$$

$$
\begin{array}{l}
level'_{Branch} = \lambda\, \alpha.\lambda\ zx\ zlt\ zrt\ i\ e\ b.\ \mathbf{case}\ i == 0\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \begin{array}{ll}
True & \to b\ tx\ e \\
False & \to \mathbf{case}\ i > 0\ \mathbf{of} \\
& \quad \begin{array}{ll}
True & \to level'\ \alpha\ tlt\ (i-1) \\
& \qquad (level'\ \alpha\ trt\ (i-1)\ e\ b) \\
& \qquad b \\
False & \to e
\end{array}
\end{array}
\end{array}
$$

The rewrite system is constructed from the function $level'$, the new, free variables tx, tlt and trt and the new appropriately typed variables zx, zlt and zrt.

$$\mathcal{R} \;=\; \{\; tx \;\to\; zx,\; level'\,\alpha\,tlt \;\to\; zlt,\; level'\,\alpha\,trt \;\to\; zrt\,\}$$

The second rewriting then gives:

$$
\begin{array}{ll}
level & :: \forall\,\alpha.\,Tree\ \alpha \;\to\; Int \;\to\; [\alpha] \\
level & = \Lambda\,\alpha.\lambda\ t\ i.build^{[]}\ (level'\ \alpha\ t\ i) \\
level' & :: \forall\,\alpha.\,Tree\ \alpha \;\to\; Int \;\to\; (\forall\,\rho.\rho \;\to\; (\alpha \to \rho \to \rho) \;\to\; \rho) \\
level' & = \Lambda\,\alpha.\lambda\ t\ i.\Lambda\,\rho.\lambda\ e\ b.\ cata^{Tree}\ \alpha\ \rho\ (level'_{Empty}\ \alpha)\ (level'_{Branch}\ \alpha)\ t\ i\ e\ b \\
level'_{Empty} & = \Lambda\,\alpha.\lambda\ i\ e\ b.e \\
level'_{Branch} & = \lambda\,\alpha.\lambda\ zx\ zlt\ zrt\ i\ e\ b.\ \mathbf{case}\ i == 0\ \mathbf{of} \\
& \qquad\qquad\qquad \begin{array}{ll}
True & \to b\ zx\ e \\
False & \to \mathbf{case}\ i > 0\ \mathbf{of} \\
& \quad \begin{array}{ll}
True & \to zlt\ (i-1)\ (zrt\ (i-1)\ e\ b)\ b \\
False & \to e
\end{array}
\end{array}
\end{array}
$$

Notice, that $cata^{Tree}\ \alpha\ \rho\ (level'_{Empty}\ \alpha)(level'_{Branch}\ \alpha)\ t$ 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 $cata^T$ 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 *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:

   $f \quad :: \forall \bar{\alpha}. T\ \bar{\tau} \rightarrow \bar{\sigma}$

$$
\begin{array}{lll}
f & = \Lambda\bar{\alpha}.\lambda\, t\, \bar{v}.e & \\
\Longrightarrow & & \\
f & :: \forall\bar{\alpha}.T\ \bar{\tau} \rightarrow \bar{\sigma} & \\
f & = \Lambda\bar{\alpha}.\lambda\, t\, \bar{v}.\ cata^T\ \bar{\tau}\ (T\ \bar{\tau})\ (f_{C_1}\ \bar{\alpha})\ \ldots\ (f_{C_n}\ \bar{\alpha})\ t & \\
f_{C_1} & = \Lambda\bar{\alpha}.\lambda\overline{z_1}.\lambda\bar{v}.\ \mathbf{let} & \\
 & \qquad\qquad f'_{C_1} = \lambda\, t.e & \\
 & \qquad\quad \mathbf{in} & \\
 & \qquad\quad f'_{C_1}\ (C_1\ \bar{\tau}\ \overline{t_1}) & \\
\vdots & & \\
f_{C_n} & = \Lambda\bar{\alpha}.\lambda\overline{z_n}.\lambda\bar{v}\ \mathbf{let} & \\
 & \qquad\qquad f'_{C_n} = \lambda\, t.e & \\
 & \qquad\quad \mathbf{in} & \\
 & \qquad\quad f'_{C_n}\ (C_n\ \bar{\tau}\ \overline{t_n}) &
\end{array}
$$

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.

$$
\begin{array}{ll}
map^{[]} & :: \forall\,\alpha\,\beta.[\alpha] \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta] \\
map^{[]} & = \Lambda\,\alpha\,\beta.\lambda\, t.\lambda\, f.\ \mathbf{case}\ t\ \mathbf{of} \\
 & \qquad\qquad [] \rightarrow []\ \beta \\
 & \qquad\qquad (x : xs) \rightarrow (:)\ \beta\ (f\ x)\ (map^{[]}\ \alpha\ \beta\ xs\ f)
\end{array}
$$

According to Step 1 we split this definition into three:

$$
\begin{array}{lll}
map^{[]} & :: & \forall\,\alpha\,\beta.[\alpha] \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta] \\
map^{[]} & = & \Lambda\,\alpha\,\beta.\lambda\,t.\lambda\,f.\ cata^{[]}\ \alpha\ [\alpha]\ (map^{[]}_{[]}\ \alpha\,\beta)\ (map^{[]}_{(:)}\ \alpha\,\beta)\ t\,f \\
map^{[]}_{[]} & :: & \forall\,\alpha\,\beta.(\alpha \rightarrow \beta) \rightarrow [\beta] \\
map^{[]}_{[]} & = & \Lambda\,\alpha\beta.\lambda\,f.\ \mathbf{let} \\
 & & \qquad map^{[]} = \Lambda\,\alpha\,\beta.\ \lambda\,t.\ \mathbf{case}\ t\ \mathbf{of} \\
 & & \qquad\qquad [] \rightarrow []\ \beta \\
 & & \qquad\qquad (x : xs) \rightarrow (:)\ \beta\ (f\ x)\ (map^{[]}\ \alpha\,\beta\ xs\ f)) \\
 & & \mathbf{in} \\
 & & map^{[]}\ \alpha\,\beta\ ([]\ \alpha) \\
map^{[]}_{(:)} & :: & \forall\,\alpha\,\beta.\alpha \rightarrow ((\alpha \rightarrow \beta) \rightarrow [\beta]) \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta] \\
map^{[]}_{(:)} & = & \Lambda\,\alpha\beta.\lambda\,z\ zs.\lambda\,f.\ \mathbf{let} \\
 & & \qquad map^{[]} = \Lambda\,\alpha\,\beta.\ \lambda\,t.\ \mathbf{case}\ t\ \mathbf{of} \\
 & & \qquad\qquad [] \rightarrow []\ \beta \\
 & & \qquad\qquad (x : xs) \rightarrow (:)\ \beta \\
 & & \qquad\qquad\qquad (f\ x) \\
 & & \qquad\qquad\qquad (map^{[]}\ \alpha\,\beta\ xs\ f)) \\
 & & \mathbf{in} \\
 & & map^{[]}\ \alpha\,\beta\ ((:)\ \alpha\ (t :: \alpha)\ (ts :: [\alpha]))
\end{array}
$$

The Simplifier is called, which performs a few β reductions:

$$
\begin{array}{lll}
map^{[]} & :: & \forall\,\alpha\,\beta.[\alpha] \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta] \\
map^{[]} & = & \Lambda\,\alpha\,\beta.\lambda\,t.\lambda\,f.\ cata^{[]}\ \alpha\ [\alpha]\ (map^{[]}_{[]}\ \alpha\,\beta)\ (map^{[]}_{(:)}\ \alpha\,\beta)\ t\,f \\
map^{[]}_{[]} & :: & \forall\,\alpha\,\beta.(\alpha \rightarrow \beta) \rightarrow [\beta] \\
map^{[]}_{[]} & = & \Lambda\,\alpha\beta.\lambda\,f.[]\ \beta \\
map^{[]}_{(:)} & :: & \forall\,\alpha\,\beta.\alpha \rightarrow ((\alpha \rightarrow \beta) \rightarrow [\beta]) \rightarrow (\alpha \rightarrow \beta) \rightarrow [\beta] \\
map^{[]}_{(:)} & = & \Lambda\,\alpha\beta.\lambda\,z\ zs.\lambda\,f.\ (:)\ \beta\ (f\ t)\ (map^{[]}\ \alpha\,\beta\ ts\ f)
\end{array}
$$

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 $map^{[]}\ \alpha\,\beta\ ts$ with z and zs. The rules are:

$$\{t \rightarrow z,\ map^{[]}\ \alpha\,\beta\ ts \rightarrow zs\}$$

After rewriting we get:

$$
\begin{array}{ll}
map^{[]} & :: \forall\, \alpha\, \beta . [\alpha] \;\rightarrow\; (\alpha \;\rightarrow\; \beta) \;\rightarrow\; [\beta] \\
map^{[]} & = \Lambda\, \alpha\, \beta . \lambda\, t . \lambda\, f .\; cata^{[]}\; \alpha\; [\alpha]\; (map^{[]}_{[]}\; \alpha\; \beta)\; (map^{[]}_{(:)}\; \alpha\; \beta)\; t\; f \\
map^{[]}_{[]} & :: \forall\, \alpha\, \beta . (\alpha \;\rightarrow\; \beta) \;\rightarrow\; [\beta] \\
map^{[]}_{[]} & = \Lambda\, \alpha\beta . \lambda\, f . []\; \beta \\
map^{[]}_{(:)} & :: \forall\, \alpha\, \beta . \alpha \;\rightarrow\; ((\alpha \;\rightarrow\; \beta) \;\rightarrow\; [\beta]) \;\rightarrow\; (\alpha \;\rightarrow\; \beta) \;\rightarrow\; [\beta] \\
map^{[]}_{(:)} & = \Lambda\, \alpha\beta . \lambda\, z\; zs . \lambda\, f . (:)\; \beta\; (f\; z)\; (zs\; f)
\end{array}
$$

Notice the similarity with the second definition of *append* given on page 66. In particular, the type of zs has changed from $[\beta]$ to $((\alpha \rightarrow \beta) \rightarrow [\beta])$, 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*.

$$
\begin{array}{ll}
lrev & :: \forall\, \alpha . [\alpha] \;\rightarrow\; [\alpha] \;\rightarrow\; [\alpha] \\
lrev & = \Lambda\, \alpha . \lambda\, xs\; ys .\; \textbf{case}\; xs\; \textbf{of} \\
 & \qquad\qquad\qquad\qquad [] \qquad\quad \rightarrow ys \\
 & \qquad\qquad\qquad\qquad (:)\; x\; xs \;\rightarrow lrev\; \alpha\; xs\; ((:)\; \alpha\; x\; ys)
\end{array}
$$

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

$$
\begin{array}{l}
\forall\, a : A \rightarrow A',\; a : A \rightarrow A',\; r : R \rightarrow R'. \\
\;\textbf{if}\; r \;\cdot\; n \;=\; n' \;\cdot\; b \\
\;\wedge\; r \;\cdot\; c\; x\; xs \;=\; c'\; x'\; xs' \;\cdot\; b \;\Leftarrow\; a\; x \;=\; x' \;\wedge\; r \;\cdot\; xs \;=\; xs' \;\cdot\; b \\
\;\wedge\; a\; ys \;=\; ys' \;.\; map^{[]}\; a \\
\;\wedge\; b\; w \;=\; w' \\
\;\wedge\; r\;\; strict \\
\;\textbf{then} \\
\qquad r\; (cata\; n\; c\; ys\; w) \;=\; cata\; n'\; c'\; ys'\; (b\; w)
\end{array}
\qquad (5.1)
$$

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 λ and type Λ) 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.

$$
\begin{array}{l}
f \;=\; \mathit{\Lambda}\,\bar{\alpha}.\lambda\,\bar{v}.e \\
\Longrightarrow \\
f \;=\; \mathit{\Lambda}\,\bar{\alpha}.\lambda\,\bar{v}.\,\mathbf{let} \\
\qquad\qquad\qquad f' \;=\; \mathit{\Lambda}\,\mathtt{notStatic}\,(\bar{\alpha}).\lambda\,\mathtt{notStatic}\,(\bar{v}).e' \\
\qquad\qquad\quad \mathbf{in} \\
\qquad\qquad\quad f'\,\mathtt{notStatic}\,(\bar{\alpha})\,\mathtt{notStatic}\,(\bar{v})
\end{array}
$$

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.

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,

$$
\begin{array}{llll}
\textbf{data}\ T\ a & = T1\ a & \mid T2\,(K\ a) & \mid T3\,(T\ a) \\
\textbf{data}\ K\ a & = K1 & \mid K2\ a\,(T\ a) & \mid K3\ a\ a\,(K\ a) \\
& \langle\text{is transformed to}\rangle & & \\
\textbf{data}\ RTK\ a & = RT1\ a & \mid RT2\,(RTK\ a) & \mid RT3\,(RTK\ a) \\
& \mid\ RK1 & \mid RK2\ a\,(RTK\ a) & \mid RK3\ a\ a\,(RTK\ a)
\end{array}
$$

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:

$$
\begin{array}{ll}
& \langle\text{type variables are dropped for simplicity}\rangle \\
map^T\ f\ (T1\ x) & = T1\ (f\ x) \\
map^T\ f\ (T2\ k) & = T2\ (map^K\ f\ k) \\
map^T\ f\ (T3\ t) & = T3\ (map^T\ f\ t) \\
map^K\ f\ K1 & = K1 \\
map^K\ f\ (K2\ x\ t) & = K2\ (f\ x)\ (map^T\ f\ t) \\
map^K\ f\ (K3\ x\ y\ k) & = K3\ (f\ x)\ (f\ y)\ (map^K\ f\ k) \\
& \langle\text{becomes}\rangle \\
map^{TK}\ f\ (RT1\ x) & = RT1\ (f\ x) \\
map^{TK}\ f\ (RT2\ k) & = RT2\ (map^{TK}\ f\ k) \\
map^{TK}\ f\ (RT3\ t) & = RT3\ (map^{TK}\ f\ t) \\
map^{TK}\ f\ RK1 & = RK1 \\
map^{TK}\ f\ (RK2\ x\ t) & = RK2\ (f\ x)\ (map^{TK}\ f\ t) \\
map^{TK}\ f\ (RK3\ x\ y\ k) & = RK3\ (f\ x)\ (f\ y)\ (map^{TK}\ f\ k)
\end{array}
$$

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: $cata^{\{T_1\,\bar{\alpha}\,+\,\ldots\,+\,T_n\,\bar{\alpha}\},\,T_j}$ will stand for the catamorphism which reduces a data structure of type T_j. Sometimes for convenience, we will use the notation $cata^{\bar{T},\,T_j}$. If we wanted to be overly precise, we could repeat the type variables $\bar{\alpha}$ for T_j, as in $cata^{\bar{T},\,T_j\,\bar{\alpha}}$, but we will refrain from doing so. We could even write $cata^{\overline{T\,\bar{\alpha}},\,T_j\,\bar{\alpha}}$ 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:

$$
\begin{aligned}
map^{\bar{T},\,T_1} &= \Lambda\,\bar{\alpha}\,\Lambda\,\bar{\beta}.\lambda\,\bar{f}.\lambda\,t.\\
&\quad \textbf{case}\ t\ \textbf{of}\\
&\quad\quad \{\ T_{1,\,i}\,\bar{v}\ \rightarrow\ T_{1,\,i}\,\bar{\beta}\,(M^{\,\bar{T},\,T_1}\,\bar{f}\,(map^{\,\bar{T},\,T_1}\,\bar{\alpha}\,\bar{\beta}\,\bar{f})\,\bar{v})\ \}_{i=1}^{n}\\
&\ \vdots\\
map^{\bar{T},\,T_m} &= \Lambda\,\bar{\alpha}\,\Lambda\,\bar{\beta}.\lambda\,\bar{f}.\lambda\,t.\\
&\quad \textbf{case}\ t\ \textbf{of}\\
&\quad\quad \{\ T_{m,\,i}\,\bar{v}\ \rightarrow\ T_{m,\,i}\,\bar{\beta}\,(M^{\,\bar{T},\,T_m}\,\bar{f}\,(map^{\,\bar{T},\,T_m}\,\bar{\alpha}\,\bar{\beta}\,\bar{f})\,\bar{v})\ \}_{i=1}^{n}
\end{aligned}
$$

$\bar{f}$, 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.

$$
\begin{array}{l}
M^{\bar{T},\,T_i}\,\bar{f}\;\bar{g}\;v \;=\; \mathcal{M}^{\bar{T},T_i}\,\bar{f}\;\bar{g}\;(\texttt{typeOf}\;v)\;v \\
\quad \textbf{where} \\
\quad\quad \begin{array}{lcl}
\mathcal{M}^{\bar{T},T_i}\,\bar{f}\;\bar{g}\;[\![\,\mathit{primitive}\,]\!] & = & \lambda\,x.x \\
\mathcal{M}^{\bar{T},T_i}\,\bar{f}\;\bar{g}\;[\![\alpha]\!] & = & \lambda\,x.\{f_i\;x \mid \texttt{sourceTypeOf}\;(f_i) \;=\; \alpha \;\wedge\; i \;\in\; \{1\ldots\;n\}\} \\
\mathcal{M}^{\bar{T},T_i}\,\bar{f}\;\bar{g}\;[\![T\;\bar{\alpha}]\!] & = & \lambda\,x.\{g_i\;x \mid \texttt{tyConOf}\;(g) \;=\; T \;\wedge\; i \;\in\; \{1\ldots\;m\}\} \\
\mathcal{M}^{\bar{T},T_i}\,\bar{f}\;\bar{g}\;[\![K\;\bar{\tau}]\!] & = & \lambda\,x.\mathit{map}^{K\bar{\alpha}}\;\overline{(\texttt{tyVarsOf}(\texttt{sourceTypeOf}\;g_1))} \\
& & \qquad\qquad\;\; \overline{(\texttt{tyVarsOf}(\texttt{targetTypeOf}\;g_1))} \\
& & \qquad\qquad\;\; (\mathcal{M}^{\bar{T},T_i}\,\bar{f}\;\bar{g}\;[\![\,\bar{\tau}\,]\!]) \\
& & \qquad\qquad\;\; x
\end{array}
\end{array}
$$

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 $\mathcal{M}$ is applied to a type variable: we select the appropriate f to rewrite the given occurrence. The third clause applies when $\mathcal{M}$ 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 α and β. 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:

$$
\mathit{map}^K \;::\; \forall\,\alpha\beta\gamma\delta.(\alpha\;\rightarrow\;\gamma)\;\rightarrow\;(\beta\;\rightarrow\delta)\;\rightarrow\;K\;\alpha\;\beta\;\rightarrow\;K\;\gamma\;\delta
$$

Applications of map^K, when applied to functions of type $(\alpha \rightarrow \gamma)$ and $(\beta \;\rightarrow\; \delta)$, 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: $map^{\overline{T\bar{\alpha}},\, T_1}\ \alpha_1\ \alpha_2\ \alpha_3\alpha_4\ f_1\ f_2$, 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 $\texttt{tyVarsOf}(\texttt{targetTypeOf}\ g_1)$ and $\texttt{tyVarsOf}(\texttt{sourceTypeOf}\ g_1)$. With a slight abuse of the notation we extend $\mathcal{M}$ to apply to a list of types ($[\![\bar{\tau}]\!]$), which explains the 'bar' over $\texttt{tyVarsOf}(\texttt{targetTypeOf}g_1)$: 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:

$$
\begin{aligned}
cata^{\bar{T}\,,\,T_1}\ &= \Lambda\,\bar{\alpha}.\Lambda\,\bar{\rho}.\,\lambda\,\overline{c_1}\ldots\overline{c_m}.\lambda\,t.\\
&\qquad\textbf{case}\ t\ \textbf{of}\\
&\qquad\quad\{\ T_{1,\,i}\ \bar{v}\ \rightarrow\ c_{1,\,i}\ (E^{\bar{T}}\ (cata^{\bar{T}}\ \bar{\alpha}\,\bar{\rho}\,\overline{c_1}\ldots\overline{c_m})\ \bar{v})\ \}_{i=1}^{n_1}\\
&\vdots\\
cata^{\bar{T}\,,\,T_m}\ &= \Lambda\,\bar{\alpha}.\Lambda\,\bar{\rho}.\,\lambda\,\overline{c_1}\ldots\overline{c_m}.\lambda\,t.\\
&\qquad\textbf{case}\ t\ \textbf{of}\\
&\qquad\quad\{\ T_{m,\,i}\ \bar{v}\ \rightarrow\ c_{m,\,i}\ (E^{\bar{T}}\ (cata^{\bar{T}}\ \bar{\alpha}\,\bar{\rho}\,\overline{c_1}\ldots\overline{c_m})\ \bar{v})\ \}_{i=1}^{n_m}
\end{aligned}
$$

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 $E^{\overline{T\bar{\alpha}}}$. $cata^{\overline{T\bar{\alpha}}}$ stands for $cata^{\overline{T\bar{\alpha}},\,T_1}\ \ldots\ cata^{\overline{T\bar{\alpha}},\,T_n}$. $\bar{\rho}$ 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 $\texttt{NumOfConstrs}\,(T)$ denotes the number of constructors T has, then $cata^{\overline{T\,\alpha}\,,\,T_m}$ will have $\sum_{i=0}^{n}\texttt{NumOfConstrs}(T_i)$ arguments.

We also need to give a definition for E:

$$E^{\bar{T}}\ \bar{g}\ v\ =\ \mathcal{E}^{\bar{T}}\ \bar{g}\ (\texttt{typeOf}\ v)\ v$$

⟨cata of case rule⟩

$$cata^{\bar{T},T_i}\ \bar{\tau}\ \bar{\rho}\ \overline{c_1}\ldots\overline{c_n}\ (\ \textbf{case}\ \mathit{Expr}\ \textbf{of}\ \overline{\{C\ \bar{v}\ \rightarrow\ e\}}) \rightarrow \textbf{case}\ \mathit{Expr}\ \textbf{of}\ \overline{\{\ C\ \bar{v}\ \rightarrow\ cata^{\bar{T},T_i}\ \bar{\tau}\ \bar{\rho}\ \overline{c_1}\ldots\overline{c_n}\ e\ \}}$$

⟨cata of known constructor rule⟩

$$cata^{\bar{T},T_i}\ \bar{\tau}\ \bar{\rho}\ \overline{c_1}\ldots\overline{c_n}\ (C_i\ \bar{v}) \rightarrow c_i\ (E^{\bar{T},T_i}\ (cata^{\bar{T},T_i}\ \bar{\tau}\ \rho\ \overline{c_1}\ldots\overline{c_n})\ \bar{v})$$

⟨`cata-build` rule⟩

$$cata^{\bar{T},T_i}\ \bar{\tau}\ \bar{\rho}\ \overline{c_1}\ldots\overline{c_n}\ (build^{\bar{T},T_i}\ \bar{\rho}\, f) \rightarrow f\ \bar{\rho}\ \overline{c_1}\ldots\overline{c_n}$$

⟨cata of error rule⟩

$$cata^{\bar{T},T_i}\ \bar{\tau}\ \bar{\rho}\ \overline{c_1}\ldots\overline{c_n}\ error \rightarrow error$$

Figure 5.1 Cata-Core rules in the presence of mutually recursive datatypes

where

$$\begin{aligned}
\mathcal{E}^{\bar{T}}\ \bar{g}\ [\![\,\mathit{primitive\ type}]\!] &= \lambda\, x.x \\
\mathcal{E}^{\bar{T}}\ \bar{g}\ [\![\alpha]\!] &= \lambda\, x.x \\
\mathcal{E}^{\bar{T}}\ \bar{g}\ [\![T\ \bar{\alpha}]\!] &= \lambda\, x.g_i\ x,\ \textbf{if}\ T\ =\ T_i \\
\mathcal{E}^{\bar{T}}\ \bar{g}\ [\![K\ \bar{\tau}]\!] &= \lambda\, x.map^{K\bar{\alpha}}\ (\texttt{sourceTypeOf}\ g) \\
&\qquad (\texttt{targetTypeOf}\ g) \\
&\qquad (\mathcal{E}^{\bar{T}}\ \bar{g}\ [\![\ \bar{\tau}\ ]\!]) \\
&\qquad x
\end{aligned}$$

The third clause of $\mathcal{E}$ selects the appropriate *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 *build* does change to

$$build^{\bar{T},\,T_i}\ g\ =\ g\ \texttt{Constrs}(T_1)\ \ldots\ \texttt{Constrs}(T_n)$$

which is reflected in the `cata-build` rule: *build* now applies its argument, g, to *all* the constructors of the mutually recursive group. The extended rules are shown in Figure 5.1.

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:

$$
\begin{array}{ll}
f_1 & :: \forall\bar{\alpha}.\bar{\sigma} \rightarrow\ T_1\bar{\tau} \\
f_1 & = \Lambda\bar{\alpha}.\lambda\bar{v}.e_1 \\
\vdots & \\
f_n & :: \forall\bar{\alpha}.\bar{\sigma} \rightarrow\ T_n\bar{\tau} \\
f_n & = \Lambda\bar{\alpha}.\lambda\bar{v}.e_n \\
\Longrightarrow & \\
 & \text{— The wrappers} \\
f_1 & :: \forall\bar{\alpha}.\bar{\sigma} \rightarrow\ T_1\bar{\tau} \\
f_1 & = \Lambda\bar{\alpha}.\lambda\bar{v}.build^{\overline{T},T_1}\ (f_1'\ \bar{\alpha}\ \bar{v}) \\
\vdots & \\
f_n & :: \forall\bar{\alpha}.\bar{\sigma} \rightarrow\ T_n\bar{\tau} \\
f_n & = \Lambda\bar{\alpha}.\lambda\bar{v}.build^{\overline{T},T_n}\ (f_n'\ \bar{\alpha}\ \bar{v}) \\
 & \text{— The workers} \\
f_1' & :: \forall\bar{\alpha}.\bar{\sigma} \rightarrow (\forall\bar{\rho}.\ \texttt{monoConstrs}(T_1\bar{\tau})[\rho_1/T_1\bar{\tau}] \rightarrow \\
 & \qquad\qquad\qquad \vdots \\
 & \qquad\qquad\qquad \texttt{monoConstrs}(T_n\bar{\tau})[\rho_n/T_n\bar{\tau}] \rightarrow\ \rho_1) \\
f_1' & = \Lambda\bar{\alpha}.\lambda\bar{v}.\Lambda\bar{\rho}.\lambda\overline{c_1}\ \ldots\lambda\overline{c_n}.\ cata^{T_1}\ \bar{\tau}\ \bar{\rho}\ \overline{c_1}\ \ldots\ \overline{c_n}\ e_1 \\
\vdots & \\
f_n' & :: \forall\bar{\alpha}.\bar{\sigma} \rightarrow (\forall\bar{\rho}.\ \texttt{monoConstrs}(T_1\bar{\tau})[\rho_1/T_1\bar{\tau}] \rightarrow \\
 & \qquad\qquad\qquad \vdots \\
 & \qquad\qquad\qquad \texttt{monoConstrs}(T_n\bar{\tau})[\rho_n/T_n\bar{\tau}] \rightarrow\ \rho_n) \\
f_n' & = \Lambda\bar{\alpha}.\lambda\bar{v}.\Lambda\bar{\rho}.\lambda\overline{c_1}\ \ldots\lambda\overline{c_n}.\ cata^{T_n}\ \bar{\tau}\ \bar{\rho}\ \overline{c_1}\ \ldots\ \overline{c_n}\ e_n
\end{array}
$$

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 $\overline{c_1}\ldots\overline{c_n}$ 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.

5.2.5 Catify

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

$$
\begin{array}{lll}
f_1 & :: \forall\bar{\alpha}.T_1\,\bar{\tau} \to \bar{\sigma} & \\
f_1 & = \Lambda\bar{\alpha}.\lambda\, t\, \overline{v_1}.e_1 & \\
\vdots & & \\
f_n & :: \forall\bar{\alpha}.T_n\,\bar{\tau} \to \bar{\sigma} & \\
f_n & = \Lambda\bar{\alpha}.\lambda\, t\, \overline{v_n}.e_n & \\
\Longrightarrow & & \\
 & \text{— The wrappers} & \\
f_1 & :: \forall\bar{\alpha}.T_1\,\bar{\tau} \to \bar{\sigma} & \\
f_1 & = \Lambda\bar{\alpha}.\lambda\, t\, \overline{v_1}.\ cata^{T_1}\,\bar{\tau} & (T_1\,\bar{\tau})\ \ldots\ (T_n\,\bar{\tau}) \quad \text{— Type arguments} \\
 & & (f^{T_1}_{C_1}\,\bar{\alpha})\ \ldots\ (f^{T_1}_{C_m}\,\bar{\alpha}) \quad \text{— Constructors of } T_1 \\
 & & \vdots \\
 & & (f^{T_n}_{C_1}\,\bar{\alpha})\ \ldots\ (f^{T_n}_{C_m}\,\bar{\alpha}) \quad \text{— Constructors of } T_n \\
 & & t \\
f_n & :: \forall\bar{\alpha}.T_n\,\bar{\tau} \to \bar{\sigma} & \\
f_n & = \Lambda\bar{\alpha}.\lambda\, t\, \overline{v_n}.\ cata^{T_n}\,\bar{\tau} & (T_1\,\bar{\tau})\ \ldots\ (T_n\,\bar{\tau}) \quad \text{— Type arguments} \\
 & & (f^{T_1}_{C_1}\,\bar{\alpha})\ \ldots\ (f^{T_1}_{C_m}\,\bar{\alpha}) \quad \text{— Constructors of } T_1 \\
 & & \vdots \\
 & & (f^{T_n}_{C_1}\,\bar{\alpha})\ \ldots\ (f^{T_n}_{C_m}\,\bar{\alpha}) \quad \text{— Constructors of } T_n \\
 & & t \\
 & \text{— The workers} & \\
f^{T_1}_{C_1} & = \Lambda\,\bar{\alpha}.\lambda\,\overline{z_1}.\lambda\bar{v}.\ (\lambda\, t.e_1)\,(C_1\,\bar{\tau}\,\overline{t_m}) & \text{— First constructor of } T_1 \\
\ldots & & \\
f^{T_1}_{C_m} & = \Lambda\,\bar{\alpha}.\lambda\,\overline{z_m}.\lambda\bar{v}.\ (\lambda\, t.e_1)\,(C_m\,\bar{\tau}\,\overline{t_m}) & \text{— Last constructor of } T_1 \\
\vdots & & \\
f^{T_n}_{C_1} & = \Lambda\,\bar{\alpha}.\lambda\,\overline{z_1}.\lambda\bar{v}.\ (\lambda\, t.e_n)\,(C_1\,\bar{\tau}\,\overline{t_m}) & \text{— First constructor of } T_n \\
\ldots & & \\
f^{T_n}_{C_m} & = \Lambda\,\bar{\alpha}.\lambda\,\overline{z_m}.\lambda\bar{v}.\ (\lambda\, t.e_n)\,(C_m\,\bar{\tau}\,\overline{t_m}) & \text{— Last constructor of } T_n
\end{array}
$$

   The comments we made when we first gave this algorithm also apply here (see page 60.) There are as many workers $f^{T_i}_{C_j}$ as constructors the of entire mutually recursive group and as many type arguments $T_i\,\bar{\tau}$ 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 $\bar{f}$ 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 $\bar{f}$. If no t's occur (the bindings are closed) we succeeded in transforming $\bar{f}$ to an explicit catamorphic form, so replace the original definition with the result of the previous step.

We continue our example with the previously buildified 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:

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

and four workers:

$$
map\#_{T_1}^{\{T,K\},T} = \Lambda\,\alpha\beta.\lambda\,z_1.\lambda\,f.\Lambda\,\tau\,\rho.\lambda\,t_1\;t_2\;k_1\;k_2.
$$

$$
\begin{array}{ll}
 & \textbf{let} \\
 & \quad l' \;=\; \lambda\, t.\, \textbf{case}\ t\ \textbf{of} \\
 & \qquad\qquad\quad T1\ a \quad\ \rightarrow\ t_1\ (f\ a) \\
 & \qquad\qquad\quad T2\ a\ k\ \rightarrow\ t_2\ (f\ a)\ (map\#^{\{T,K\},K}\ \alpha\ \beta\ k\ f\ \beta\ \tau\ \rho \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad t_1\ t_2\ k_1\ k_2) \\
 & \textbf{in} \\
 & l'\ (T1\ ta) \\
map\#^{\{T,K\},T}_{T_2} & = \Lambda\, \alpha\beta.\lambda\, z_1\ z_2.\lambda\, f.\Lambda\, \tau\ \rho.\lambda\, t_1\ t_2\ k_1\ k_2. \\
 & \textbf{let} \\
 & \quad l' \;=\; \lambda\, t.\, \textbf{case}\ t\ \textbf{of} \\
 & \qquad\qquad\quad T1\ a \quad\ \rightarrow\ t_1\ (f\ a) \\
 & \qquad\qquad\quad T2\ a\ k\ \rightarrow\ t_2\ (f\ a)\ (map\#^{\{T,K\},K}\ \alpha\ \beta\ k\ f\ \beta\ \tau\ \rho \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad t_1\ t_2\ k_1\ k_2) \\
 & \textbf{in} \\
 & l'\ (T2\ ta\ tk) \\
map\#^{\{T,K\},K}_{K_1} & = \Lambda\, \alpha\beta.\lambda\, f.\Lambda\, \tau\ \rho.\lambda\, t_1\ t_2\ k_1\ k_2. \\
 & \textbf{let} \\
 & \quad l' \;=\; \lambda\, k.\, \textbf{case}\ k\ \textbf{of} \\
 & \qquad\qquad\quad K1 \quad \rightarrow\ k_1 \\
 & \qquad\qquad\quad K2\ t\ \rightarrow\ k_2\ (map\#^{\{T,K\},T}\ \alpha\ \beta\ t\ f\ \beta\ \tau\ \rho\ t_1\ t_2\ k_1\ k_2) \\
 & \textbf{in} \\
 & l'\ K1 \\
map\#^{\{T,K\},K}_{K_2} & = \Lambda\, \alpha\beta.\lambda\, z_1.\lambda\, f.\Lambda\, \tau\ \rho.\lambda\, t_1\ t_2\ k_1\ k_2. \\
 & \textbf{let} \\
 & \quad l' \;=\; \lambda\, k.\, \textbf{case}\ k\ \textbf{of} \\
 & \qquad\qquad\quad K1 \quad \rightarrow\ k_1 \\
 & \qquad\qquad\quad K2\ t\ \rightarrow\ k_2\ (map\#^{\{T,K\},T}\ \alpha\ \beta\ t\ f\ \beta\ \tau\ \rho\ t_1\ t_2\ k_1\ k_2) \\
 & \textbf{in} \\
 & l'\ (K2\ tt)
\end{array}
$$

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

$$
\begin{array}{ll}
map\#^{\{T,K\},T}_{T_1} & = \Lambda\, \alpha\beta.\lambda\, z_1.\lambda\, f.\Lambda\, \tau\ \rho.\lambda\, t_1\ t_2\ k_1\ k_2.\ t_1\ (f\ ta) \\
map\#^{\{T,K\},T}_{T_2} & = \Lambda\, \alpha\beta.\lambda\, z_1\ z_2.\lambda\, f.\Lambda\, \tau\ \rho.\lambda\, t_1\ t_2\ k_1\ k_2. \\
 & t_2\ (f\ ta)\ (map\#^{\{T,K\},K}\ \alpha\ \beta\ tk\ f\ \beta\ \tau\ \rho\ t_1\ t_2\ k_1\ k_2) \\
map\#^{\{T,K\},K}_{K_1} & = \Lambda\, \alpha\beta.\lambda\, f.\Lambda\, \tau\ \rho.\lambda\, t_1\ t_2\ k_1\ k_2.k_1 \\
map\#^{\{T,K\},K}_{K_2} & = \Lambda\, \alpha\beta.\lambda\, z_1.\lambda\, f.\Lambda\, \tau\ \rho.\lambda\, t_1\ t_2\ k_1\ k_2. \\
 & k_2\ (map\#^{\{T,K\},T}\ \alpha\ \beta\ tt\ f\ \beta\ \tau\ \rho\ t_1\ t_2\ k_1\ k_2)
\end{array}
$$

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

$$
\begin{aligned}
map\#^{\{T,K\},T} &= \Lambda\, \alpha\beta.\lambda\, t\, f.\Lambda\, \tau\, \rho.\lambda\, t_1\, t_2\, k_1\, k_2.\\
&\quad cata^{\{T,K\},T}\, \beta\, \tau\, \rho\\
&\qquad (map\#^{\{T,K\},T}_{T_1}\, \beta)\, (map\#^{\{T,K\},T}_{T_2}\, \beta)\\
&\qquad (map\#^{\{T,K\},K}_{K_1}\, \beta)\, (map\#^{\{T,K\},K}_{K_2}\, \beta)\\
&\qquad t\, f\, \tau\, \rho\, t_1\, t_2\, k_1\, k_2\\
map\#^{\{T,K\},K} &= \Lambda\, \alpha\beta.\lambda\, k\, f.\Lambda\, \tau\, \rho.\lambda\, t_1\, t_2\, k_1\, k_2.\\
&\quad cata^{\{T,K\},K}\, \beta\, \tau\, \rho\\
&\qquad (map\#^{\{T,K\},T}_{T_1}\, \beta)\, (map\#^{\{T,K\},T}_{T_2}\, \beta)\\
&\qquad (map\#^{\{T,K\},K}_{K_1}\, \beta)\, (map\#^{\{T,K\},K}_{K_2}\, \beta)\\
&\qquad k\, f\, \tau\, \rho\, t_1\, t_2\, k_1\, k_2\\
map\#^{\{T,K\},T}_{T_1} &= \Lambda\, \alpha\beta.\lambda\, z_1.\lambda\, f.\Lambda\, \tau\, \rho.\lambda\, t_1\, t_2\, k_1\, k_2.\ t_1\, (f\ z_1)\\
map\#^{\{T,K\},T}_{T_2} &= \Lambda\, \alpha\beta.\lambda\, z_1\, z_2.\lambda\, f.\Lambda\, \tau\, \rho.\lambda\, t_1\, t_2\, k_1\, k_2.\ t_2\, (f\ z_1)\, (z_2\, f\, \beta\, \tau\, \rho\, t_1\, t_2\, k_1\, k_2)\\
map\#^{\{T,K\},K}_{K_1} &= \Lambda\, \alpha\beta.\lambda\, f.\Lambda\, \tau\, \rho.\lambda\, t_1\, t_2\, k_1\, k_2.k_1\\
map\#^{\{T,K\},K}_{K_2} &= \Lambda\, \alpha\beta.\lambda\, z_1.\lambda\, f.\Lambda\, \tau\, \rho.\lambda\, t_1\, t_2\, k_1\, k_2.\ k_2\, (z_1\, f\, \beta\, \tau\, \rho\, t_1\, t_2\, k_1\, k_2)
\end{aligned}
$$

All the pre-recursion variables and recursive calls have been eliminated, the transformation is successful, therefore we replace the original definition of $map\#^{\{T,K\},T}$ and $map\#^{\{T,K\},K}$ 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, $\overline{t_i}$, appropriately typed — with respect to the constructor they belong to — variables, and $\overline{z_i}$ fresh,* *appropriately typed variables, we define the rules of the rewrite system to be:*

$$\mathcal{R} = \{E^{\bar{T},T_i}\ \bar{g}\ \overline{t_i} \rightarrow E^{\bar{T},T_i}\ id\ \overline{z_i} \mid i \in \texttt{Constrs}(T_j) \wedge j \in \{\overline{T\bar{\alpha}}\}\}$$

This set of rewrite rules are valid only *in the body of the function being transformed. Typing of $\overline{t_i}$ and $\overline{z_i}$ 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. $\bar{g}$ 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 $\bar{g}$. Note that, j varies over type constructors (members of a possibly mutually recursive group $\overline{T\ \bar{\alpha}}$), 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.*□

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.

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 τ_{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 σ, 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.*$\square$

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.*$\square$

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):

$$
\begin{array}{ll}
f & = \Lambda\bar{\alpha}.\lambda\bar{v}.body \\
\Longrightarrow & \\
& \langle v' \text{ denotes the better ordering of arguments}\rangle \\
f & = \Lambda\bar{\alpha}.\lambda\bar{v}.f'\ \bar{\alpha}\ \overline{v'} \\
f' & = \Lambda\bar{\alpha}.\lambda\overline{v'}.body
\end{array}
$$

Lets see an example for the transformation:

$$
\begin{array}{ll}
mapFilter & :: \forall\alpha\beta.(\alpha \rightarrow\ Bool) \ \rightarrow [\alpha]\ \rightarrow (\alpha \rightarrow \beta)\ \rightarrow [\beta] \\
mapFilter & = \Lambda\alpha\beta.\lambda\ p\ xs\ f. \\
& \quad \textbf{case}\ xs\ \textbf{of} \\
& \qquad [] \quad \rightarrow []\ \beta \\
& \qquad y : ys \ \rightarrow \textbf{case}\ p\ y\ \textbf{of} \\
& \qquad\qquad True \ \rightarrow\ (:)\ \beta\ (f\ y)\ (mapFilter\ \alpha\ \beta\ p\ ys\ f) \\
& \qquad\qquad False \ \rightarrow\ mapFilter\ \alpha\ \beta\ p\ ys\ f
\end{array}
$$

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

$$
\begin{array}{ll}
mapFilter^* & :: \forall\alpha\beta.(\alpha \rightarrow\ Bool)\ \rightarrow [\alpha]\ \rightarrow (\alpha \rightarrow \beta)\ \rightarrow [\beta] \\
mapFilter^* & = \Lambda\alpha\beta.\lambda\ p\ xs\ f.mapFilter'\ \alpha\ \beta\ xs\ p\ f \\
mapFilter' & :: \forall\alpha\beta.[\alpha]\ \rightarrow (\alpha \rightarrow\ Bool)\ \rightarrow (\alpha \rightarrow \beta)\ \rightarrow [\beta] \\
mapFilter' & = \Lambda\alpha\beta.\lambda\ xs\ p\ f.
\end{array}
$$

$$
\begin{array}{l}
\textbf{case}\ xs\ \textbf{of} \\
\quad [] \quad\quad \to [\,]\ \beta \\
\quad y : ys \ \to \textbf{case}\ p\ y\ \textbf{of} \\
\quad\quad\quad\quad\quad \mathit{True} \ \to (:)\ \beta\ (f\ y)\ (\mathit{mapFilter}\ \alpha\ \beta\ p\ ys\ f) \\
\quad\quad\quad\quad\quad \mathit{False} \ \to \mathit{mapFilter}\ \alpha\ \beta\ p\ ys\ f
\end{array}
$$

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:

$$
\begin{array}{ll}
\mathit{mapFilter}^* & :: \forall\alpha\beta.(\alpha \to \ \mathit{Bool}) \ \to [\alpha] \ \to (\alpha \to \beta) \ \to [\beta] \\
\mathit{mapFilter}^* & = \Lambda\alpha\beta.\lambda\ p\ xs\ f.\mathit{mapFilter}'\ \alpha\ \beta\ xs\ p\ f \\
\mathit{mapFilter}' & :: \forall\alpha\beta.[\alpha] \ \to (\alpha \to \ \mathit{Bool}) \ \to (\alpha \to \beta) \ \to [\beta] \\
\mathit{mapFilter}' & = \Lambda\alpha\beta.\lambda\ xs\ p\ f. \\
& \quad \textbf{case}\ xs\ \textbf{of} \\
& \quad\quad [] \quad\quad \to [\,]\ \beta \\
& \quad\quad y : ys \ \to \textbf{case}\ p\ y\ \textbf{of} \\
& \quad\quad\quad\quad\quad\quad \mathit{True} \ \to (:)\ \beta\ (f\ y)\ (\mathit{mapFilter}'\ \alpha\ \beta\ ys\ p\ f) \\
& \quad\quad\quad\quad\quad\quad \mathit{False} \ \to \mathit{mapFilter}'\ \alpha\ \beta\ ys\ p\ f
\end{array}
$$

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:

$$
\begin{array}{ll}
\textbf{module}\ \mathit{Set} & (\mathit{Set},\ \mathit{empty},\ \mathit{insert},\ \mathit{isEmpty})\ \textbf{where} \\
\textbf{data}\ \mathit{Set}\ a & = \mathit{EmptySet} \mid \mathit{Insert}\ a\ (\mathit{Set}\ a) \\
 & \langle\text{Implementation based on lists}\rangle \\
\mathit{empty} & :: \mathit{Set}\ a \\
\mathit{empty} & = \mathit{EmptySet} \\
\mathit{insert} & :: a \rightarrow \mathit{Set}\ a \rightarrow \mathit{Set}\ a \\
\mathit{insert}\ x\ s & = \mathit{Insert}\ x\ s \\
 & \langle\text{more functions (destructors and predicates) on Sets}\rangle \\
\mathit{isEmpty} & :: \mathit{Set}\ a - > \mathit{Bool}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{isEmpty}\ \mathit{EmptySet} & = \mathit{True} \\
\mathit{isEmpty}\ \_ & = \mathit{False}
\end{array}
$$

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 $\mathit{isEmpty}\ (\mathit{insert}\ 1 \mathit{empty})$ 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 $\mathit{cata}^{\mathit{Set}}$ and $\mathit{build}^{\mathit{Set}}$ and the wrappers of *insert*, *empty*, *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 $\mathit{cata}^{\mathit{Set}}$, $\mathit{build}^{\mathit{Set}}$ is exported. To make this argument more concrete consider the following example.

Given the module *Set*, fusion transformation would derive the following functions:

$$
\begin{array}{rcl}
\mathit{cata}^{\mathit{Set}} & :: & \forall \alpha\rho.\rho \to (\alpha \to \rho \to \rho) \to \mathit{Set}\ \alpha \to \rho \\
\mathit{build}^{\mathit{Set}} & :: & \forall \alpha.\forall \rho.(\rho \to (\alpha \to \rho \to \rho) \to \rho) \to \mathit{Set}\ \alpha
\end{array} \tag{5.2}
$$

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

$$
\begin{array}{ll}
\mathbf{module}\ \mathit{Set} & (\mathit{Set},\ \mathit{empty},\ \mathit{insert},\ \mathit{isEmpty})\ \mathbf{where} \\
\mathbf{data}\ \mathit{Set}\ a & = \mathit{EmptySet} \mid \mathit{Insert}\ a\ (\mathit{Set}\ a)\ (\mathit{Set}\ a) \quad \text{— !!! CHANGED !!!} \\
 & \langle \text{Implementation based on trees} \rangle \\
\mathit{empty} & :: \mathit{Set}\ a \\
\mathit{empty} & = \mathit{EmptySet} \\
\mathit{insert} & :: a \to \mathit{Set}\ a \to \mathit{Set}\ a \\
\mathit{insert}\ x\ s & = \langle \text{insertion to a balanced Tree} \rangle \\
 & \langle \text{more functions (destructors and predicates) on Sets} \rangle \\
\mathit{isEmpty} & :: \mathit{Set}\ a -> \mathit{Bool} \\
\mathit{isEmpty}\ \mathit{EmptySet} & = \mathit{True} \\
\mathit{isEmpty}\ \_ & = \mathit{False}
\end{array}
$$

The interface of the module is not changed, so one could reasonably expect that modules importing *Set* need not be recompiled. This is not the case, if $build^{Set}$ and $cata^{Set}$ is 'silently' exported to expose opportunities of fusion, because the type of these functions change to:

$$\begin{aligned} cata^{Set} &:: \forall\alpha\rho.\rho \to (\alpha \to \rho \to \rho \to \rho) \to Set\ \alpha \to \rho \\ build^{Set} &:: \forall\alpha.\forall\rho.(\rho \to (\alpha \to \rho \to \rho \to \rho) \to \rho) \to Set\ \alpha \end{aligned} \tag{5.3}$$

Since the *Set* is exported abstractly we are not expecting recompilation of modules which import *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:* $cata^T$, $build^T$ 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:* $cata^T$, $build^T$ 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 *one cons cell* is used for each element in the result, in other words the translation scheme is such that *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.

$$
\begin{array}{lll}
[\, e \mid\,] & = [e] \\
[\, e \mid b,\ Q\,] & = \textbf{if}\ b\ \textbf{then}\ [\, e \mid Q\,]\ \textbf{else}\ [] \\
[\, e \mid p \leftarrow l,\ Q\,] & = \textbf{let} \\
 & \qquad \mathit{ok}\ p\ = [\, e \mid Q\,] \\
 & \qquad \mathit{ok}\ \_\ = [] \\
 & \quad \textbf{in} \\
 & \quad \mathit{concatMap}\ \mathit{ok}\ l \\
[\, e \mid \textbf{let}\ \mathit{decls},\ Q\,] & = \textbf{let}\ \mathit{decls}\ \textbf{in}\ [\, e \mid Q\,]
\end{array}
$$

Figure 5.2 Semantics of Haskell list comprehensions

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:

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

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 *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 $\mathcal{TE}$ or $\mathcal{TQ}$ in front of untranslated subexpressions. Also for convenience we do not

$$
\begin{array}{lll}
\mathcal{TE}\,[\![\, [E \mid QS] \,]\!] & \equiv & \mathcal{TQ}\,[\![\, [E \mid QS] \mathbin{+\!\!+} [\,] \,]\!] \\
\mathcal{TQ}\,[\![\, [E \mid\,] \mathbin{+\!\!+} L \,]\!] & \equiv & (:)\,\mathcal{TE}\,[\![\, E \,]\!]\,\mathcal{TE}\,[\![\, L \,]\!] \\
\mathcal{TQ}\,[\![\, [E \mid B,\ QS] \mathbin{+\!\!+} L \,]\!] & \equiv & \textbf{case}\,\mathcal{TE}\,[\![\, B \,]\!]\,\textbf{of} \\
 & & \quad \mathit{True} \ \rightarrow\ \mathcal{TQ}\,[\![\, [E \mid QS] \mathbin{+\!\!+} L \,]\!] \\
 & & \quad \mathit{False} \ \rightarrow\ \mathcal{TQ}\,[\![\, L \,]\!] \\
\mathcal{TQ}\,[\![\, [E \mid P \ \leftarrow\ L_1,\ QS] \mathbin{+\!\!+} L_2 \,]\!] & \equiv & \textbf{let} \\
 & & \quad h \ =\ \lambda\, us.\ \textbf{case}\ us\ \textbf{of} \\
 & & \qquad\qquad [\,] \ \rightarrow\ \mathcal{TE}\,[\![\, L_2 \,]\!] \\
 & & \qquad\qquad (:)\ u \\
 & & \qquad\qquad\quad us' \ \rightarrow\ \textbf{case}\ u\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad P \ \rightarrow\ \mathcal{TQ}\,[\![\, [E \mid QS] \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \mathbin{+\!\!+} \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (h\ us') \,]\!] \\
 & & \qquad\qquad\qquad\qquad \_ \ \rightarrow\ h\ us' \\
 & & \textbf{in} \\
 & & (h\,\mathcal{TE}\,[\![\, L_1 \,]\!]) \\
\mathcal{TQ}\,[\![\, [E \mid \textbf{let}\ DS,\ QS] \mathbin{+\!\!+} L \,]\!] & \equiv & \textbf{let} \\
 & & \quad \mathcal{TE}\,[\![\, DS \,]\!] \\
 & & \textbf{in} \\
 & & \mathcal{TQ}\,[\![\, [E \mid QS] \mathbin{+\!\!+} L \,]\!]
\end{array}
$$

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 $cata^{[\alpha]}$ and $build^{[\alpha]}$ with their expected types and definitions.

$$
\begin{array}{ll}
 & \mathcal{TE}\,[\![\, [\,E \mid QS\,] \,]\!] \\
 & \langle\text{introduce build}\rangle \\
= & \mathit{build}\ (\mathcal{TQ}\,[\![\, [\,E \mid QS\,] \,]\!]) \\
 & \langle\text{there are four subcases: we deal with them one by one}\rangle \\
\textbf{Case}\ 1. & \langle\text{empty generator}\rangle \\
 & \mathcal{TQ}\,[\![\, [\,E \mid\,] \,]\!]\ \mathit{nil\ cons} \\
= & \mathit{cata\ nil\ cons}\ [\,\mathcal{TE}\,[\![\, E \,]\!]\,] \\
 & \langle\text{definition of } [\mathit{expr}]\rangle \\
= & \mathit{cata\ nil\ cons}\ ((:)\,\mathcal{TE}\,[\![\, E \,]\!]\,[\,]) \\
 & \langle\text{cata of known constructor rule for } (:)\rangle \\
= & \mathit{cons}\ (\mathcal{TE}\,[\![\, E \,]\!])\ (\mathit{cata\ nil\ cons}\ [\,]) \\
 & \langle\text{cata of known constructor for } [\,]\rangle \\
= & \mathit{cons}\ (\mathcal{TE}\,[\![\, E \,]\!])\ \mathit{nil}. \\
\textbf{Case}\ 2. & \langle\text{filter}\rangle \\
 & \mathcal{TQ}\,[\![\, [\,E \mid B,\ QS\,] \,]\!]\ \mathit{nil\ cons}
\end{array}
$$

$$
\begin{array}{ll}
& \langle \text{definition} \rangle \\
= & \mathit{cata\ nil\ cons}\ (\textbf{if}\ \mathcal{TE}\ [\![\ B\ ]\!]\ \textbf{then}\ \mathcal{TE}\ [\![\ [\ E \mid QS\ ]\ ]\!]\ \textbf{else}\ [\,]) \\
& \langle \text{translation of } \textbf{if} \rangle \\
= & \mathit{cata\ nil\ cons}\ (\ \textbf{case}\ \mathcal{TE}\ [\![\ B\ ]\!]\ \textbf{of} \\
& \qquad\qquad\quad \mathit{True}\ \to\ \mathcal{TE}\ [\![\ [\ E \mid QS\ ]\ ]\!] \\
& \qquad\qquad\quad \mathit{False}\ \to\ [\,]) \\
& \langle \text{cata of case rule} \rangle \\
= & \textbf{case}\ \mathcal{TE}\ [\![\ B\ ]\!]\ \textbf{of} \\
& \quad \mathit{True}\ \to\ \mathit{cata\ nil\ cons}\ (\mathcal{TE}\ [\![\ [\ E \mid QS\ ]\ ]\!]) \\
& \quad \mathit{False}\ \to\ \mathit{cata\ nil\ cons}\ [\,] \\
& \langle \text{definition of } \mathcal{TE} \text{ and cata of known constructor} \rangle \\
= & \textbf{case}\ \mathcal{TE}\ [\![\ B\ ]\!]\ \textbf{of} \\
& \quad \mathit{True}\ \to\ \mathit{cata\ nil\ cons}\ (\mathit{build}\ (\mathcal{TQ}\ [\![\ [\ E \mid QS\ ]\ ]\!])) \\
& \quad \mathit{False}\ \to\ \mathit{nil} \\
& \langle \texttt{cata-build} \text{ rule} \rangle \\
= & \textbf{case}\ \mathcal{TE}\ [\![\ B\ ]\!]\ \textbf{of} \\
& \quad \mathit{True}\ \to\ \mathcal{TQ}\ [\![\ [\ E \mid QS\ ]\ ]\!]\ \mathit{nil\ cons} \\
& \quad \mathit{False}\ \to\ \mathit{nil}.
\end{array}
$$

Case 3. $\langle \text{generator} \rangle$

$$
\begin{array}{ll}
& \mathcal{TQ}\ [\![\ [\ E \mid P\ \leftarrow\ L,\ QS\ ]\ ]\!]\ \mathit{nil\ cons} \\
& \langle \text{definition} \rangle \\
= & \mathit{cata\ nil\ cons}\ (\ \textbf{let} \\
& \qquad\qquad\quad \mathit{ok}\ P'\ =\ \mathcal{TE}\ [\![\ [\ E \mid QS\ ]\ ]\!] \\
& \qquad\qquad\quad \mathit{ok}\ \_\ =\ [\,] \\
& \qquad\qquad \textbf{in} \\
& \qquad\qquad \mathit{concatMap\ ok}\ (\mathcal{TE}\ [\![\ L\ ]\!])) \\
& \langle \text{translation of pattern matching} \rangle \\
= & \mathit{cata\ nil\ cons}\ (\ \textbf{let} \\
& \qquad\qquad\quad \mathit{ok}\ =\ \lambda\, P'.\ \textbf{case}\ P'\ \textbf{of} \\
& \qquad\qquad\qquad\qquad P\ \to\ \mathcal{TE}\ [\![\ [\ E \mid QS\ ]\ ]\!] \\
& \qquad\qquad\qquad\qquad \_\ \to\ [\,] \\
& \qquad\qquad \textbf{in} \\
& \qquad\qquad \mathit{concatMap\ ok}\ (\mathcal{TE}\ [\![\ L\ ]\!])) \\
& \langle \mathit{ok} \text{ is buildified and we get} \rangle \\
= & \mathit{cata\ nil\ cons}\ (\ \textbf{let} \\
& \qquad\qquad\quad \mathit{ok}\ =\ \lambda\, P'.\mathit{build}\ (\lambda\, \mathit{nil}'\ \mathit{cons}'.\ \textbf{case}\ P'\ \textbf{of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad P\ \to\ \mathcal{TQ}\ [\![\ [\ E \mid QS\ ]\ ]\!]\ \mathit{nil}'\ \mathit{cons}' \\
& \qquad\qquad\qquad\qquad\qquad\qquad \_\ \to\ \mathit{nil}') \\
& \qquad\qquad \textbf{in} \\
& \qquad\qquad \mathit{concatMap\ ok}\ (\mathcal{TE}\ [\![\ L\ ]\!]))
\end{array}
$$

$$
\begin{array}{ll}
& \langle \text{cata of let} \rangle \\
= & \textbf{let} \\
& \quad ok \;=\; \lambda\, P'.build\; (\lambda\; nil'\; cons'.\; \textbf{case}\; P'\; \textbf{of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad P \;\rightarrow\; \mathcal{TQ}\, [\![\, [\, E \mid QS\,]\,]\!]\; nil'\; cons' \\
& \qquad\qquad\qquad\qquad\qquad\qquad \_ \;\rightarrow\; nil') \\
& \textbf{in} \\
& cata\; nil\; cons\; (concatMap\; ok\; (\mathcal{TE}\, [\![\, L\,]\!])) \\
& \langle \texttt{build-cata} \text{ form definition of concatMap} \rangle \\
= & \textbf{let} \\
& \quad ok \;=\; \lambda\, P'.build\; (\lambda\; nil'\; cons'.\; \textbf{case}\; P'\; \textbf{of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad P \;\rightarrow\; \mathcal{TQ}\, [\![\, [\, E \mid QS\,]\,]\!]\; nil'\; cons' \\
& \qquad\qquad\qquad\qquad\qquad\qquad \_ \;\rightarrow\; nil') \\
& \textbf{in} \\
& cata\; nil\; cons \\
& \qquad ((\lambda\, f\; xs.build\; (\lambda\; nil'\; cons'.cata\; cm_n\; cm_c\; xs\; f\; nil'\; append))\; ok\; (\mathcal{TE}\, [\![\, L\,]\!])) \\
& \langle \beta \text{ reductions} \rangle \\
= & \textbf{let} \\
& \quad ok \;=\; \lambda\, P'.build\; (\lambda\; nil'\; cons'.\; \textbf{case}\; P'\; \textbf{of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad P \;\rightarrow\; \mathcal{TQ}\, [\![\, [\, E \mid QS\,]\,]\!]\; nil'\; cons' \\
& \qquad\qquad\qquad\qquad\qquad\qquad \_ \;\rightarrow\; nil') \\
& \textbf{in} \\
& cata\; nil\; cons\; (build\; (\lambda\; nil'\; cons'.cata\; cm_n\; cm_c\; (\mathcal{TE}\, [\![\, L\,]\!])\; ok\; nil'\; append)) \\
& \langle \texttt{cata-build} \text{ rule} \rangle \\
= & \textbf{let} \\
& \quad ok \;=\; \lambda\, P'.build\; (\lambda\; nil'\; cons'.\; \textbf{case}\; P'\; \textbf{of} \\
& \qquad\qquad\qquad\qquad\qquad\qquad P \;\rightarrow\; \mathcal{TQ}\, [\![\, [\, E \mid QS\,]\,]\!]\; nil'\; cons' \\
& \qquad\qquad\qquad\qquad\qquad\qquad \_ \;\rightarrow\; nil') \\
& \textbf{in} \\
& cata\; cm_n\; cm_c\; (\mathcal{TE}\, [\![\, L\,]\!])\; ok\; nil\; append.
\end{array}
$$

Case 4. $\langle$**let**$\rangle$

$$
\begin{array}{ll}
& \mathcal{TQ}\, [\![\, [\, E \mid \textbf{let}\; DS,\; QS\,]\,]\!]\; nil\; cons \\
& \langle \text{definition} \rangle \\
= & cata\; nil\; cons\; (\; \textbf{let} \\
& \qquad\qquad\qquad\quad DS \\
& \qquad\qquad\qquad \textbf{in} \\
& \qquad\qquad\qquad \mathcal{TE}\, [\![\, [\, E \mid QS\,]\,]\!]) \\
& \langle \text{case of let} \rangle
\end{array}
$$

$$
\begin{array}{ll}
= & \textbf{let} \\
 & \quad DS \\
 & \textbf{in} \\
 & cata\ nil\ cons\ (\mathcal{TE}\ [\![\ [\ E \mid QS\ ]\ ]\!]) \\
 & \langle \texttt{cata-build}\ \text{rule} \rangle \\
= & \textbf{let} \\
 & \quad DS \\
 & \textbf{in} \\
 & \mathcal{TQ}\ [\![\ [\ E \mid QS\ ]\ ]\!]\ nil\ cons.
\end{array}
$$

Cases 1, 2 and 4 are clearly optimal: only one list is built with the abstracted constructors *nil* and *cons*. If a good consumer (a function which consumes its argument with a *cata*) is applied to the resulting list the `cata-build` rule applies the intermediate list is not built. Case 3 is somewhat subtle. In particular, the presence of *append* is worrying. Consider however, that L is a piece of source program and as such is always finite. Its translation proceeds via the $\mathcal{TE}$ 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 *build* (see the definition of $\mathcal{TE}$ on Page 93). So the translation of Case 3, produces a chain of applications of catas to build. This chain is then reduced via the `cata-build` rule.

To see one example that *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.

$$
\begin{array}{l}
\quad \mathcal{TE}\ [\![\ [\ x \mid x\ \leftarrow\ [1,\ 2]\ ]\ ]\!] \\
\quad \langle \text{definition} \rangle \\
= build\ (\lambda\ nil\ cons.\mathcal{TQ}\ [\![\ [\ x \mid x\ \leftarrow\ [1,\ 2]\ ]\ ]\!]) \\
\quad \langle \text{generator} \rangle \\
= build\ (\lambda\ nil\ cons.\ \textbf{let} \\
\qquad\qquad\qquad ok\ =\ \lambda\ x'.build\ (\lambda\ nil'\ cons'.\ \textbf{case}\ x'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad x\ \rightarrow\ \mathcal{TQ}\ [\![\ [\ x \mid\ ]\ ]\!]\ nil'\ cons' \\
\qquad\qquad\qquad\qquad\qquad\qquad \_\ \rightarrow\ nil') \\
\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\quad cata\ cm_n\ cm_c\ (\mathcal{TE}\ [\![\ [1,\ 2]\ ]\!])\ ok\ nil\ append) \\
\quad \langle \text{definition of } \mathcal{TQ}[\![[x|]]\!], \text{ and the variable } x \text{ always matches in the case} \rangle \\
= build\ (\lambda\ nil\ cons.\ \textbf{let} \\
\qquad\qquad\qquad ok\ =\ \lambda\ x'.build\ (\lambda\ nil'\ cons'.cons'\ x\ nil') \\
\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\quad cata\ cm_n\ cm_c\ (\mathcal{TE}\ [\![\ [1,\ 2]\ ]\!])\ ok\ nil\ append) \\
\quad \langle \text{skipping several steps: } \mathcal{TE}[\![[1,2]]\!] \text{ gives} \rangle
\end{array}
$$

$$
\begin{aligned}
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\ \mathbf{let}\\
&\qquad\qquad \mathit{ok}\ =\ \lambda\, x'.\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ x\ \mathit{nil}')\\
&\qquad\quad \mathbf{in}\\
&\qquad\quad \mathit{cata}\ \mathit{cm}_n\ \mathit{cm}_c\ (\mathit{build}\ (\lambda\ n\ c.\ c\ 1\ (c\ 2\ n)))\ \mathit{ok}\ \mathit{nil}\ \mathit{append})\\
&\langle \texttt{cata-build}\text{ rule and beta reductions}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\ \mathbf{let}\\
&\qquad\qquad \mathit{ok}\ =\ \lambda\, x'.\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ x\ \mathit{nil}')\\
&\qquad\quad \mathbf{in}\\
&\qquad\quad \mathit{cm}_c\ 1\ (\mathit{cm}_c\ 2\ \mathit{cm}_n)\ \mathit{ok}\ \mathit{nil}\ \mathit{append})\\
&\langle\text{defintion: } \mathit{cm}_c = \lambda z\ \ zs\ \ f\ \ n\ \ c.c(f\ \ z)(zs\ \ f\ \ n\ \ c)\text{ and beta reductions}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\ \mathbf{let}\\
&\qquad\qquad \mathit{ok}\ =\ \lambda\, x'.\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ x\ \mathit{nil}')\\
&\qquad\quad \mathbf{in}\\
&\qquad\quad \mathit{append}\ (\mathit{ok}\ 1)\ (\mathit{cm}_c\ 2\ \mathit{cm}_n\ \mathit{ok}\ \mathit{nil}\ \mathit{append})\\
&\langle\text{defintion: } \mathit{cm}_c = \lambda z\ \ zs\ \ f\ \ n\ \ c.c(f\ \ z)(zs\ \ f\ \ n\ \ c)\text{ again and beta reductions}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\ \mathbf{let}\\
&\qquad\qquad \mathit{ok}\ =\ \lambda\, x'.\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ x\ \mathit{nil}')\\
&\qquad\quad \mathbf{in}\\
&\qquad\quad \mathit{append}\ (\mathit{ok}\ 1)\ (\mathit{append}\ (\mathit{ok}\ 2)\ (\mathit{cm}_n\ \mathit{ok}\ \mathit{nil}\ \mathit{append}))\\
&\langle\text{definition: } \mathit{cm}_n = \lambda f\ \ n\ \ c.n\text{ and beta reductions}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\ \mathbf{let}\\
&\qquad\qquad \mathit{ok}\ =\ \lambda\, x'.\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ x\ \mathit{nil}')\\
&\qquad\quad \mathbf{in}\\
&\qquad\quad \mathit{append}\ (\mathit{ok}\ 1)\ (\mathit{append}\ (\mathit{ok}\ 2)\ \mathit{nil})\\
&\langle\text{definition of append for empty list}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\ \mathbf{let}\\
&\qquad\qquad \mathit{ok}\ =\ \lambda\, x'.\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ x\ \mathit{nil}')\\
&\qquad\quad \mathbf{in}\\
&\qquad\quad \mathit{append}\ (\mathit{ok}\ 1)\ (\mathit{ok}\ 2)\\
&\langle\text{definition: } \mathit{append} = \lambda xs\ \ ys\ \ n\ \ c.\mathit{cata}\ \ (\mathit{cata}\ \ n\ \ c\ \ ys)\ \ c\ \ xs\text{ and beta reductions}\rangle\\
&\langle\text{definition of } \mathit{ok}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\mathit{cata}\ (\mathit{cata}\ \mathit{nil}\ \mathit{cons}\ (\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ 2\ \mathit{nil}')))\ \mathit{cons}\ (\mathit{ok}\ 1))\\
&\langle \texttt{cata-build}\text{ and the definition of } \mathit{ok}\text{ again}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\mathit{cata}\ (\mathit{cons}\ 2\ \mathit{nil})\ \mathit{cons}\ (\mathit{build}\ (\lambda\ \mathit{nil}'\ \mathit{cons}'.\mathit{cons}'\ 1\ \mathit{nil}')))\\
&\langle \texttt{cata-build}\text{ rule}\rangle\\
=&\mathit{build}\ (\lambda\ \mathit{nil}\ \mathit{cons}.\mathit{cons}\ 1\ (\mathit{cons}\ 2\ \mathit{nil})).
\end{aligned}
$$

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-

$$
\begin{array}{ll}
\mathcal{TE}\,[\![\, [E \mid QS] \,]\!] & \equiv build\,(\mathcal{TQ}\,[\![\, [E \mid QS] \,]\!]) \\
\mathcal{TQ}\,[\![\, [E \mid\,] \,]\!]\; n\; c & \equiv c\,(\mathcal{TE}\,[\![\, E \,]\!])\; n \\
\mathcal{TQ}\,[\![\, [E \mid B,\; QS] \,]\!]\; n\; c & \equiv \mathbf{case}\; \mathcal{TE}\,[\![\, B \,]\!]\; \mathbf{of} \\
& \quad True \;\rightarrow\; \mathcal{TQ}\,[\![\, [E \mid QS] \,]\!]\; n\; c \\
& \quad False \;\rightarrow\; n \\
\mathcal{TQ}\,[\![\, [E \mid P \leftarrow L,\; QS] \,]\!]\; n\; c & \equiv \mathbf{let} \\
& \quad ok \;=\; \lambda\, P'.build\,(\lambda\, n'\; c'.\; \mathbf{case}\; P'\; \mathbf{of} \\
& \qquad\qquad P \;\rightarrow\; \mathcal{TQ}\,[\![\, [E \mid QS] \,]\!]\; n'\; c' \\
& \qquad\qquad \_ \;\rightarrow\; n') \\
& \qquad \text{— } cm_n \text{ and } cm_c \text{ are part of the } \texttt{build-cata} \text{ form} \\
& \qquad \text{— of } concatMap \\
& \quad cm_n \;=\; \lambda\, f\; n\; c.n \\
& \quad cm_c \;=\; \lambda\, z\; zs\; f\; n\; c.\; c\; (f\; z)\; (zs\; f\; n\; c) \\
& \mathbf{in} \\
& cata\; cm_n\; cm_c\; (\mathcal{TE}\,[\![\, L \,]\!])\; ok\; n\; append \\
\mathcal{TQ}\,[\![\, [E \mid \mathbf{let}\; DS,\; QS] \,]\!]\; n\; c & \equiv \mathbf{let} \\
& \quad \mathcal{TE}\,[\![\, DS \,]\!] \\
& \mathbf{in} \\
& \mathcal{TQ}\,[\![\, [E \mid QS] \,]\!]\; n\; c
\end{array}
$$

Figure 5.4 Optimal list comprehension desugaring scheme

dious. The function definitions in `build-cata` form we implicitly used in the derivation for *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 `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.

- *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.
- `-O` optimised. Compilation takes visibly longer, but the code resulting code is definitely faster. This is the most frequently used level of optimisation.
- `-O2` 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 `-O`. 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.

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

```
Rec {
map_List :: (forall t_x2u5 t_x2u6.(t_x2u5 -> t_x2u6) -> List t_x2u5 -> List t_x2u6)
[NoDiscard]
map_List
  = \ @ 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)
                                 (map_List @ t_x2u2 @ t_x2uI f_x2uJ b_x2u3)
        }
end Rec }

Rec {
cata_List :: (forall t_x2us t_x2ur.t_x2ur -> (t_x2us -> t_x2ur -> t_x2ur) -> List t_x2us -> t_x2ur)
[NoDiscard] __S LLS
cata_List
  = \ @ 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)
                                             (cata_List @ t_x2uh @ t_x2uK nil_x2uL zczuzc_x2uM b_x2uj)
        }
end Rec }

Rec {
build_List :: (forall t_x2uw.(forall t_x2uv. t_x2uv -> (t_x2uw -> t_x2uv -> t_x2uv) -> t_x2uv)
             -> List t_x2uw)
[NoDiscard]
build_List
  = \ @ t_x2uw g_x2uu :: (forall t_x2uv.t_x2uv -> (t_x2uw -> t_x2uv -> t_x2uv) -> t_x2uv) ->
        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_x2uv.t_x2uv -> (t_x2uw -> t_x2uv -> t_x2uv) -> t_x2uv)}
                        cata_List @ t_x2us @ t_x2ur a_x2uQ b_x2uR (build_List @ 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)}
               cata_List @ t_x2uE @ t_x2uT nil_x2uU zczuzc_x2uV ($wNil @ t_x2uE)
               = nil_x2uU ;
```

```
"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 ($w:^: @ t_x2uE a_x2uD b_x2uG)
               = (zczuzc_x2uV a_x2uD (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 `wnmap`. `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 `wnmap` is the worker for `map` and all the wrappers have been eliminated.

```
Rec {
wnmap :: (forall a b.List a -> (a -> b)
          -> __u - (forall t_s2CW.t_s2CW -> __u - ((b -> t_s2CW -> t_s2CW) -> t_s2CW)))
__AL 4
wnmap
  = \ @ 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) (wnmap @ 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. `wnmap_List` is the worker of the generated map.

```
Rec {
wnmap_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
wnmap_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)
                        (wnmap_List @ t_x2u5 @ t_x2u6 b_x2u3 x_s2C9 @ t_s2CS c1_s2CT c2_s2CU)
        }
end Rec }

nmap_List :: (forall t_x2u5 t_x2u6.List t_x2u5 -> (t_x2u5 -> t_x2u6) -> List t_x2u6)
__AL 2
nmap_List
  = __inline_me (\ @ t_x2u5 @ t_x2u6 x_s2Cc :: (List t_x2u5) x_s2C9 :: (t_x2u5 -> t_x2u6) ->
                     build_List @ t_x2u6 (wnmap_List @ t_x2u5 @ t_x2u6 x_s2Cc x_s2C9))

map_List :: (forall t_x2u5 t_x2u6.(t_x2u5 -> t_x2u6) -> List t_x2u5 -> List t_x2u6)
```

```
[NoDiscard] __AL 2
map_List
  = __inline_me (\ @ t_x2u5 @ t_x2u6 x_s2C9 :: (t_x2u5 -> t_x2u6) x_s2Cc :: (List t_x2u5) ->
                    nmap_List @ t_x2u5 @ t_x2u6 x_s2Cc x_s2C9)
```

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)))
__AL 4
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)))
__AL 4
witerate
```

```
  = \ @ a f :: (a -> a) x :: a @ t_s2D0
      c1_s2D1 OneShot :: t_s2D0 c2_s2D2 OneShot :: (a -> t_s2D0 -> t_s2D0) ->
        c2_s2D2 x (witerate @ a f (f x) @ t_s2D0 c1_s2D1 c2_s2D2)
end Rec }
```

Finally, `main`. All the normalised wrappers and the build wrappers are inlined so only calls to the workers remain.

```
main :: (IO ())
[NoDiscard] __AL 1
main
  = __coerce (IO ())
    (\ s5 :: (State# RealWorld) ->
         case nfoldl'
                  @ Int
                  @ Integer
                  (build_List
                       @ Integer
                       (wnmap
                            @ Integer
                            @ Integer
                            (build_List
                                 @ Integer
                                 (wnmap
                                      @ Integer
                                      @ Integer
                                      (build_List
                                           @ Integer
                                           (wntake
                                                @ Integer
                                                (build_List
                                                     @ Integer
                                                     (witerate
                                                          @ Integer
                                                          (\ s_s2wb :: Integer ->
                                                               PrelNum.+1 s_s2wb lit_a1Yu)
                                                          lit_a1Yu))
                                                ($wI# 1000)))
                                      (\ s_s2w9 :: Integer -> PrelNum.*1 s_s2w9 lvl_s2AL)))
                            (\ s_s2w7 :: Integer -> PrelNum.+1 s_s2w7 lit_a1Yu)))
                  (\ n :: Int ds_d2tS :: Integer ->
                       case n of wild { I# x1 ->
                       let {
                         s_s2BM :: Int#
                         __AL 0
                         s_s2BM
                           = +# x1 1
                       } in  $wI# s_s2BM
                       })
                  ($wI# 0)
         of w { I# ww ->
         case PrelIO.$whPutStr
```

```
                  PrelHandle.stdout (PrelShow.$wshowSignedInt 0 ww ($w[] @ Char)) s5
         of wild { (# new_s, a1 #) ->
         case PrelIO.$whPutChar PrelHandle.stdout '
' new_s
         of wild { (# new_s, a1 #) ->
         (# new_s, $w() #)
         }
         }
         })
```

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 succeeds.

```
wnmap :: (forall a b.List a -> (a -> b)
          -> __u - (forall t_s2CW.t_s2CW -> __u - ((b -> t_s2CW -> t_s2CW) -> t_s2CW)))
__AL 4
wnmap
  = \ @ 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) ->
        __letrec {
          _sat_s37m :: (List a -> t_s2CW)
          __AL 1
          _sat_s37m
            = \ x_X2Cr :: (List a) ->
                  case x_X2Cr of wild_B1 {
                      Nil -> c1_s2CX; :^: x xs -> c2_s2CY (x_s2Co x) (_sat_s37m xs)
                  };
        } in  _sat_s37m x_s2Cr
```

Notice, that `wntake` 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.

```
wntake :: (forall a.List a -> Int
           -> __u - (forall t_s2D4.t_s2D4 -> __u - ((a -> t_s2D4 -> t_s2D4) -> t_s2D4)))
__AL 4
wntake
  = \ @ a x_s2CB :: (List a) x_s2Cz :: Int @ t_s2D4
      c1_s2D5 OneShot :: t_s2D4 c2_s2D6 OneShot :: (a -> t_s2D4 -> t_s2D4) ->
        __letrec {
          _sat_s37o :: (List a -> Int -> t_s2D4)
          __AL 2
          _sat_s37o
            = \ x_X2CB :: (List a) x_X2Cz :: Int ->
                  case x_X2Cz of wild_B1 { I# ds_d2nA ->
                  case ds_d2nA of ds_X2nA {
                      0 -> c1_s2D5;
                      __DEFAULT ->
                          case x_X2CB 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  _sat_s37o xs ($wI# 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)
          -> __u - (forall t_s2CW.t_s2CW -> __u - ((b -> t_s2CW -> t_s2CW) -> t_s2CW)))
__AL 4
wnmap
  = \ @ 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) ->
        cata_List @ a @ t_s2CW c1_s2CX
                  (\ r_s37y :: a r_s37A :: t_s2CW -> c2_s2CY (x_s2Co r_s37y) r_s37A)
                  x_s2Cr
```

The catamorphism for `wntake` is higher-order (see the second type argument to `cata_List`)!

```
wntake :: (forall a.List a -> Int
           -> __u - (forall t_s2D4.t_s2D4 -> __u - ((a -> t_s2D4 -> t_s2D4) -> t_s2D4)))
__AL 4
wntake
  = \ @ a x_s2CB :: (List a) x_s2Cz :: Int @ t_s2D4
      c1_s2D5 OneShot :: t_s2D4 c2_s2D6 OneShot :: (a -> t_s2D4 -> t_s2D4) ->
        cata_List @ a @ (Int -> t_s2D4)
                  (\ x_X2Cz :: Int -> case x_X2Cz of wild_B1 { I# ds_d2nA -> c1_s2D5 })
                  (\ r_s385 :: a r_s387 :: (Int -> t_s2D4) x_X2Cz :: Int ->
                       case x_X2Cz of wild_B1 { I# ds_d2nA ->
                       case ds_d2nA of ds_X2nA {
                           0 -> c1_s2D5;
                           __DEFAULT ->
                               case ># ds_X2nA 0 of wild_X2 {
                                   True -> c2_s2D6 r_s385 (let {s_s2BI :: Int#
                                                                __AL 0
```

```
                                                      s_s2BI = -# ds_X2nA 1
                                                  } in  r_s387 ($wI# s_s2BI));
                              False -> __coerce t_s2D4 (error @ (List a) lvl_s2AA)
                          }
                  })
              x_s2CB
              x_s2Cz
```

After catify, the already transformed workers `wnmap` and `wntake` have become non-recursive and therefore they could be inlined into `main`. After applying the `cata-build` rule we get:

```
$wmain :: (__u - (State# RealWorld -> (# State# RealWorld, () #)))
[NoDiscard] __AL 1 __S P
$wmain
  = \ w :: (State# RealWorld) ->
        case nfoldl'
                 @ Int
                 @ Integer
                 (witerate
                      @ Integer
                      (\ s3 :: Integer -> PrelNum.+1 s3 lit)
                      lit
                      @ (Int -> List Integer)
                      (\ x :: Int -> case x of wild { I# ds -> $wNil @ Integer })
                      (\ r :: Integer r1 :: (Int -> List Integer) x :: Int ->
                           case x of wild { I# ds ->
                           case ds of ds1 {
                               0 -> $wNil @ Integer;
                               __DEFAULT ->
                                   case ># ds1 0 of wild1 {
                                       True ->
                                           let {
                                             s3 :: Integer
                                             s3 =
                                                 PrelNum.+1 (PrelNum.*1 r lvl) lit } in
                                           let { s4 :: (List Integer)
                                                 s4 =
                                             let {
                                               s5 :: Int#
                                               s5
                                                 = -# ds1 1
                                             } in  r1 ($wI# s5)
                                           } in
                                           $w:^: @ Integer s3 s4;
                                       False -> error @ (List Integer) lvl1
                                   }
                           }
                           })
                      ($wI# 1000))
                 (\ n :: Int ds :: Integer ->
                      case n of wild { I# x1 ->
                      let {
```

```
                    s3 :: Int#
                    s3
                      = +# x1 1
                  } in  $wI# s3
                  })
             s1
        of w1 { I# ww ->
        case PrelIO.$whPutStr
                 PrelHandle.stdout (PrelShow.$wshowSignedInt 0 ww ($w[] @ Char)) w
        of wild { (# new_s, a1 #) ->
        case PrelIO.$whPutChar PrelHandle.stdout '
' new_s
        of wild1 { (# new_s1, a11 #) ->
        (# new_s1, $w() #)
        }
        }
        }
```

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 -Sstderr
unopt +RTS -Sstderr
unopt +RTS -Sstderr
    Alloc    Collect    Live    GC    GC     TOT     TOT  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)

          1 Mb 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:

```
angel 168 (haskell/andreas): opt +RTS -Sstderr
opt +RTS -Sstderr
opt +RTS -Sstderr
    Alloc    Collect    Live    GC    GC     TOT     TOT  Page Flts
    bytes     bytes     bytes  user  elap    user    elap
1000
   106912                      0.00  0.00

    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)
  Total 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
               c1_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)))
```

```
              x_s2Bx :: (a -> b)
              @ t_s2C4
              c1_s2C5 OneShot :: t_s2C4
              c2_s2C6 OneShot :: (b -> t_s2C4 -> t_s2C4) ->
                c2_s2C6 (x_s2Bx r_s2CC) (r_s2CE x_s2Bx @ t_s2C4 c1_s2C5 c2_s2C6))

wntake :: (forall a.List a -> Int
           -> __u - (forall t_s2Cc.t_s2Cc -> __u - ((a -> t_s2Cc -> t_s2Cc) -> t_s2Cc)))
__AL 4
wntake
  = \ @ a ->
        cata_List
            @ a
            @ (Int
               -> __u - (forall t_s2Cc.
                         t_s2Cc -> __u - ((a -> t_s2Cc -> t_s2Cc) -> t_s2Cc)))
            (\ x_s2BI :: Int
               @ t_s2Cc
               c1_s2Cd OneShot :: t_s2Cc
               c2_s2Ce OneShot :: (a -> t_s2Cc -> t_s2Cc) ->
                 case x_s2BI of wild_B1 { I# ds_d2nc ->
                 case ds_d2nc of ds_X2nc { 0 -> c1_s2Cd; __DEFAULT -> c1_s2Cd }
                 })
            (\ r_s2D9 :: a
               r_s2Db :: (Int
                          -> __u - (forall t_s2Cc.
                                    t_s2Cc -> __u - ((a -> t_s2Cc -> t_s2Cc) -> t_s2Cc)))
               x_s2BI :: Int
               @ t_s2Cc
               c1_s2Cd OneShot :: t_s2Cc
               c2_s2Ce OneShot :: (a -> t_s2Cc -> t_s2Cc) ->
                 case x_s2BI of wild_B1 { I# ds_d2nc ->
                 case ds_d2nc of ds_X2nc {
                     0 -> c1_s2Cd;
                     __DEFAULT ->
                         case ># ds_X2nc 0 of wild_X2 {
                             True ->
                                 c2_s2Ce
                                     r_s2D9
                                     (let {
                                        s_s2AQ :: Int#
                                        __AL 0
                                        s_s2AQ
                                          = -# ds_X2nc 1
                                      } in  r_s2Db ($wI# s_s2AQ) @ t_s2Cc c1_s2Cd c2_s2Ce);
                             False -> __coerce t_s2Cc (error @ (List a) lvl_s2zI)
                         }
                 }
                 })
```

Contrast these with the previously given definitions! The only difference is that now all the arguments to `wntake` and `wnmap` are passed around in the recursive call. `main` changes accordingly to:

```
$wmain :: (__u - (State# RealWorld -> (# State# RealWorld, () #)))
[NoDiscard] __AL 1 __S P
$wmain
  = \ w :: (State# RealWorld) ->
        case nfoldl'
                 @ Int
                 @ Integer
                 (witerate
                      @ Integer
                      (\ s7 :: Integer -> PrelNum.+1 s7 lit)
                      lit
                      @ (Int -> __u - (forall t. t -> __u - ((Integer -> t -> t) -> t)))
                      (\ x :: Int @ t c1 :: t c2 :: (Integer -> t -> t) ->
                           case x of wild { I# ds -> c1 })
                      (\ r :: Integer
                         r1 :: (Int
                                -> __u - (forall t. t -> __u - ((Integer -> t -> t) -> t)))
                         x :: Int
                         @ t
                         c1 :: t
                         c2 :: (Integer -> t -> t) ->
                           case x of wild { I# ds ->
                           case ds of ds1 {
                               0 -> c1;
                               __DEFAULT ->
                                   case ># ds1 0 of wild1 {
                                       True ->
                                           c2  r
                                               (let {
                                                  s7 :: Int#
                                                  s7
                                                    = -# ds1 1
                                                } in  r1 ($wI# s7) @ t c1 c2);
                                       False -> __coerce t (error @ (List Integer) lvl1)
                                   }
                           }
                           })
                      ($wI# 1000)
                      @ ((Integer -> Integer)
                         -> __u - (forall t. t -> __u - ((Integer -> t -> t) -> t)))
                      (s3 @ Integer @ Integer)
                      (s2 @ Integer @ Integer)
                      (\ s7 :: Integer -> PrelNum.*1 s7 lvl)
                      @ ((Integer -> Integer)
                         -> __u - (forall t. t -> __u - ((Integer -> t -> t) -> t)))
                      (s3 @ Integer @ Integer)
                      (s2 @ Integer @ Integer)
                      (\ s7 :: Integer -> PrelNum.+1 s7 lit)
                      @ (List Integer)
                      ($wNil @ Integer)
                      ($w:^: @ Integer))
```

```
                 (\ n :: Int ds :: Integer ->
                      case n of wild { I# x1 ->
                      let {
                        s7 :: Int#
                        s7
                          = +# x1 1
                      } in  $wI# s7
                      })
                 s5
        of w1 { I# ww ->
        case PrelIO.$whPutStr
                 PrelHandle.stdout (PrelShow.$wshowSignedInt 0 ww ($w[] @ Char)) w
        of wild { (# new_s, a1 #) ->
        case PrelIO.$whPutChar PrelHandle.stdout '
' new_s
        of wild1 { (# new_s1, a11 #) ->
        (# new_s1, $w() #)
        }
        }
        }
```

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 :: (__u - (State# RealWorld -> (# State# RealWorld, () #)))
[NoDiscard] __AL 1 __S P
$wmain
  = \ w :: (State# RealWorld) ->
        case PrelIO.$whPutStr
                 PrelHandle.stdout
                 (PrelNum.showSignedInteger
                      PrelBase.zeroInt
                      (witerate
                           @ Integer
                           (\ s3 :: Integer -> PrelNum.+1 s3 lit)
                           lit
                           @ (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))
```

```
                    ($w[] @ Char))
               w
        of wild { (# new_s, a1 #) ->
        case PrelIO.$whPutChar PrelHandle.stdout '
' new_s
        of wild1 { (# new_s1, a11 #) ->
        (# new_s1, $w() #)
        }
        }
```

Not a single list constructor remains: we managed to eliminate all the intermediate data structures. This is because `length` is now a catamorphism (GHC also reports that the `cata-build` rule has been applied 8 times), the intermediate list between `witerate` and `length` also disappeared.

```
angel 57 (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
    71028                      0.00  0.00

     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

  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)

  %GC time       0.0%  (0.0% elapsed)

  Alloc rate    7,102,800 bytes per MUT second

  Productivity 100.0% of total user, 35756475700.0% of total elapsed
```

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.

Program	Description
`exp3_8`	Calculate 3^8 using Naturals
`gen_regexps`	Generate all the expansions of a generalised regular expression
`paraffins`	Generation of radicals
`primes`	Generate the first 1500 prime numbers
`queens`	Count the the number of solutions to the "n queens" problem
`rfib`	nfib 30 with *Doubles*
`tak`	Calculate *tak* 24 16 8
`x2n1`	Calculate a root to the equation $x^n = 1$ using complex numbers

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.

Program	Description	Author
`awards`	Public awards scheme	Kevin Hammond
`banner`	Simple banner program	Mark P Jones
`boyer`	Boyer benchmark	Denis Howe
`boyer2`	Gabriel benchmark 'Boyer'	
`calendar`	Calendar program	Mark P Jones
`cichelli`	Perfect hashing function	Iain Checkland
`circsim`	Circuit simulator	David King
`clausify`	Reducing propositions to clausal form	Colin Runciman
`cse`	Common subexpression elimination	Mark P Jones
`eliza`	Pseudo-psychoanalyst	Mark P Jones
`expert`	Minimal expers system	Ian Holyer
`fibheaps`	Fibonacci heaps	Chris Okasaki
`fish`		
`knights`	Knights tour	Jonathan Hill
`life`	Game of life	John Launchbury
`mandel`	Mandelbrot set generator	Jonathan Hill
`mandel2`	Mandelbrot set generator	David Hanley
`minimax`	Tic-tac-toe	Iain Checkland
`multiplier`	Binary multiplier	John T O'Donnell
`pretty`	Pretty printer	
`primetest`	Probabilistic primality testing	David Lester
`rewrite`	Rewriting system	Mike Spivey
`scc`	Strongly connected components of a graph	John Launchbury
`simple`	Standard Id benchmark	
`sorting`	Sorting algorithms	Will Partain
`sphere`	Ray tracer for spheres	David King

Table 6.2 Programs of the spectral subset

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

Program	Description
`comp_lab_zift`	Image processing application
`event`	Event driven simulation
`fft`	Two Fast Fourier Transforms
`genfft`	Generation of synthetic FFT programs
`ida`	Solution of a particular configuration of the n-puzzle
`listcompr`	Compilation of list comprehensions
`listcopy`	Compilation of list comprehensions
`parstof`	Wadler's method for lexing and parsing
`sched`	Calculation of an optimum schedule of parallel jobs
`solid`	Point membership classification algorithm
`transform`	Transformation of a number of programs represented as synchronous process networks into master-slave style parallel programs
`typecheck`	Polymorphic typechecking of a set of function definitions
`wang`	Wang's algorithm for solving a system of linear equations

Table 6.3 Programs of the spectral subset: the Hartel Benchmarks

Program	Description	Author
`anna`	Strictness analyser	
`bspt`	BSP tree modeller	Iain Checkland
`compress`	Text compression	Paul Sanders
`ebnf2ps`	Syntax diagram generator	Peter Thiemann
`fluid`	Fluid dynamics program	Xiaoming Zhang
`fulsom`	Solid modeling	Duncan Sinclair
`gamteb`	Monte Carlo photon transport	Pat Fasel
`gg`	Graphs from GRIP statistics	Iain Checkland
`grep`	Grep program	
`hpg`	Haskell program generator	Nick North
`infer`	Hindley-Milner type inference	Phil Wadler
`lift`	Fully-lazy lambda lifter	David Lester & Simon Peyton Jones
`maillist`	Mailing list generator	Paul Hudak
`mkhprog`	Command line parser generator	N D North
`parser`	Partial Haskell parser	Julian Seward
`prolog`	mini-Prolog interpreter	Mark P Jones
`reptile`	Escher tiling program	Sandra Foubister
`rsa`	RSA encryption	John Launchbury
`symalg`	Command line evaluator	
`veritas`	Theorem prover	Gareth Howells

Table 6.4 Programs of the real subset

type $\forall\, \alpha\, \beta . \beta \rightarrow (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$ i.e. the two arguments standing for [] and (:) are swapped around. The two methods, the hard-wired and somewhat optimised functions in the Prelude, and the full-blown implementation of warm fusion would compete with most probably unimaginable consequences.

2. **Separate compilation**. In the three subsets of the `nofib` benchmark suite the programs are written rather differently. The *imaginary* subset consists of small programs, therefore all the necessary type declarations are within the same file. Under these circumstances attempting fusion is not a problem (Section 5.5.1).

   The *spectral* subset is somewhat similar: with the exception of `boyer2` all the programs consist of one file, so fusion for these programs are still not problematic. `boyer2` exports one of its 'central' datatype — the one on which great many functions are defined — abstractly.

   The *real* subset is rather different: in these programs the datatypes are usually defined in separate files and in some cases are exported abstractly. As explained in Section 5.5.1, fusion for abstractly exported datatypes (datatypes exported without their constructors) is not attempted.

These two limitations suggest that most `nofib` programs will not be affected by our transformations.

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.

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`exp3_8`	4.34	175002	2.82	161962500	2920
`gen_regexps`	4.77	201747	0.09	1061736	4192
`paraffins`	9.26	192330	1.85	25412408	10873068
`primes`	3.01	169882	1.07	28184000	19816
`queens`	3.03	167146	0.47	12114868	628
`rfib`	2.66	377418	0.41	7448	0
`tak`	2.59	178042	0.41	14963024	1168
`x2n1`	4.25	407802	0.49	18812528	1396
`wheel-sieve1`	4.56	175114	9.15	10270256	98972
`wheel-sieve2`	5.18	175706	3.64	38202816	9812932
`integrate`	4.99	385082	8.22	448223840	4958888

Table 6.5 Control run: imaginary subset

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`event`	10.96	182618	2.28	54794652	2944284
`fft`	14.71	404394	0.41	10804340	83084
`genfft`	13.31	188602	0.46	26356304	3372
`listcompr`	12.83	188010	2.13	129027180	2372
`listcopy`	12.9	188282	2.37	144099284	2152
`nucleic2`	92.4	479130	1.18	47629352	5632
`parstof`	93.07	327290	0.16	1350940	41204
`sched`	11.27	180074	0.45	17368564	3176
`solid`	40.63	424890	1.84	86154404	295332
`transform`	56.28	298602	6.51	307153768	27600
`typecheck`	18.19	201770	4.44	146058248	3552
`wang`	12.65	397674	1.51	44075624	3922944
`wave4main`	16.31	205450	5.39	48742552	1835352

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

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`atom`	6.69	389866	5.55	181389016	3172
`awards`	6.67	204922	0.12	105220	0
`banner`	9.38	203379	0.12	83584	0
`boyer`	13.34	199226	0.62	27229080	31376
`boyer2`	20.54	231610	0.23	1730252	31832
`calendar`	6.69	180858	0.10	346404	2336
`cichelli`	18.31	225235	1.56	34323100	21680
`circsim`	25.24	284522	16.13	684021348	12754416
`clausify`	7.97	190746	0.66	19524212	3432
`constraints`	12.32	197386	64.27	965843528	20793288
`cryptarithm1`	3.57	173082	16.86	926684208	2228
`cryptarithm2`	13.51	211482	0.49	19785960	4248
`cse`	11.58	209562	0.11	529672	3608
`eliza`	13.89	228259	0.13	262428	0
`expert`	25.01	241539	0.15	111236	0
`fibheaps`	9.97	234250	0.56	25572344	154980
`fish`	10.34	185658	0.24	7152788	2024
`gcd`	3.8	178618	0.59	25989928	5428
`life`	5.24	182730	4.61	222927164	12512
`knights`	32.39	235930	0.16	968360	16540
`mandel`	13.34	466627	3.38	103856092	19332
`mandel2`	6.55	186634	0.24	4512064	2352
`minimax`	21.38	212499	0.13	2684428	2992
`multiplier`	9.27	188634	1.99	95067900	18616
`para`	9.74	227779	9.65	350640368	95228
`primetest`	13.82	251267	7.77	89433208	62176
`puzzle`	10.8	191578	2.75	58604352	2140
`rewrite`	21.52	220842	0.53	14027148	8048
`scc`	4.81	171962	0.10	6284	0
`sorting`	9.62	200467	0.15	524164	59448
`sphere`	20.72	466010	2.15	55503740	3000

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.

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
anna	319.18	969180	1.90	30906900	138540
compress	23.94	275203	3.02	134907892	107440
ebnf2ps	132.33	538316	0.18	2872968	102792
fem	94.78	516243	0.55	26373684	28724
fulsom	77.16	560122	8.10	274028100	2626908
gamteb	55.01	510579	1.47	48100772	7040
gg	78.3	574531	0.19	4339328	100328
grep	22.87	238675	0.10	5196	0
hpg	60.12	641594	1.19	40928592	4872
infer	58.13	323411	1.19	13664232	16312
lift	31.53	239002	0.13	331456	4980
maillist	5.46	203763	0.22	5615964	108232
mkhprog	17.17	222234	0.13	1173676	5608
parser	60.53	343571	0.51	14785448	102512
pic	44.21	480115	0.25	3794652	152796
prolog	28.71	244563	0.13	755952	13852
reptile	75.43	368083	0.19	7315340	25492
rsa	7.16	240227	1.58	22060704	34444
veritas	287.93	840883	0.11	585408	15216

Table 6.8 Control run: the real subset

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
exp3_8	1.21	1.01	1.01	1.00	1.00
gen_regexps	1.01	1.00	1.22	1.00	1.00
paraffins	1.09	1.01	1.02	1.00	1.00
primes	1.03	1.00	0.98	1.00	1.00
queens	1.02	1.00	1.13	1.00	1.00
rfib	1.00	1.00	0.95	1.00	NA
tak	1.04	1.00	0.95	1.00	1.00
x2n1	1.06	1.00	1.16	1.00	1.00
wheel-sieve1	1.00	1.00	1.00	1.00	1.00
wheel-sieve2	1.00	1.00	1.01	1.00	1.00
integrate	1.01	1.00	1.03	1.00	1.00
Minimum	1.00	1.00	0.95	1.00	1.00
Maximum	1.21	1.01	1.22	1.00	1.00
Geometric mean	1.04	1.00	1.03	1.00	1.00

Table 6.9 Normalise: imaginary subset

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

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`event`	1.11	1.01	1.05	1.00	1.00
`fft`	1.07	1.00	1.12	1.00	1.00
`genfft`	1.05	1.00	1.09	1.00	1.00
`listcompr`	1.07	1.00	1.06	1.00	1.00
`listcopy`	1.07	1.00	1.04	1.00	1.00
`nucleic2`	1.12	1.01	1.01	1.00	1.00
`parstof`	1.09	1.00	0.94	1.00	1.00
`sched`	1.17	1.02	0.96	1.00	1.00
`solid`	1.12	1.01	1.09	1.00	1.00
`transform`	1.07	1.00	1.08	1.00	1.00
`typecheck`	1.09	1.00	1.07	1.00	1.00
`wang`	1.06	1.00	1.10	1.00	1.00
`wave4main`	1.08	1.00	1.06	1.00	1.00
Minimum	1.05	1.00	0.94	1.00	1.00
Maximum	1.12	1.02	1.12	1.00	1.00
Geometric mean	1.08	1.01	1.05	1.00	1.00

Table 6.10 Normalise: the Hartel Benchmarks

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.

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
atom	1.01	1.00	1.05	1.00	1.00
awards	1.01	1.00	0.67	1.00	NA
banner	1.04	1.00	0.83	1.00	NA
boyer	1.23	1.01	0.90	1.00	1.00
boyer2	1.02	1.00	1.26	1.00	1.00
calendar	1.11	1.00	1.10	1.00	1.00
cichelli	1.07	1.00	1.07	1.00	1.00
circsim	1.14	1.01	1.08	1.00	1.00
clausify	1.27	1.02	1.12	1.00	25.68[2]
constraints	1.13	1.01	1.06	0.98	1.00
cryptarithm1	1.11	1.00	1.08	1.00	1.00
cryptarithm2	1.11	1.00	0.88	1.00	1.00
cse	1.08	1.00	1.09	1.00	1.00
eliza	1.09	1.00	0.92	1.00	NA
expert	1.12	1.01	1.00	1.00	NA
fibheaps	1.17	1.01	1.07	1.00	1.00
fish	1.12	1.00	0.92	1.00	1.00
gcd	1.09	1.00	1.05	1.00	1.00
life	1.06	1.00	1.08	1.00	1.00
knights	1.08	1.01	1.00	1.00	1.00
mandel	1.08	1.00	1.05	1.00	1.00
mandel2	1.11	1.01	1.21	1.00	1.00
minimax	1.09	1.00	0.85	1.00	1.00
multiplier	1.07	1.00	1.08	1.00	1.00
para	1.06	1.00	1.05	1.00	1.00
primetest	1.06	1.00	1.05	1.00	1.00
puzzle	1.13	1.01	1.07	1.00	1.00
rewrite	1.12	1.01	1.09	1.00	1.00
scc	1.09	1.00	1.00	1.00	NA
sorting	1.16	1.01	0.80	1.00	1.00
sphere	1.11	1.00	1.01	1.00	1.00
Minimum	1.01	1.00	0.67	0.98	1.00
Maximum	1.27	1.02	1.26	1.00	1.00
Geometric mean	1.09	1.01	1.00	0.99	1.00

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 microprocessors, larger and larger amounts of memory and improvements to GHC. Run-

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`anna`	1.06	1.00	0.95	1.00	1.00
`compress`	1.06	1.01	1.10	1.00	1.00
`ebnf2ps`	1.07	1.00	1.00	1.00	1.00
`fem`	1.09	1.01	0.84	1.00	1.00
`fulsom`	1.13	1.01	1.27	1.10	0.75
`gamteb`	1.08	1.00	1.10	1.00	1.00
`gg`	1.08	1.00	0.95	1.00	1.00
`grep`	1.01	1.00	1.10	1.00	NA
`hpg`	1.00	1.00	0.98	1.00	1.00
`infer`	1.01	1.00	1.05	1.00	1.00
`lift`	1.08	1.00	0.92	1.00	1.00
`maillist`	1.09	1.00	1.27	1.00	1.00
`mkhprog`	1.08	1.00	0.69	1.00	1.00
`parser`	1.11	1.01	1.16	1.00	1.00
`pic`	1.07	1.00	1.08	1.00	1.00
`prolog`	1.06	1.00	0.92	1.00	1.00
`reptile`	1.08	1.00	1.26	1.00	1.00
`rsa`	1.07	1.00	1.09	1.00	1.00
`veritas`	1.04	1.01	1.09	1.00	1.00
Minimum	1.00	1.00	0.69	1.00	0.75
Maximum	1.13	1.01	1.27	1.00	1.00
Geometric mean	1.06	1.00	1.03	1.00	0.98

Table 6.12 Normalise: the real subset

$$
\begin{aligned}
\textbf{data}\ \mathit{Formula}\ =\ & \mathit{Sym\ Char} \\
\mid\ & \mathit{Not\ Formula} \\
\mid\ & \mathit{Dis\ Formula\ Formula} \\
\mid\ & \mathit{Con\ Formula\ Formula} \\
\mid\ & \mathit{Imp\ Formula\ Formula} \\
\mid\ & \mathit{Eqv\ Formula\ Formula}
\end{aligned}
$$

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

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`exp3_8`	1.29	1.01	1.56	1.40	0.66
`gen_regexps`	1.01	1.00	1.44	1.00	1.00
`paraffins`	1.09	1.01	1.00	1.00	1.00
`primes`	1.06	1.00	1.04	1.00	1.00
`queens`	1.02	1.00	1.36	1.00	1.00
`rfib`	0.98	1.00	0.98	1.00	NA
`tak`	1.00	1.00	1.05	1.00	1.00
`x2n1`	1.05	1.00	0.96	1.00	1.00
`wheel-sieve1`	1.03	1.00	0.99	1.00	1.00
`wheel-sieve2`	1.02	1.00	1.03	1.00	1.00
`integrate`	1.04	1.00	1.05	1.00	1.00
Minimum	0.98	1.00	0.96	1.00	0.66
Maximum	1.29	1.01	1.56	1.40	1.00
Geometric mean	1.05	1.00	1.11	1.03	0.96

Table 6.14 Buildify only: imaginary subset

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`event`	1.06	1.01	1.05	1.01	1.04
`fft`	1.03	1.00	0.98	1.00	1.00
`genfft`	1.01	1.00	0.93	1.00	1.00
`listcompr`	1.02	1.00	0.99	1.00	1.00
`listcopy`	1.02	1.00	0.94	1.00	1.00
`nucleic2`	1.10	1.01	0.98	1.01	1.03
`parstof`	1.04	1.00	0.88	1.00	1.00
`sched`	1.15	1.02	0.84	1.19	0.93
`solid`	1.11	1.02	1.81	1.97	1.00
`transform`	1.02	1.00	1.00	1.00	1.00
`typecheck`	1.10	1.01	1.01	1.07	0.92
`wang`	1.01	1.00	0.99	1.00	1.00
`wave4main`	1.03	1.00	1.00	1.00	1.00
Minimum	1.01	1.00	0.88	1.00	0.92
Maximum	1.15	1.02	1.81	1.97	1.04
Geometric mean	1.05	1.01	1.01	1.07	0.99

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

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`atom`	1.03	1.00	1.03	1.00	1.00
`awards`	1.03	1.00	0.83	1.00	NA
`banner`	1.06	1.00	1.08	1.00	NA
`boyer`	1.24	1.01	0.97	1.00	1.00
`boyer2`	1.01	1.00	0.96	1.00	1.00
`calendar`	1.06	1.00	1.40	1.00	1.00
`cichelli`	1.04	1.00	1.01	1.08	0.97
`circsim`	1.13	1.01	1.03	1.00	1.00
`clausify`	1.43	1.05	23.94	11.72	3234.33
`constraints`	1.07	1.01	1.00	0.98	1.00
`cryptarithm1`	1.05	1.00	1.03	1.00	1.00
`cryptarithm2`	1.05	1.00	0.92	1.00	1.00
`cse`	1.05	1.00	1.27	1.00	1.00
`eliza`	1.05	1.00	0.85	1.00	NA
`expert`	1.12	1.02	0.80	1.01	NA
`fibheaps`					
`fish`	1.07	1.00	0.79	1.00	1.00
`gcd`	1.04	1.00	1.00	1.00	1.00
`life`	1.00	1.00	1.00	1.00	1.00
`knights`	1.04	1.01	1.25	1.00	1.00
`mandel`	1.02	1.00	0.99	1.00	1.00
`mandel2`	1.05	1.01	1.08	1.00	1.00
`minimax`	1.06	1.01	1.00	1.01	1.14
`multiplier`	1.03	1.00	0.97	1.00	1.00
`para`	1.01	1.00	1.00	1.00	1.00
`primetest`	1.00	1.00	0.99	1.00	1.00
`puzzle`	1.12	1.01	1.08	1.20	1.18
`rewrite`	1.12	1.01	0.89	1.00	1.02
`scc`	1.03	1.00	1.10	1.00	NA
`sorting`	1.22	1.02	0.53	1.00	1.00
`sphere`	1.07	1.00	0.96	1.00	1.00
Minimum	1.01	1.00	0.79	1.00	0.97
Maximum	1.43	1.05	23.49	11.72	3234.23
Geometric mean	1.06	1.01	1.08	1.09	1.3

Table 6.16 Buildify only: spectral subset

stack space. These are omitted from the benchmarks without further notice.

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`anna`	1.05	1.00	0.99	1.00	1.00
`compress`	1.09	1.02	1.03	1.01	1.47
`ebnf2ps`	1.02	1.00	0.94	1.00	1.00
`fem`	1.04	1.01	0.96	1.00	1.00
`fulsom`	1.13	1.03	1.04	1.15	0.96
`gamteb`	1.01	1.00	0.97	1.00	1.00
`gg`	1.05	1.01	1.11	1.00	1.00
`grep`	1.02	1.00	1.20	1.00	NA
`hpg`	1.02	1.00	0.99	1.00	1.00
`infer`	1.05	1.01	1.03	0.99	1.05
`lift`	1.03	1.00	0.69	1.00	1.00
`maillist`	1.02	1.00	1.14	1.00	1.00
`mkhprog`	1.03	1.00	0.77	1.00	1.00
`parser`	1.10	1.01	0.80	1.00	1.00
`pic`	1.01	1.00	0.92	1.00	1.00
`prolog`	1.03	1.00	0.77	1.00	1.00
`reptile`	1.02	1.00	0.95	1.00	1.00
`rsa`	0.99	1.00	0.99	1.00	1.00
`veritas`	1.05	1.01	1.27	1.00	1.00
Minimum	0.99	1.00	0.69	0.99	0.96
Maximum	1.13	1.03	1.27	1.15	1.47
Geometric mean	1.03	1.01	0.96	1.00	1.02

Table 6.17 Buildify only: the real subset

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.

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`exp3_8`	1.20	1.01	0.95	1.00	1.00
`gen_regexps`	1.03	1.00	1.00	1.00	1.00
`paraffins`	1.10	1.01	0.94	1.00	1.00
`primes`	1.04	1.00	1.03	1.00	1.00
`queens`	1.03	1.00	1.13	1.00	1.00
`rfib`	1.05	1.00	1.10	1.00	NA
`tak`	0.99	1.00	1.05	1.00	1.00
`x2n1`	1.06	1.00	1.10	1.00	1.00
`wheel-sieve1`	0.99	1.00	0.99	1.00	1.00
`wheel-sieve2`	1.02	1.00	1.00	1.00	1.00
`integrate`	1.00	1.00	1.01	1.00	1.00
Minimum	0.99	1.00	0.94	1.00	1.00
Maximum	1.20	1.01	1.13	1.00	1.00
Geometric mean	1.04	1.00	1.02	1.00	1.00

Table 6.18 Catify only: imaginary subset

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.

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
atom	1.03	1.00	0.97	1.00	1.00
awards	1.03	1.00	0.83	1.00	NA
banner	1.05	1.00	1.00	1.00	NA
boyer	1.74	1.08	0.98	1.00	1.00
boyer2	1.02	1.00	0.91	1.00	1.00
calendar	1.03	1.00	1.20	1.00	1.00
cichelli	1.05	1.01	1.01	1.00	1.00
circsim	1.11	1.01	1.00	1.00	1.00
clausify	1.20	1.02	1.11	1.00	25.68
constraints	1.08	1.01	0.99	0.98	1.00
cryptarithm1	1.04	1.00	1.00	1.00	1.00
cryptarithm2	1.03	1.00	0.94	1.00	1.00
cse	1.04	1.00	1.00	1.00	1.00
eliza	1.06	1.00	0.62	1.00	NA
fish	1.08	1.00	1.00	1.00	1.00
gcd	1.03	1.00	1.02	1.00	1.00
life	1.02	1.00	1.00	1.00	1.00
knights	1.10	1.03	1.06	0.97	1.05
mandel	1.01	1.00	0.99	1.00	1.00
mandel2	1.06	1.01	0.88	1.00	1.00
minimax	1.05	1.00	1.38	1.00	1.00
multiplier	1.01	1.00	1.01	1.00	1.00
para	1.02	1.00	1.00	1.00	1.00
primetest	1.01	1.00	0.99	1.00	1.00
puzzle	1.47	1.07	1.00	1.00	1.00
scc	1.02	1.00	1.00	1.00	NA
sorting	1.11	1.01	0.73	1.00	1.00
sphere	1.06	1.00	0.98	1.00	1.00
Minimum	1.01	1.00	0.62	0.97	1.00
Maximum	1.74	1.08	1.38	1.00	1.00
Geometric mean	1.08	1.02	0.97	1.00	1.00

Table 6.19 Catify only: spectral subset

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`event`	1.06	1.01	1.00	1.00	1.00
`fft`	1.03	1.00	0.95	1.00	1.00
`genfft`	1.00	1.00	1.02	1.00	1.00
`listcompr`	1.02	1.00	1.00	1.00	1.00
`listcopy`	1.02	1.00	0.97	1.00	1.00
`nucleic2`	1.10	1.01	0.96	1.00	1.00
`parstof`	1.05	1.00	1.00	1.00	1.00
`sched`	1.11	1.02	0.98	1.00	1.00
`transform`	1.03	1.00	1.00	1.00	1.00
`typecheck`	1.04	1.00	1.00	1.00	1.00
`wang`	1.01	1.00	0.97	1.00	1.00
`wave4main`	1.03	1.00	1.00	1.00	1.00
Minimum	1.00	1.00	0.95	1.00	1.00
Maximum	1.11	1.02	1.02	1.00	1.00
Geometric mean	1.04	1.01	0.98	1.00	1.00

Table 6.20 Catify only: the Hartel Benchmarks

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`ebnf2ps`	1.02	1.00	1.00	1.00	1.00
`fem`	1.05	1.01	0.80	1.00	1.00
`gamteb`	1.02	1.00	1.01	1.00	1.00
`grep`	1.02	1.00	1.10	1.00	NA
`hpg`	1.03	1.00	0.99	1.00	1.00
`infer`	1.02	1.00	1.01	1.00	0.98
`lift`	1.03	1.00	0.69	1.00	1.00
`maillist`	1.01	1.00	1.14	1.00	1.00
`mkhprog`	1.02	1.00	1.00	1.00	1.00
`parser`	1.28	1.04	0.98	1.00	1.00
`pic`	1.01	1.00	1.12	1.00	1.00
`prolog`	1.02	1.00	1.00	1.00	1.00
`reptile`	1.02	1.00	1.16	1.00	1.00
`rsa`	1.01	1.00	0.99	1.00	1.00
`veritas`	1.08	1.03	0.73	1.00	1.00
Minimum	1.01	1.00	0.69	1.00	0.98
Maximum	1.28	1.04	1.16	1.00	1.00
Geometric mean	1.03	1.01	0.97	1.00	0.99

Table 6.21 Catify only: the real subset

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`exp3_8`	1.39	1.02	1.90	1.80	0.35
`gen_regexps`	1.02	1.00	1.33	1.00	1.00
`paraffins`	1.10	1.01	0.97	1.00	1.00
`primes`	1.07	1.00	1.12	1.00	1.00
`queens`	1.03	1.00	1.32	1.00	1.00
`rfib`	1.00	1.00	0.95	1.00	NA
`tak`	1.02	1.00	1.00	1.00	1.00
`x2n1`	1.07	1.00	1.08	1.00	1.00
`wheel-sieve1`	1.00	1.00	0.98	1.00	1.00
`wheel-sieve2`	1.04	1.00	1.01	1.00	1.00
`integrate`	1.03	1.00	1.01	1.00	1.00
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

Table 6.22 Buildify, catify and the `cata-build` rule: imaginary subset

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`atom`	1.02	1.00	0.98	1.00	1.00
`awards`	1.04	1.00	1.00	1.00	NA
`banner`	1.08	1.00	0.83	1.00	NA
`boyer`	1.78	1.08	1.06	1.00	1.00
`boyer2`	1.02	1.00	0.96	1.00	1.00
`calendar`	1.06	1.00	1.20	1.00	1.00
`cichelli`	1.09	1.01	1.01	1.08	0.97
`circsim`	1.19	1.02	1.04	1.03	0.98
`clausify`	1.46	1.05	23.35	11.72	3234.33
`constraints`	1.08	1.01	0.99	0.98	1.00
`cryptarithm1`	1.06	1.00	1.00	1.00	1.00
`cryptarithm2`	1.04	1.00	0.88	1.00	1.00
`cse`	1.06	1.00	0.73	1.00	1.00
`eliza`	1.05	1.00	0.77	1.00	NA
`fish`	1.07	1.00	0.75	1.00	1.00
`gcd`	1.03	1.00	1.08	1.00	1.00
`life`	1.04	1.00	1.00	1.00	1.00
`knights`	1.11	1.03	0.88	0.97	1.05
`mandel`	1.03	1.00	0.99	1.00	1.00
`mandel2`	1.10	1.01	0.88	1.04	1.03
`minimax`	1.09	1.01	0.92	1.01	1.14
`multiplier`	1.03	1.00	0.96	1.00	1.00
`para`	1.03	1.00	1.00	1.00	1.00
`primetest`	1.02	1.00	1.01	1.00	1.00
`puzzle`	1.41	1.05	1.08	1.23	1.08
`scc`	1.05	1.00	1.10	1.00	NA
`sorting`	1.26	1.03	0.73	1.00	1.00
`sphere`	1.06	1.00	0.96	1.00	1.00
Minimum	1.02	1.00	0.73	0.97	0.97
Maximum	1.78	1.08	23.35	11.72	3234.33
Geometric mean	1.10	1.01	1.06	1.10	1.34

Table 6.23 Buildify, catify and the `cata-build` rule: spectral subset

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`wave4main`	1.06	1.00	1.00	1.00	1.00
`wang`	1.01	1.00	1.03	1.00	1.00
`typecheck`	1.11	1.01	1.00	1.07	0.92
`transform`	1.04	1.00	1.01	1.00	1.00
`sched`	1.17	1.02	1.00	1.18	1.05
`nucleic2`	1.16	1.01	1.00	1.01	1.03
`parstof`	1.05	1.00	1.06	1.00	1.00
`listcopy`	1.03	1.00	1.00	1.00	1.00
`listcompr`	1.03	1.00	1.01	1.00	1.00
`genfft`	1.02	1.00	1.00	1.00	1.00
`fft`	1.04	1.00	1.05	1.00	1.00
`event`	1.08	1.01	1.09	1.01	1.04
Minimum	1.01	1.00	1.00	1.00	0.92
Maximum	1.17	1.02	1.06	1.18	1.05
Geometric mean	1.06	1.01	1.02	1.02	1.00

Table 6.24 Buildify, catify and the `cata-build` rule: the Hartel Benchmarks

Program	Time to compile	Binary size	Time to run	Total allocation	Max Heap Residency
`ebnf2ps`	1.03	1.00	0.72	1.00	1.00
`fem`	1.07	1.01	0.85	1.00	1.00
`gamteb`	1.03	1.00	1.03	1.00	1.00
`grep`	1.01	1.00	0.90	1.00	NA
`hpg`	1.03	1.00	1.00	1.00	1.00
`infer`	1.05	1.01	1.01	1.00	0.98
`lift`	1.04	1.00	1.08	1.00	1.00
`maillist`	1.01	1.00	1.14	1.00	1.00
`mkhprog`	1.04	1.00	1.00	1.00	1.00
`parser`	1.32	1.04	0.94	1.00	1.00
`pic`	1.01	1.00	0.84	1.00	1.00
`prolog`	1.03	1.00	0.92	1.00	1.00
`reptile`	1.03	1.00	0.79	1.00	1.00
`rsa`	1.01	1.00	1.01	1.00	1.00
`veritas`	1.10	1.03	1.00	1.00	1.00
Minimum	1.01	1.00	0.72	1.00	0.98
Maximum	1.32	1.04	1.14	1.00	1.00
Geometric mean	1.05	1.01	0.94	1.00	0.99

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 complier 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 `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, *Place* α, 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 **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 α. Nothing else changes, apart from the recursive uses of the type being defined, where the extra type argument is needed, since the *Place* constructor remains hidden from the user.

- The typechecker needs to be modified to restrict the uses of the new constructor.
- The `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 *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. *filter*) would use the familiar fold operator, while another function (e.g. *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 the 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^{nd}-order lambda calculus, which in GHC terminology is called the

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 pases 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.

- **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: β-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

Rule	Before	After	Condition
beta reduction	$(\lambda\, v.e)\; x$	$e[x/v]$	
typed beta reduction	$(\Lambda\, \tau.e)\; \sigma$	$e[\sigma/\tau]$	
dead code removal	**let** $v = e_v$ **in** e	e	v doesn't occur free in e
inlining	**let** $v = e_v$ **in** e	**let** $v = e_v$ **in** $e[e_v/v]$	several see Santos's thesis [San95]
case of known constructor	**case** $C_i\; v_1 \ldots\; v_n$ **of** $C_1\; \ldots \quad \rightarrow\; e_1$ $\vdots$ $C_i\; w_1 \ldots\; w_n\; \rightarrow\; e_i$	$e_i[v_1/w_1 \ldots\; v_n/w_n]$	
case of error	**case** *error* E **of** $\vdots$	*error* E	
case elimination	**case** v_1 **of** $v_2 \rightarrow e$	$e[v_1/v_2]$	
let to case	**let** $v = e_v$ **in** e	**case** e_v **of** $v \rightarrow e$	e is strict in v and e_v is not in weak head normal form

Table A.1 Local transformations

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 refered to in the body of the thesis by their names without further discussion. The interested reader is again refered 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 `lets`.

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^{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.

- **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.

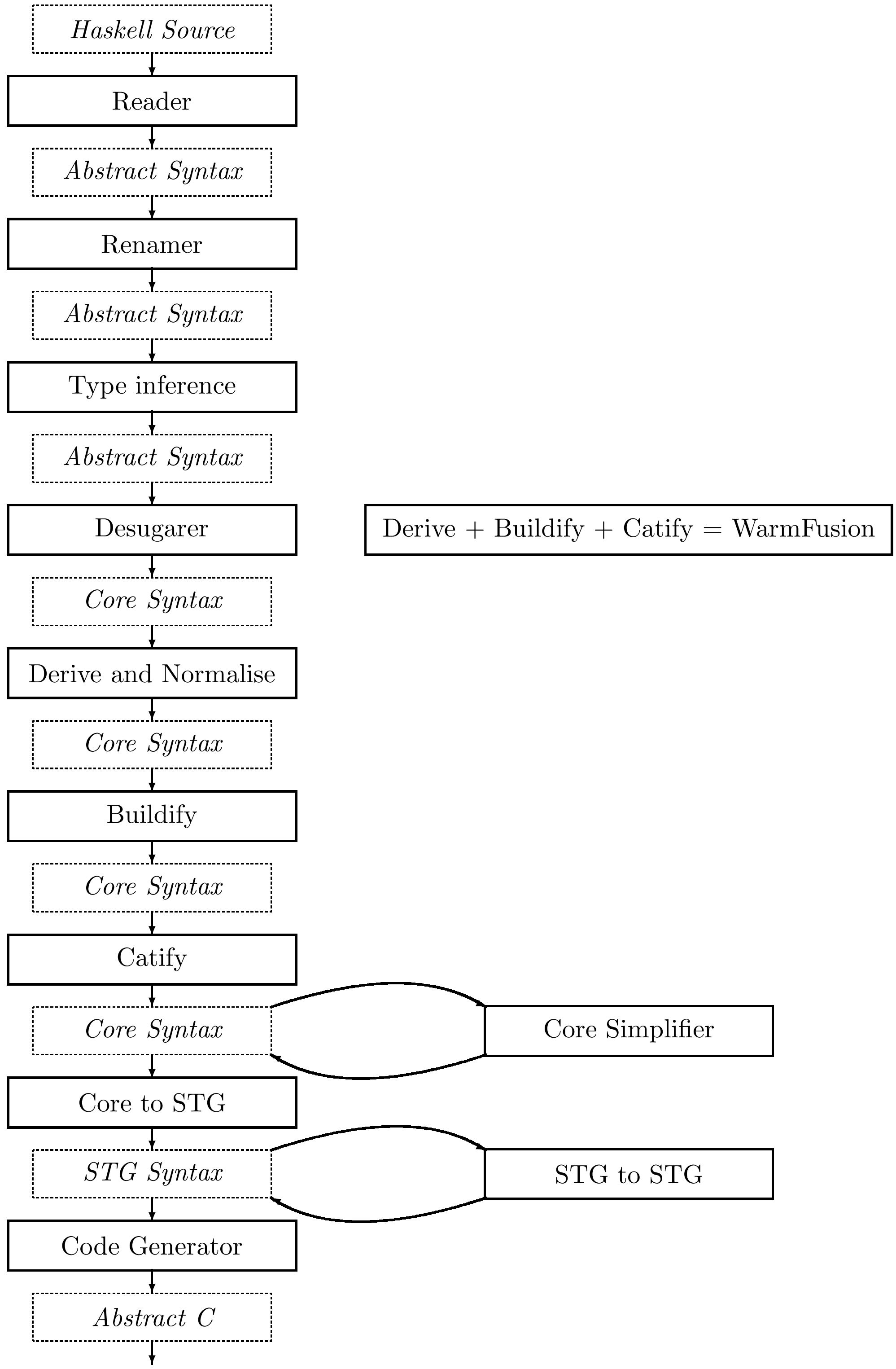

Figure A.1 Glasgow Haskell Compiler passes

Program	$Prog$	::=	$TopDecl_1$; ... ; $TopDecl_n$	$n \geq 1$
Declarations	$TopDecl$	::=	$Binding \mid TypeDecl$	
Declaration	$TypeDecl$	::=	`data` $Con\ \bar{\alpha}$ = $\{C_i\ \bar{\tau}_i\}_{i=1}^{n}$	
Types	τ	::=	$TyCon\ [\tau]$	Constructor application
		|	$\tau \rightarrow \tau'$	Function space
		|	$\forall \alpha . \tau$	Universal quantification
		|	α	Type variable
Bindings	$Binding$	::=	$Bind \mid$ `rec` $Bind_1 \ldots Bind_n$	
	$Bind$	::=	$var :: \tau$ = $Expr$	
Expression	$Expr$	::=	$Expr\ Atom$	Application
		|	$Expr\ \tau$	Type application
		|	$\lambda\ var_1 :: \tau_1 \ldots var_n :: \tau_n . Expr$	Lambda abstraction
		|	$\Lambda\ ty\ .\ Expr$	Type abstraction
		|	`case` $Expr$ `of` $Alts$	Case expression
		|	`let` $Binding$ `in` $Expr$	Local definition
		|	$con\ var_1 \ldots var_n$	Constructor $n \geq 0$
		|	$prim\ var_1 \ldots var_n$	Primitive $n \geq 0$
		|	$Atom$	
Atoms	$Atom$	::=	$var :: \tau$	Variable
		|	$Literal$	Unboxed Object
Literal values	$Literal$	::=	$integer \mid float \mid \ldots$	
Alternatives	$Alts$	::=	$Calt_1$; ... ; $Calt_n$; $Default$	$n \geq 0$
		|	$Lalt_1$; ... ; $Lalt_n$; $Default$	$n \geq 0$
Constr. alt	$Calt$	::=	$Con\ var_1 \ldots var_n$ -> $Expr$	$n \geq 0$
Literal alt	$Lalt$	::=	$Literal$ -> $Expr$	
Default alt	$Default$	::=	`NoDefault` | var -> $Expr$	

Figure A.2 Syntax of the Core language

Bibliography

[ASU86] Alfred V Aho, R Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, tools.* Addison-Wesley, 1986.

[Aug87] Lennart Augustsson. *Compiling lazy functional languages, Part II.* PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 1987.

[BC85] Joseph L Bates and Robert L Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, pages 113–136, 1985.

[BD77] Rodney Martineau Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BDM97] Richard S Bird and Oege De Moor. *Algebra of Programming.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1997.

[Bir86] Richard S Bird. An Introduction to the Theory of Lists. Technical Report PRG-56, Oxford University, Computing Laboratory, Programming Research Group, October 1986.

[Bir87] Richard S Bird. A Calculus of Functions for Program Derivation. Technical Report PRG-64, Oxford University, Computing Laboratory, Programming Research Group, December 1987.

[Bir89] Richard S Bird. Algebraic Identities for Program Calculation. *The Computer Journal*, 32(2), 1989.

[BM75] R S Boyer and J S Moore. Proving theorems about LISP programs. *Journal of the ACM*, 22(1), 1975.

[BM98] Richard S Bird and Lambert G T L Meertens. Nested datatypes. In *4th International Conference on Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–??, 1998.

[Boq99] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages.* PhD thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, 1999.

[BP99] Richard S Bird and Ross Paterson. Generalised Folds for Nested Datatypes. *Formal Aspects of Computing*, 11(2):200–222, September 1999.

[Car82] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1982.

[CF91] J Robin B Cockett and T Fukushima. About Charity. Technical Report 92/480/18, Department of Computer Science, University of Calgary, Canada, 1991.

[Chi90] Wei-Ngan Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, 1990.

[Chi92a] Wei-Ngan Chin. Fully lazy higher-order removal. In Charles Consel, editor, *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 38–47. Yale Uni., June 1992. YALEU/DCS/RR-909.

[Chi92b] Wei-Ngan Chin. Safe fusion of functional expressions. *ACM LISP Pointers*, 5(1):11–20, 1992. Proceedings of the 1992 ACM Conference on LISP and Functional Programming.

[Chi93] Wei-Ngan Chin. Towards an automated tupling strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. PEPM'93*, pages 119–132. ACM Press, 1993.

[Chi94] Wei-Ngan Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, October 1994.

[Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. *ACM Sigplan Notices, International Conference of Functional Programming (ICFP'99)*, 34(9):249–260, 1999.

[Chi00] Olaf Chitil. Type-inference based short cut deforestation (nearly) without inlining. In *Proceedings of the 11th International Workshop on Implementation of Functional Languages, Lochem, Netherlands*, 2000.

[CK93] Wei-Ngan Chin and S C Khoo. Tupling functions with multiple recursion parameters. *Lecture Notes in Computer Science*, 724:124–??, 1993.

[Coo66] D C Cooper. The equivalence of certain computations. *The Computer Journal*, 9:45–52, 1966.

[Cou90] B Courcelle. Recursive applicative program schemes. In J van Leuveen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 459–492. Elsevier, 1990.

[DB76] John Darlington and Rodney Martineau Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[DC94] Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Conference on Lisp and Functional Programming*, pages 273–282. LISP Pointers, July-September 1994.

[Der93] Nachum Dershowitz. A taste of rewrite systems. In P. E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 199–228. Springer-Verlag, 1993. Proceedings of International Lecture Series 1991-92, McMaster University Lecture Notes in Computer Science 693.

[DMS99] Oege De Moor and G Sittampalam. Generic program transformation. *Lecture Notes in Computer Science*, 1608:116–??, 1999.

[Feg96] Leonidas Fegaras. Fusion for free! Technical Report CSE-96-001, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, January 8, 1996.

[FM94] Maarten M Fokkinga and Lambert G T L Meertens. Adjunctions. Memoranda informatica, University of Twente, June 1994.

[Fok92a] Maarten M Fokkinga. *A Gentle Introduction to Category Theory — the calculational approach*. University of Utrecht, 1992.

[Fok92b] Maarten M Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Technical University Twente, The Netherlands, 1992.

[Fok94] Maarten M Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[Fre90] Peter Freyd. Recursive types reduced to inductive types. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 498–507, 1990.

[FS95] Leonidas Fegaras and Tim Sheard. Using compact data representations for languages based on catamorphisms. Technical Report 95-025, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 1995.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Proceedings of 23rd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, 21–24 January 1996.

[FSS92] Leonidas Fegaras, Tim Sheard, and David Stemple. Uniform traversal combinators: Definition, use and properties. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *LNAI*, pages 148–162, Saratoga Springs, NY, June 1992. Springer-Verlag.

[FSZ94] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–32, Orlando, Florida, 25 June 1994.

[FW86] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.

[FW89] Alex Ferguson and Philip Wadler. When will deforestation stop? In *Proceedings of the 1989 Glasgow Functional Programming Workshop*, 1989.

[Gil96] Andrew John Gill. *Cheap Deforestation for Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1996.

[GLPJ93] Andrew John Gill, John Launchbury, and Simon L Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the 6th ACM Conference on Functional Programming and Computer Architecture*, April 1993.

[GM93] M J C Gordon and Thomas F Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Hag88] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. Technical Report ECS-LFCS-88-44, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, January 1988.

[Har94] Pieter H Hartel. Benchmarking implementations of lazy functional languages II – Two years later. Technical Report Cs-94-21, Deptartment of Comp. Sys, University of Amsterdam, December 1994.

[HIT96a] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Calculating accumulations. Technical Report METR 96-03, Dept. of Mathematical Engineering, Univ. of Tokyo, March 1996.

[HIT96b] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Cheap tupling in calculational form. *Lecture Notes in Computer Science*, 1140:471–??, 1996.

[HIT96c] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Construction of list homomorphisms by tupling and fusion. *Lecture Notes in Computer Science*, 1113:407–418, 1996.

[HIT96d] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. *ACM Sigplan Notices*, 31(6):73–82, June 1996.

[HIT96e] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Formal derivation of parallel program for 2-Dimensional maximum segment sum problem. *Lecture Notes in Computer Science*, 1123:553–??, 1996.

[HIT97] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An extension of the acid rain theorem. In T. Ida, A. Ohori, and M. Takeichi, editors, *Proceedings 2nd Fuji Intl. Workshop on Functional and Logic Programming, Shonan Village Center, Japan, 1–4 Nov 1996*, pages 91–105. World Scientific, Singapore, 1997.

[HITT97] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. *ACM Sigplan Notices*, 32(8):164–??, August 1997.

[HJ94] Fritz Henglein and Jesper Jorgensen. Formally Optimal Boxing. In *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[HL93] Pieter H Hartel and Koen Langendoen. Benchmarking implementations of lazy functional languages. In *Functional Programming & Computer Architecture*, pages 341–349, June 1993.

[Hu96] Zhenjiang Hu. *A Calculational Approach to Optimising Functional Programs*. PhD thesis, Department of Information Engineering, University of Tokyo, 1996.

[IHT98] Hideya Iwasaki, Zhenjiang Hu, and Masato Takeichi. Towards manipulation of mutually recursive definitions. To appear in Proceedings FUJI'98, 1998.

[Jeu95] Johan Jeuring. Polytypic pattern matching. In *Conference on Functional Programming and Computer Architecture*, 1995.

[JJ97] Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *Conference record of POPL '97: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France*, 1997.

[JM95] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Joh] Thomas Johnsson. Sharing Analysis + EVAL inlining + Unboxing = Deforestation.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jouannaud [Jou85], pages 190–203.

[Joh94] Thomas Johnsson. Fold-unfold transformations on state monadic interpreters. In *Proceedings of the 1994 Glasgow Functional Programming Workshop*, Workshops in Computing, Ayr, 1994. Springer-Verlag.

[Joh98] Thomas Johnsson. Graph reduction, and how to avoid it. *Theoretical Computer Science*, 194(1–2):244–??, March 1998.

[Jou85] Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1985.

[KH89] Richard Kelsey and Paul Hudak. Realistic Compilation by Program Transformation. In *Principles of Programming Languages*, January 1989.

[KL95] Richard B Kieburtz and Jeffrey R Lewis. Programming with algebras. In Jeuring and Meijer [JM95], pages 267–307.

[Klo96] Jan Willem Klop. Term graph rewriting. *Lecture Notes in Computer Science*, 1074, 1996.

[KT92] K Kaneko and Masato Takeichi. Relationship between lambda hoisting and fully lazy lambda lifting. *Journal of Information Processing*, 15(4):564–569, 1992.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference record of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, 1992.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 314–323, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[MA86] E G Manes and M A Arbib. *Algebraic Approaches to Program Semantics*. Springer-Verlag, 1986.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[Mal89] Grant R Malcolm. Homomorphisms and promotability. In J L A van de Snepscheut, editor, *Proceedings of the International Conference on Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, June 1989.

[Mal90] Grant R Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–280, 1990.

[Mar95] Simon David Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Department of Computing Science, University of Glasgow, 1995.

[Mee86] Lambert G T L Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 189–334, 1986.

[Mee90] Lambert G T L Meertens. Paramorphisms. Technical Report CS-R9005, CWI, 1990.

[Mei92] Erik Meijer. *Calculating Compilers*. PhD thesis, University of Nijmegen, The Netherlands, 1992.

[Mel88] Thomas F Melham. Automating Recursive Type Definitions in Higher Order Logic. Technical Report 146, University of Cambridge, Computer Laboratory, September 1988.

[MFP91] Erik Meijer, Maarten M Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In Simon L Peyton Jones, editor, *Functional Programming & Computer Architecture*, pages 324–333. ACM, 1995.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MJ95] Erik Meijer and Johan Jeuring. Merging Monads and Folds for Functional Programming. In Jeuring and Meijer [JM95].

[Mog91] Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.

[MWCG97] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language (extended version). Technical Report TR97-1651, Cornell University, Computer Science, November 1997.

[NPJ98] László Németh and Simon L Peyton Jones. A design for warm fusion. In *Conference Record of the 10th International Workshop on Implementation of Functional Languages*, pages 381–393, 1998.

[OHIT97] Y Onue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In Richard S Bird and Lambert G T L Meertens, editors, *Proceedings IFIP TC 2 WG 2.1 Working Conf. on Algorithmic Languages and Calculi, Le Bischenberg, France, 17–22 Feb 1997*, pages 76–106. Chapman & Hall, London, 1997.

[Par90] H A Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[Par92] G Park. Semantic analyses for storage management optimizations in functional language implementations. Technical Report TR-597, Department of Computer Science, New York University, February 1992.

[Pie91] Benjamin C Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[PJ87] Simon L Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[PJ92] Simon L Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 1992.

[PJ96] Simon L Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44, Linköping, Sweden, 22–24 April 1996. Springer-Verlag.

[PJH99] Simon L Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell 98*. February 1999.

[PJL91a] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. *Lecture Notes in Computer Science*, 523, 1991.

[PJL91b] Simon L Peyton Jones and David R Lester. A modular fully-lazy lambda lifter in HASKELL. *Software – Practice & Experience*, 21(5):479–506, 1991. Also Research Report CSC/90/R17, Department of Computer Science, University of Glasgow (1990).

[PJM99] Simon L Peyton Jones and Simon David Marlow. Secrets of the Glasgow Haskell Compiler inliner. In *IDL'99*, 1999.

[PJS96] Simon L Peyton Jones and André Luís de Medeiros Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1996.

[PK82] Robert Paige and S Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, 1982.

[PP96a] Alberto Pettorossi and Maurizio Proietti. Future directions in program transformation. *ACM, Computing Surveys*, 28(4), 1996.

[PP96b] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM, Computing Surveys*, 28(2), 1996.

[PS87] Alberto Pettorossi and A Skowron. Higher order generalisation in program derivation. In *Proceedings of Tapsoft'87 (Pisa, Italy)*, volume 250 of *Lecture Notes in Computer Science*, pages 182–196. Springer-Verlag, 1987.

[Rey83] John C Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, pages 513–523, 1983.

[San95] André Luís de Medeiros Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1995.

[San96a] Dave Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(10), October 1996. Preliminary version in TAPSOFT'95, LNCS 915.

[San96b] Dave Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[SDM93] Doaitse S Swierstra and Oege De Moor. Virtual data structures. *Lecture Notes in Computer Science*, 155:355–??, 1993.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the 6th ACM Conference on Functional Programming and Computer Architecture*, pages 233–242. ACM, 1993.

[SF94] Tim Sheard and Leonidas Fegaras. Optimizing algebraic programs. Technical Report CSE-94-004, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, February 1994.

[SGJ94] Morten Heine Sorensen, Robert Glück, and Neil D Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. *Lecture Notes in Computer Science*, 788, 1994.

[SRA94] Zhong Shao, John H Reppy, and Andrew W Appel. Unrolling lists. In *Conference record of the 1994 ACM Conference on Lisp and Functional Programming*, pages 185–191, June 1994.

[SW94] Manuel Serrano and Pierre Weis. 1+1=1: An optimizing Caml compiler. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 101–111, Orlando (Florida, USA), June 1994.

[TA90] Masato Takeichi and Yoji Akama. Deriving a functional Knuth-Morris-Pratt algorithm by transformation. *Journal of Information Processing*, 13(4):522–528, 1990.

[Tak87] Masato Takeichi. Partial parameterization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.

[THT98] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program transformation in calculational form. *ACM, Computing Surveys*, 30, September 1998.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In Simon L Peyton Jones, editor, *Programming of the 8th ACM Conference on Functional Programming and Computer Architecture*, pages 306–313. ACM, 1995.

[TMC+96] Dave Tarditi, Greg Morrisett, P Cheng, C Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. *ACM Sigplan Notices*, 31(5):181–192, May 1996. Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[Tur86] Valentin F Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[TWM95] David N Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Programming of the 8th ACM Conference on Functional Programming and Computer Architecture*, San Diego, California, 1995.

[Wad81] Philip Wadler. Applicative style programming, program transformation and list operators. In *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture*, pages 25–32, 1981.

[Wad84] Philip Wadler. Listlessness is better than laziness. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 45–52. ACM, August 1984.

[Wad85a] Philip Wadler. How to replace failure by a list of successes. In Jouannaud [Jou85], pages 113–128.

[Wad85b] Philip Wadler. Views: A way for elegant definitions and efficient representations to coexist. In Thomas Johnsson et al., editor, *Aspenæs Workshop on Implementation of Functional Languages*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1985.

[Wad86] Philip Wadler. Listlessness is better than laziness II: Composing listless functions. In *Lecture Notes in Computer Science*, volume 217. Springer-Verlag, October 1986.

[Wad87a] Philip Wadler. Fixing some space leaks with a garbage collector. *Software – Practice & Experience*, 1987.

[Wad87b] Philip Wadler. *List comprehensions*, chapter 7. In spj:book [PJ87], 1987.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the 4th ACM Conference on Functional Programming and Computer Architecture*, pages 347–359. ACM Press, London, September 1989.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, June 1990.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In Jeuring and Meijer [JM95].

[WPJ93] Philip Wadler and Simon L Peyton Jones. Imperative functional programming. In *Proceeding of the 20th Annual ACM SIGACT-SIGPLAN Symposium on Pronciple of Programming Languages*, pages 71–84, 1993.

[WPJ99] Keith Wansbrough and Simon L Peyton Jones. Once upon a polymorphic type. In *Conference record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–28, 1999.

[WS72] S A Walker and H R Strong. Characterisation of flowchartable recursions. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, Denver, Co., USA*, 1972.

[XIT94] L Xu, Hideya Iwasaki, and Masato Takeichi. Derivation of algorithms by introduction of generation functions. *New Generation Computing*, 13(1):75–98, 1994.

Index