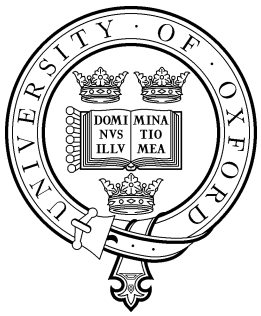
Theory and Practice of Shortcut Fusion



Thomas Harper

Oriel College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2013

Abstract

There are a number of approaches for eliminating intermediate data structures in functional programs by rewriting a composition of recursive functions as a single recursive function. Such a transformation is called *fusion*. One such approach is to encapsulate a structured recursion scheme in two combinators for consumption and production of data, and use algebraic transformations to rewrite these where possible, letting local compiler optimisations fuse the remaining nonrecursive portions of the program. This approach is called *shortcut fusion*, which has been implemented using various recursion schemes in the programming language Haskell. Despite their obvious similarities, however, the relationship between these techniques has not been formalised.

In this thesis, three techniques, chosen for their success in practical applications and prominence in previous literature, are analysed. Their relationship is examined on three different levels: theory, practice, and pragmatics. Theoretically, the relationship between their underlying recursion schemes is examined. In the right setting, it is possible to compare them side-by-side, which makes clearer their differences in expressibility as well as the foundations for their correctness. On the practical level, the similarities in their implementations in Haskell can be generalised using the concept of data abstraction, and a general semantic framework is developed for shortcut fusion without reference to a specific recursion scheme. Finally, the pragmatics of these techniques are investigated, and it turns out that the these, too, are not dependent on the specific technique used.

The results from this analysis demonstrate that shortcut fusion is actually a single program transformation, defined by a general semantic framework that can be instantiated for a variety of recursion schemes. Furthermore, it is possible to use this information to create a declarative infrastructure for implementing shortcut fusion within a compiler. This results in a more robust program transformation that is less complicated to implement than before.

Acknowledgements

I would first like to thank my supervisor, Ralf Hinze. I am so grateful for all the support and guidance he provided over the years, including no small amount of time spent proofreading. I would also like to thank Atze Dijkstra for supervising a portion of my thesis at Utrecht University, and helping me obtain the grant to do so.

My examiners, Jeremy Gibbons and Andy Gill, also deserve thanks in advance for agreeing to read my thesis, something I hope they do not regret too much after passing this page.

I would also like to thank Jeremy along with my other friends and colleagues at the Oxford University Department of Computer Science for providing such a great community. John Lyle, Cornelius Namiluko, Shamal Faily, Nicolas Wu, and José Pedro Magalhães were all excellent officemates with whom cups of tea were just as important to my research as their insights into my work. Daniel James, with whom I shared a supervisor and worked closely with over the years, deserves special thanks for the intellectual and emotional support he provided over the years we worked together.

I have also had the privilege befriending some spectacular non-computer scientists during my DPhil, and I could not have done it all without them. I fear that any attempt to properly name them all would leave someone out, so I must simply hope they know who they are. One in particular, however, deserves special mention. Thank you, Jamie Anderson, for inspiring me to apply to a university a world away and then sharing this incredible experience with me. I look forward to sharing many more.

Finally, thanks goes to my fiancé, James Whitehead. While doing his own DPhil, he has been with me on this journey the whole way, and always believed in me and supported me no matter what. His devotion was never more clearly demonstrated than when he proofread my entire thesis in one sitting.

Contents

Chapter 1

Introduction

In programming, there is a well-recognised tension between abstraction and performance. A good abstraction hides unnecessary, repetitive details, allowing a programmer to reuse code to write simpler, clearer programs. Abstraction, however, also hides implementation details so that the programmer cannot control how these discrete, modular pieces of code interact from the outside. From a design perspective, such opacity is desirable, but the solutions it yields are often not the most performant, because good performance is often the result of more tightly coupled, problem-specific code. Such a tension arises in the use of recursive, higher-order functions in the functional programming language Haskell. Consider the example program[1]

$$sumSq\ x = sum\ (map\ (\uparrow 2)\ (enumFromTo\ 1\ x))$$

which generates the enumeration from 1 to x, squares each of the resulting numbers, and sums the squares together. Such a code snippet demonstrates the elegance that this modular, compositional style can provide; a function can be constructed by composing simpler functions together. In this case, these functions encapsulate common recursion patterns over lists. The result is a concise and modular program in which the programmer's intent is clear, and only requires that they specify the details relevant to the problem at hand.

The downside, however, is the impact that this style has on performance. When combining recursive functions together in this way, each one must communicate its results to the next as a list of numbers. In the example above, a list of the integers from 1 to x is generated, followed by a list containing the square of these numbers. These data structures are the means by which such modularity is achieved, but they appear neither in the argument nor in the result of the function, both of which are merely

[1]This example first appeared in Wadler's deforestation paper [Wadler, 1988], but continues to be used as one of the canonical example programs for fusion in Haskell.

single integers. For this reason, these structures are referred to as *intermediate data structures*. Despite the fact that the abstraction provided by these functions hides them, intermediate data structures can have a very visible effect on performance. They occupy memory, and their allocation and garbage collection can cause significant slowdown.

To remove the slowdown, the function can be rewritten so that it produces no intermediate data structures.

```
sumSq' x = go 1
  where
    go y = if y > x
           then 0
           else y ↑ 2 + go (y + 1)
```

The new function performs the same task as the original, but does not produce any intermediate data structures. Instead, each integer is calculated, squared, and immediately added to the result of the recursive call. The process of transforming a program that uses intermediate data structures into one that does not is called *deforestation*. In this case, deforestation is accomplished by combining the multiple recursive traversals found in the original function into a single one. The resulting function completely processes each element before moving on to the next, and combines that result with the result of the recursive call. Because each element is processed entirely in this first pass, there is no need to create a data structure with the intermediate results. Such a program transformation is known as *fusion*, which refers to the fact that a composition of small, modular functions is combined into a single, monolithic one that achieves the same result.

Although the intermediate data structures have disappeared, so have all of the benefits of the original function. The new, performant version is also more specialised, without any of the modularity, concision, or clarity of the original. Ideally, we would like have our cake and eat it, too, by writing the original version of the function and having the compiler produce the fused one automatically. Elegance is also not the only motivation for such a transformation; in programs of non-trivial size, it can be infeasible to spot all opportunities for fusion and take advantage of them. Doing so also requires direct access to the data structure in question. This is not always possible, for example, if the data structure is only usable via an API, nor is it preferable, because it prevents the programmer from taking advantage of the abstraction provided by such an interface.

Automating the fusion of generally recursive functions is no easy task. Among the difficulties faced is termination; transforming a recursive function involves unfolding its definition, which, of course, contains a call to the function itself. One way to bring this problem into a more tractable realm is to consider only a subset of recursive functions that can be transformed without sophisticated analysis. This approach originated with Wadler's Deforestation Algorithm [Wadler, 1988], which was to be implemented in an optimising compiler for a functional language. The algorithm is quite ambitious in attempting to fuse all recursive functions over datatypes, but requires that all programs be written in a restricted form in order to guarantee correctness and termination. This limitation, in addition to the complexity of implementation, prevented it from being widely adopted. However, it served as the inspiration for later, more targeted techniques, which this thesis collectively refers to as *shortcut fusion*. Such techniques can be characterised by their common usage of combinators that encapsulate recursive consumption and production of data structures. The combinators, if defined appropriately, can be used to implement functions over these data structures. The composition of functions so defined can then be easily transformed by simple syntactic transformations into fully fused programs.

Despite the high-level description just given that is inclusive of all of these techniques, they remain, for the most, only considered individually as distinct but related approaches to fusion. They are often developed and presented in an *ad hoc* manner in the sense that they are presented specialised to a specific data structure and set of functions. Furthermore, these techniques depend on a compiler's existing collection of optimisations to accomplish fusion in passing; although the result is often a fused function, these optimisations can conflict with each other or be applied according to heuristics, making the desired transformation outcome, fusion, rather fragile.

With all these ancillary details in the way, the commonalities of these techniques are often obscured in their native settings. It is clear, however, that they nevertheless share certain characteristics that put them into the same 'family'. The purpose of this thesis is to define precisely what that family is and how each of the techniques presented fits into it. By lifting these techniques out of their syntax- and data structure-dependent settings and examining them more closely, insights about their relationships can be gleaned that in turn lead to insights about their shared practical and pragmatic characteristic. All of these serve to reinforce the central idea of this thesis, which is that *shortcut fusion is a general program transformation of which each of the techniques analysed is but an instantiation.*

In this thesis, three stand-out shortcut fusion techniques are analysed: *foldr/build* fusion, *destroy/unfoldr* fusion, and stream fusion. This analysis is conducted on three different levels. First, they are considered in a theoretical setting that is particularly suited to comparison. Second, these insights are used to develop a framework in which shortcut fusion is presented without reference to specific data structures or recursion schemes. Third, the issues associated with depending upon local compiler optimisations are addressed. The use of these optimisations with shortcut fusion is examined in order to develop a more robust infrastructure for instantiating it for a given data structure and set of functions. More specifically, the contributions of the thesis are as follows:

- **A comparison of shortcut fusion techniques in the setting of recursive coalgebras**. The fusion techniques examined will be modelled and analysed in the setting of recursive coalgebras. As will be shown, recursive coalgebras provide a useful setting for examining the fundamental relationships between the shortcut fusion techniques shown. Of particular note is that the differences in the expressibility of different techniques are clearer in this setting, and can be explained without resorting to syntactic comparisons (Chapter 3). This unifies previous work on the modelling of shortcut fusion and provides a starting point for tying theoretical work more closely to the practical. These contributions have been published in Hinze et al. [2010], as a joint with Ralf Hinze and Daniel James. The author of this thesis claims credit for the presentation of stream fusion in terms of data abstraction, the relationship between fusion and Church and Cochurch encodings, and the discussion of recursiveness as his individual contributions.

- **A framework for implementing shortcut fusion in Haskell along with sample instantiations**. Exploring the theoretical foundations is useful for tying together the underpinnings of each technique. It provides a vocabulary for elucidating how and why they differ, but examining them in this way focuses on the underlying recursion schemes, which emphasizes the differences rather than similarities among these techniques. On an implementation level, however, they can be drawn back together by providing a general framework that describes shortcut fusion without referring to details specific to any technique. The framework includes shared set of definitional principles and proof obligations, which can be instantiated for each of the given techniques. This is done using leaf trees as an example data structure. The details of how these proof

obligations are satisfied relate back to the theory (Chapter 4). This is an extended version of the work published in Harper [2011], a paper solely authored by the author of this thesis.

- **A robust infrastructure for implementing shortcut fusion in the Utrecht Haskell Compiler**. Shortcut fusion is highly dependent on local compiler optimisations to actually remove intermediate data structures. These local optimisations are not traditionally implemented with shortcut fusion in mind, and without proper tuning, they can actually impede the fusion process. The solution has been to contort these with a series of compiler pragmas that result in a working but fragile program transformation. Because of the shared implementation procedures of these shortcut fusion techniques, however, a more declarative infrastructure can be developed. Furthermore, it turns out that despite the differing underlying recursion schemes, the local transformations by each technique and how they are applied is the same for all the shortcut fusion techniques analysed in this thesis. Such an infrastructure is implemented and the details of how this dictates the use of the local transformations is discussed (Chapter 5). In contrast with the rest of the thesis, this research was supervised by Atze Dijkstra of Utrecht University. It is solely the work of the author of this thesis. It was presented at the Implementation and Application of Functional Languages 2012.

In addition to these chapters, Chapter 2 provides the necessary background. This consists of an overview of each of the shortcut fusion techniques covered, along with a more precise definition of the term 'shortcut fusion' for the purpose of this thesis. It also introduces the necessarily theoretical concepts leveraged in later chapters. Chapter 6 covers the relevant related work, and Chapter 7 reviews the material presented and concludes.

Chapter 2

Background

This chapter reviews the concepts that set the stage for the rest of the thesis. It introduces the concept of 'shortcut fusion', providing an overview of the incarnations under discussion as well as a definition of what makes a fusion technique an instance of shortcut fusion. It also provides an overview of the mathematical concepts used for modelling programs and program transformations in subsequent chapters

The material presented in Sections 2.2 and 2.3 is adapted from introductory material presented in Harper [2011] and Hinze et al. [2011]. With the latter being a work of joint authorship, particular credit goes to Daniel James, who drew many of the commutative diagrams and coauthored the original introduction to recursive coalgebras with me.

2.1 Shortcut fusion

In the introduction, the concept of fusion as an optimisation was presented, and shortcut fusion as a means of achieving automated fusion was introduced. The analysis of three shortcut fusion techniques is central to this thesis, and therefore these techniques are presented in detail in this section. The presentations of these techniques follow those given in their original papers in order to set the stage for the analyses presented later on. These descriptions are then used to arrive at a more precise definition of 'shortcut fusion' for the purpose of this thesis, both in order to justify the choice of techniques and more clearly define the scope of the thesis.

2.1.1 foldr/build fusion

The first shortcut fusion technique was originally christened 'shortcut deforestation' [Gill et al., 1993]. Due to the proliferation of other techniques, it is also known

as *foldr/build* fusion, and will be so called in this thesis. The creation of *foldr/build* fusion arose as a way to provide fusion in Haskell for lists, its workhorse data structure, without the drawbacks in the Deforestation Algorithm that made it undesirable to implement in a compiler. In particular, the Deforestation Algorithm introduced 'substantial cost and complexity' in order to maintain correctness and termination. Even so, programs had to be written in a restricted style to guarantee the termination of the transformations.

These shortcomings arise from the fact that the Deforestation Algorithm is designed to be implemented as an optimisation that would attempt to fuse *all* recursive functions in an input program. Therefore, all the programs had to be in the appropriate, restricted form in order to keep the transformation from breaking. The *foldr/build* approach is to make the problem more tractable by focussing on fusing a pre-defined set of functions using 'algebraic transformations'. Such transformations are expressed as equalities where expressions matching the left-hand side can be transformed into an equivalent, but more efficient expression matching the right-hand side. An example of such a transformation is *map* fusion:

$$\forall f\ g.\ map\, f\, (map\, g\, xs) = map\, (f \circ g)\, xs$$

Here, the transformation depends on a property of *map*, i.e. that it preserves composition. By combining two *map*s into one, the intermediate data structure that would have contained the result of $map\, g\, xs$ and been passed to $map\, f$ has been removed. The implicit assumption is that, while *map* is recursive, and therefore not optimisable via inlining, f and g are not and can therefore be inlined and optimised as necessary. Although the solution is simple for a single function, it does not scale; writing algebraic transformations for all possible function pipelines would result in a combinatorial explosion of rules for even a small API.

This can be addressed by moving the abstraction up another level. Just as *map* encapsulates a list transformation that applies a function to each element, there is a more general pattern that encapsulates *map* along with other functions over a list. Similarly, a property of this pattern can be exploited in order to achieve fusion between any two functions defined thus. As its name would suggest, the general pattern for *foldr/build* fusion is *foldr*, which has the following definition:

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ c\ n\ [\,] \quad\ = n \\ &foldr\ c\ n\ (x : xs) = c\ x\ (foldr\ c\ n\ xs) \end{aligned}$$

Figure 2.1 Some common list functions defined using *foldr*

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum}\ \mathit{xs} = \mathit{foldr}\ (+)\ 0\ \mathit{xs} \\
&\mathit{filter} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{filter}\ p\ \mathit{xs} = \mathit{foldr}\ (\lambda y\ \mathit{ys} \to \mathbf{if}\ p\ y\ \mathbf{then}\ y : \mathit{ys}\ \mathbf{else}\ b)\ [\,]\ \mathit{xs} \\
&(+\!\!+) :: [a] \to [a] \to [a] \\
&\mathit{xs} +\!\!+ \mathit{ys} = \mathit{foldr}\ (:)\ \mathit{ys}\ \mathit{xs} \\
&\mathit{concat} :: [[a]] \to [a] \\
&\mathit{concat}\ \mathit{xss} = \mathit{foldr}\ (+\!\!+)\ [\,]\ \mathit{xss} \\
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to b \\
&\mathit{foldl}\ f\ z\ \mathit{xs} = \mathit{foldr}\ (\lambda b\ g\ a \to g\ (f\ a\ b))\ \mathit{id}\ \mathit{xs}\ z
\end{aligned}
$$

The *foldr* function recursively replaces the constructors (:) (pronounced *cons*) and [ ] (pronounced *nil*) of a list by the function c and value n, respectively. This provides a uniform way to consume lists by combining the head of a list with the result of recursive combining the tail, replacing the empty list with n. Therefore, c and n each describe the details of a single, nonrecursive step, while *foldr* encapsulates the recursive pattern. Using *foldr*, it is possible to define *map* as

$$
\begin{aligned}
&\mathit{map}_f :: (a \to b) \to [a] \to [b] \\
&\mathit{map}_f\ f\ \mathit{xs} = \mathit{foldr}\ (\lambda a\ b \to f\ a : b)\ [\,]\ \mathit{xs}
\end{aligned}
$$

which applies f to each element a, and then *cons*es this result onto the result of the recursive call, replacing [ ] with itself when it reaches the end of the list. Figure 2.1 contains some more examples of common list functions defined using *foldr*.

It might seem as though the abstraction work stops here, and all that remains is simply to fuse *foldr*s in the same way as *map*s. This is not the case, however. In order to fuse such a program, it would be necessary to be able to fuse programs of the form

$$\mathit{foldr}\ c_1\ n_1\ (\mathit{foldr}\ c_2\ n_2\ \mathit{xs})$$

This requires some way to combine c_1 and c_2 into function that can be passed to a single *foldr*, but yield the same result. This is difficult because, although it is clear that c_2 constructs a list, both its and c_1's definitions are opaque, which does not provide much insight into how they may be rewritten into a single function without further analysis not afforded by these simple algebraic transformations. This was the

approach taken by Sheard [Sheard and Fegaras, 1993], which required such analysis and had restrictions similar to the Deforestation Algorithm.

The program can be rephrased in such a way that it is possible to exert more control, however. This can be accomplished by abstracting away the list constructors as arguments of a function that performs the fold and then uses said arguments in place of (:) and [ ]. When this function is applied to the list constructors themselves, the expected list is produced. As an example, consider *map* defined thus:

$$
\begin{aligned}
&map'_f :: (a \to b) \to [\,a\,] \to [\,b\,] \\
&map'_f\, f\; xs = (\lambda c\; n \to foldr\; (\lambda a\; b \to f\; a\; `c`\; b)\; n\; xs)\; (:)\; [\,]
\end{aligned}
$$

(Note that Haskell allows c to be used as an infix operator by enclosing it backticks (‘), which is done to emphasise that it replaces the infix constructor (:).) Now, even if the placement of the list constructors within the arguments to *foldr* is not known precisely, it is nevertheless possible to control when and with which functions the list is constructed. The next step is to abstract this list construction pattern away into another function, called *build*:

$$
\begin{aligned}
&build :: (\forall b.(a \to b \to b) \to b \to b) \to [\,a\,] \\
&build\; g = g\; (:)\; [\,]
\end{aligned}
$$

which yields the following definition of *map*:

$$
\begin{aligned}
&map_{fb} :: (a \to b) \to [\,a\,] \to [\,b\,] \\
&map_{fb}\, f\; xs = build\; (\lambda c\; n \to foldr\; (\lambda a\; b \to f\; a\; `c`\; b)\; n\; xs)
\end{aligned}
$$

The construction of lists is now syntactically apparent; it happens wherever a *build* appears. Furthermore, this happens in a standardised way: *foldr* consumes the list according to its pattern, putting the placeholders where the constructors should go, and *build* is responsible for passing the list constructors to the resulting anonymous function. (Therefore, a subscripted *fb* will henceforth be used to denote *foldr/build*-style functions). Additionally, *build* allows for the definition of functions that produce a list, but do not consume one. For example, *enumFromTo*, can be defined as

$$
\begin{aligned}
&enumFromTo :: (Ord\; a, Enum\; a) \Rightarrow a \to a \to [\,a\,] \\
&enumFromTo\; m\; n = build\; (enumFromTo'\; m\; n) \\
&enumFromTo'\; i\; j\; c\; n = \mathbf{if}\; i > j \\
&\qquad\qquad\qquad\qquad \mathbf{then}\; n \\
&\qquad\qquad\qquad\qquad \mathbf{else}\; c\; i\; (enumFromTo'\; (succ\; i)\; j\; c\; n)
\end{aligned}
$$

Figure 2.2 Some common list functions defined using *foldr* and *build*

$$
\begin{aligned}
&\mathit{repeat}_{fb} :: a \to [a] \\
&\mathit{repeat}_{fb}\, x = \mathit{build}\, (\lambda c\, n \to \mathbf{let}\, r = c\, x\, r\, \mathbf{in}\, r) \\
&\mathit{filter}_{fb} :: (a \to \mathit{Bool}) \to [a] \to [a] \\
&\mathit{filter}_{fb}\, p\, xs = \mathit{build}\, (\lambda c\, n \to \mathit{foldr}\, (\lambda a\, b \to \mathbf{if}\, p\, a\, \mathbf{then}\, c\, a\, b\, \mathbf{else}\, b)\, n\, xs) \\
&(+\!\!+) :: [a] \to [a] \to [a] \\
&xs +\!\!+ ys = \mathit{build}\, (\lambda c\, n \to \mathit{foldr}\, c\, (\mathit{foldr}\, c\, n\, ys)\, xs) \\
&\mathit{concat}_{fb} :: [[a]] \to [a] \\
&\mathit{concat}_{fb}\, xss = \mathit{build}\, (\lambda c\, n \to \mathit{foldr}\, (\lambda x\, y \to \mathit{foldr}\, c\, y\, x)\, n\, xss) \\
&\mathit{zip}_{fb} :: [a] \to [b] \to [(a, b)] \\
&\mathit{zip}_{fb}\, xs\, ys = \mathit{build}\, (\lambda c\, n \to \mathbf{let}\, \mathit{zip}'\, (x : xs)\, (y : ys) = c\, (x, y)\, (\mathit{zip}'\, xs\, ys) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{zip}'\, \_\quad\ \_\quad\ = n \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ \mathit{zip}'\, xs\, ys)
\end{aligned}
$$

The recursive function *enumFromTo′* describes how the values of an enumeration are generated and specifies where the constructors would be placed, but the actual creation of the list is performed by *build*. Therefore, the appearance of *build* signifies that a function produces a list, and the appearance of *foldr* that a function consumes one. Many functions, such as *map* and *filter*, do both. The list functions from Figure 2.1 can be also redefined in this style, and are shown in Figure 2.2. Note that *sum*, which consumes a list but does not produce one, remains unchanged. Furthermore, the *repeat* function, which produces a list by repeating its argument infinitely, does not have a definition in terms of *foldr* at all, but is now shown since it can be defined in terms of *build*.

Returning to the motivating example for fusion, inlining it with these new definitions yields the program

$$
\begin{aligned}
\mathit{sumSq}_{fb}\, x = \mathit{foldr}\, (+)\, 0\, (\mathit{build}\, (\lambda c\, n \to \mathit{foldr}\, (\lambda a\, b \to c\, (a \uparrow 2)\, b)\, n \\
(\mathit{build}\, (\mathit{enumFromTo}'\, 1\, x))))
\end{aligned}
$$

Now, a *foldr* is consuming a list created by a *build*, which means that a list is being created only to be immediately consumed. This means an intermediate data structure is created. The goal is therefore to fuse programs of the form

$$\mathit{foldr}\, c\, n\, (\mathit{build}\, g)$$

Here, g takes two arguments, which are used as placeholders for the list constructors. As said, g is expecting an argument that combines the head of a list with the result

of a recursive call on the tail, and a representation of the empty list. When applied to the list constructors, the function yields the underlying list. The *foldr* will then replace these same constructors with c and n. It is possible, therefore, to avoid creating the intermediate data structure by instead applying g to c and n directly, as it these functions that ultimately take their place in the computation. This idea leads to the following algebraic transformation:

$$\forall c\ n\ g.\ foldr\ c\ n\ (build\ g) = g\ c\ n$$

Applying this transformation to the example program yields the following:

$$sumSq_{fb}\ x = (\lambda c\ n \rightarrow foldr\ (\lambda a\ b \rightarrow c\ (a \uparrow 2)\ b)\ n\ (build\ (enumFromTo'\ 1\ x)))\ (+)\ 0$$

which leaves one more opportunity to apply the rule:

$$sumSq_{fb}\ x = (\lambda c\ n \rightarrow enumFromTo'\ 1\ x\ (\lambda a\ b \rightarrow c\ (a \uparrow 2)\ b)\ n)\ (+)\ 0$$

And finally, using beta reduction:

$$sumSq_{fb}\ x = enumFromTo'\ 1\ x\ (\lambda a\ b \rightarrow (a \uparrow 2) + b)\ 0$$

Although this is not precisely the transformed function show in he introduction (as *enumFromTo′* is recursive and therefore not inlinable), it nevertheless has no intermediate data structures. The values generated by *enumFromTo′* are added together as they are generated rather than being collected in a list. The above transformations depend entirely on functionality already available in the Glasgow Haskell Compiler (GHC) [Peyton Jones and Marlow, 2012], the canonical Haskell compiler. Specifically, GHC provides the ability to specify *rewrite rules* [Peyton Jones et al., 2001] that perform the algebraic transformations, and the inliner [Peyton Jones and Santos, 1998] performs both the inlining and beta reduction necessary to complete the transformation. The pragmatics of this approach are discussed in Chapter 5 and Appendix A. Key to this approach is that GHC can inline *nonrecursive* functions safely, but not recursive ones. This is not a problem here, because all the recursion is encapsulated by *foldr*, which is removed by the rewrite rules, and the arguments themselves describe only single, nonrecursive steps. This issue will crop up again the subsequent sections, however.

As with *map fusion*, the correctness of *foldr/build* fusion depends on a property of folds. In the case of *map* fusion, it was that maps preserve composition. In the case of *foldr/build*, we rely on the fact that c and n can be passed directly to g, because

foldr would ultimately use them to combine values of a list in the same way g would. This however, depends on an important property of g, namely that it actually uses its arguments in the way that is expected. The reassurance that this is the case comes from the type of g. The type signature of *build* specifies that g has the type

$$\forall b.(a \to b \to b) \to b \to b$$

for some fixed type a, which corresponds to the type of the elements of the underlying list (*build* itself is polymorphic in the type of the element, because it works for g's with underlying lists with elements of any type). Because g is polymorphic in b, its definition does not depend on the type of the result. This means that it can only obtain that value by using its arguments and using them in the expected way. This can be formalised by using the free theorem [Wadler, 1989] of g. This theorem states that, for all functions $f : A \to B \to B$, $f' : A \to B' \to B'$ and $h : B \to B'$ where h is strict:

$$\forall a\ b.h\ (f\ a\ b) = f'\ a\ (h\ b) \Rightarrow \forall b.h\ (g\ f\ b) = g\ f'\ (h\ b)$$

This can be instantiated such that $h := \mathit{foldr}\ c\ n$, $f := (:)$, and $f' := c$, obtaining the equation

$$\forall a\ b.\mathit{foldr}\ c\ n\ (a : b) = c\ a\ (\mathit{foldr}\ c\ n\ b) \Rightarrow \forall b.\mathit{foldr}\ c\ n\ (g\ (:)\ b) = g\ c\ (\mathit{foldr}\ c\ n\ b)$$

The precondition is satisfied by the definition of *foldr*. Furthermore, if we instantiate $b := [\,]$ on the right hand side, and $\mathit{foldr}\ c\ n\ [\,]$ is replaced with $[\,]$ according to the definition of *foldr*, then

$$\mathit{foldr}\ c\ n\ (g\ (:)\ [\,]) = g\ c\ n$$

Finally, since $\mathit{build}\ g = g\ (:)\ [\,]$ for any instantiation of g, it is possible to rewrite this expression as

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

which is precisely the equality sought after. Note that being able to depend on the type for this proof requires that *build* have a rank-2 polymorphic type [Leivant, 1983]. This is not supported by the Haskell 98 language standard [Peyton Jones and Others, 2003], but is now supported as a GHC extension. The use of free theorems for proofs in Haskell comes with certain caveats. These, along with other aspects of proving the correctness of shortcut fusion, are discussed in Chapter 4.

This *foldr/build* approach to fusion has enjoyed considerable success. It has been implemented in GHC versions of the *Data.List* library to optimise programs on lists that are written using library functions. The downside of this approach is that, like all shortcut fusion systems, it restricts list functions to those that can be defined using a specific recursion scheme, in this case *foldr* and *build*. Two salient examples of where this is a problem are *foldl* and *zip*. Although it is possible to define *foldl* in terms of *foldr*, as shown in Figure 2.1, this does not actually yield an increase in performance, even when fused. The reason for this is that this implementation builds up a series of thunks, one for each constructor, whereas the accumulating parameter of a true *foldl* may be strict. A second example is the *zip* function. Although it is possible to fuse the result of *zip* by defining it using *build*, as shown in Figure 2.2, this does not fuse the incoming lists. It is possible to define *zip* using *foldr*, but this still only fuses one of the input lists. The next section deals with *destroy/unfoldr* fusion, which attempts to address these shortcomings by looking at fusion centered on unfolds rather than folds.

2.1.2 destroy/unfoldr fusion

Whereas *foldr/build* depends on consuming lists with folds, *destroy/unfoldr* dualises this concept by focussing on creating lists as unfolds [Svenningsson, 2002]. The recursive pattern is encapsulated by *unfoldr*

$$
\begin{aligned}
&\mathit{unfoldr} :: (b \to \mathit{Maybe}\,(a, b)) \to b \to [a] \\
&\mathit{unfoldr}\; h\; b = \mathbf{case}\; h\; b\; \mathbf{of} \\
&\quad \mathit{Just}\,(a, b') \to a : \mathit{unfoldr}\; h\; b' \\
&\quad \mathit{Nothing} \quad\; \to [\,]
\end{aligned}
$$

As the dual to *foldr*, *unfoldr* provides a standardised way of *producing* lists rather than consuming them. It accomplishes this using a stepper function h, which it applies to a seed value b. The definition of h describes a single (nonrecursive) step of creating a list. When applied to b, it either yields *Just* an element, along with a new seed, or *Nothing*, meaning there is nothing left to be unfolded. In the former case, the element is *cons*ed onto the result of the recursive call with the new seed value, and in the latter case it simply returns the empty list. Therefore, *unfoldr* encapsulates a recursive pattern that describes how to yield a list of elements one a time, and h and b detail how a single step of this unfolding is done. As with *foldr*, many functions can be defined using *unfoldr*, and some examples are shown in Figure 2.3.

Figure 2.3 Some common list functions defined using *unfoldr*

$$
\begin{aligned}
&map_u\, f\; xs = unfoldr\; h_{map}\; xs \\
&\quad \textbf{where} \\
&\qquad h_{map}\; [\,] = Nothing \\
&\qquad h_{map}\; (x : xs) = Just\; (f\; x, xs) \\
&enumFromTo_u\; m\; n = unfoldr\; h_{eft}\; m \\
&\quad \textbf{where} \\
&\qquad h_{eft}\; i = \textbf{if}\; i > n \\
&\qquad\qquad\quad \textbf{then}\; Nothing \\
&\qquad\qquad\quad \textbf{else}\; Just\; (i, i+1) \\
&repeat_u\; x = unfoldr\; (\lambda x \rightarrow Just\; (x, x))\; x \\
&zip_u\; xs\; ys = unfoldr\; h_{zip}\; (xs, ys) \\
&\quad \textbf{where} \\
&\qquad h_{zip}\; ([\,], \_) \qquad\quad = Nothing \\
&\qquad h_{zip}\; (\_, [\,]) \qquad\quad = Nothing \\
&\qquad h_{zip}\; (x : xs, y : ys) = Just\; ((x, y), (xs, ys))
\end{aligned}
$$

In these definitions, *unfoldr* handles the production of the list, and the details of the consumption are hidden in the argument, dual to the situation seen in *foldr/build*. In those functions that consume a list, the list itself serves as the seed, and how it is consumed is determined by the stepper function. In the case of map_u, for example, h_{map} breaks off the head of the list, and yields the result of applying the f to it, and then leaves the tail as the next seed. This pattern, where each step takes an element of the list to yield a new one, is repeated throughout the example function definitions. Therefore, this can be abstracted into a second function,

$$
\begin{aligned}
&destroy :: (\forall b.(b \rightarrow Maybe\; (a, b)) \rightarrow b \rightarrow c) \rightarrow [\,a\,] \rightarrow c \\
&destroy\; g\; xs = g\; listpsi\; xs \\
&\quad \textbf{where} \\
&\qquad listpsi\; [\,] \qquad = Nothing \\
&\qquad listpsi\; (x : xs) = Just\; (x, xs)
\end{aligned}
$$

The argument g is a function that, like *unfoldr*, takes a stepper function $b \rightarrow Maybe\; (b, a)$ and a corresponding initial seed of type b. This function can be used to consume a list if the seed is a list and the stepper function one that breaks up a list one element at time. The function *listpsi* is such a function. It describes the single step of converting a list to a *Maybe* type, and breaking a non-empty list into

its head and tail. By applying g to *listpsi* along with a list xs, g is told to use *listpsi* to consume xs, which consumes one element at every step.

While *build* was used to standardize the creation of lists, which *foldr* does not encapsulate, *destroy* serves the dual purpose of standardising list consumption, while *unfoldr* encapsulates the list creation pattern. It is notable, however, that while *unfoldr* is the dual of *foldr*, *destroy*, with extra type parameter c, is not the dual of *build*. This is more obvious when the type of *build* is recast using *Maybe* in the same manner as *destroy*:

$$build :: (\forall b.Maybe\,(a, b) \to b) \to [\,a\,]$$

It is possible that *build* is actually an instantiation of a more general function where the extra parameter of type c would appear, which would be the actual dual to *destroy*. This has, perhaps, not surfaced because this parameter would not provide any additional utility in applying *foldr/build* fusion.

Using *destroy* and *unfoldr*, *map*, for example, can be defined as

$$\begin{aligned}
&map_{du}\,f\;xs = destroy\,(\lambda next\;s \to unfoldr\,(h_{map}\;next)\;s)\;xs\\
&\quad \textbf{where}\\
&\qquad h_{map}\;next\;s = \textbf{case}\;next\;s\;\textbf{of}\\
&\qquad\quad Nothing \quad\;\; \to Nothing\\
&\qquad\quad Just\,(x, ys) \to Just\,(f\;x, ys)
\end{aligned}$$

The idea is that the anonymous function takes the place of the argument g, and therefore should take a stepper function and a seed. (By analogy with *foldr/build*-style functions, subscripted *du* will be used to denote *destroy/unfoldr*-style functions). The h_{map} function calls this stepper function to get a value, which it then transforms using f. Using *unfoldr*, $h_{map}\;next$ is applied recursively to any successive seeds. The use of *destroy* means that *listpsi* is passed in for *next*, and xs as the seed. This means that when *unfoldr* applies h_{map} to the seed, it calls *listpsi* to split the seed (which is a list) into its head and tail. It then yields the head of the list as transformed by f, and the tail is unfolded recursively. Examples of other functions defined this way are shown in Figure 2.4. Of particular importance are the definitions of *foldl* and *zip*. Unlike their *foldr/build* counterparts, these functions are fully fusible. In the case of *foldl*, the stepper function is used to yield an element, but the element is then combined with an accumulated result, which is passed on in the recursive call. For *zip*, the stepper function of each list can be used to alternately yield an element of one list, then the other, and then they can be combined into a pair to yield an

Figure 2.4 Some common functions defined in terms of *destroy* and *unfoldr*

$$
\begin{aligned}
&sum_{du}\ xs = destroy\ h_{sum}\ xs\\
&\quad \textbf{where}\\
&\qquad h_{sum}\ next\ s = \textbf{case}\ next\ s\ \textbf{of}\\
&\qquad\quad Nothing \rightarrow 0\\
&\qquad\quad Just\ (a, s') \rightarrow a + h_{sum}\ next\ s'\\
&foldl_{du}\ f\ z\ xs = destroy\ (h_{foldl}\ z)\ xs\\
&\quad \textbf{where}\\
&\qquad h_{foldl}\ z\ next\ s = \textbf{case}\ next\ s\ \textbf{of}\\
&\qquad\quad Nothing \rightarrow z\\
&\qquad\quad Just\ (a, s') \rightarrow h_{foldl}\ (f\ z\ a)\ next\ s'\\
&foldr_{du}\ f\ z\ xs = destroy\ h_{foldr}\ xs\\
&\quad \textbf{where}\\
&\qquad h_{foldr}\ next\ s = \textbf{case}\ next\ s\ \textbf{of}\\
&\qquad\quad Nothing \rightarrow z\\
&\qquad\quad Just\ (a, s') \rightarrow f\ a\ (h_{foldr}\ next\ s')\\
&zip_{du}\ xs\ ys = destroy\ (\lambda next_1\ s_1 \rightarrow destroy\ (\lambda next_2\ s_2 \rightarrow\\
&\quad unfoldr\ (h_{zip}\ next_1\ next_2)\ (s_1, s_2))\ ys)\ xs\\
&\quad \textbf{where}\\
&\qquad h_{zip}\ next_1\ next_2\ (s_1, s_2) = \textbf{case}\ next_1\ s_1\ \textbf{of}\\
&\qquad\qquad\qquad Nothing \rightarrow Nothing\\
&\qquad\qquad\qquad Just\ (a, s_1') \rightarrow \textbf{case}\ next_2\ s_2\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad Nothing \rightarrow Nothing\\
&\qquad\qquad\qquad\qquad\qquad\qquad Just\ (b, s_2') \rightarrow Just\ ((a, b), (s_1', s_2'))\\
&filter_{du}\ p\ xs = destroy\ (\lambda next\ s \rightarrow unfoldr\ (h_{filter}\ next)\ s)\ xs\\
&\quad \textbf{where}\\
&\qquad h_{filter}\ next\ s = \textbf{case}\ next\ s\ \textbf{of}\\
&\qquad\quad Nothing \rightarrow Nothing\\
&\qquad\quad Just\ (a, s') \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ Just\ (a, s')\ \textbf{else}\ h_{filter}\ next\ s'
\end{aligned}
$$

element of the resulting list. As the recursiveness of both the consumption and the production are encapsulated by the combinators, both the arguments and the result can potentially be fused away. Those functions in Figure 2.3 that only produce a list remain unchanged (e.g. *enumFromTo*).

In a pipeline of such functions, consumption is denoted by *destroy* and production by *unfoldr*. The goal is to spot programs that produce a list only to immediately consume it, i.e. programs of the form

$$destroy\ g\ (unfoldr\ h\ s)$$

In such a program, *unfoldr* creates a list by applying argument h to s and then recursively to successive seeds. Then, g iterates over that list using *listpsi*, which simply breaks apart the list one element at a time. Instead, g could use h and s to yield the elements and process them as they are yielded. This leads to the rewrite rule:

$$destroy\ g\ (unfoldr\ h\ s) = g\ h\ s$$

It is now possible to see how the running example is optimised by this fusion technique. Inlining the definitions yields the function

$$\begin{aligned}
&sumSq_{du}\ x = destroy\ h_{sum}\ (destroy\ (\lambda next\ s \to unfoldr\ (h_{map}\ next)\ s)\ (unfoldr\ h_{eft}\ 1)) \\
&\quad \textbf{where} \\
&\qquad h_{sum}\ next\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\qquad\qquad\quad Nothing \quad\ \to 0 \\
&\qquad\qquad\qquad\quad Just\ (a, s') \to a + h_{sum}\ next\ s' \\
&\qquad h_{map}\ next\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\quad Nothing \quad\ \to Nothing \\
&\qquad\quad Just\ (x, ys) \to Just\ (x \uparrow 2, ys) \\
&\qquad h_{eft}\ i = \textbf{if}\ i > x \\
&\qquad\qquad\quad \textbf{then}\ Nothing \\
&\qquad\qquad\quad \textbf{else}\ Just\ (i, i + 1)
\end{aligned}$$

There is already one opportunity to apply the rewrite rule, yielding the following program (the **where** clauses are omitted for brevity):

$$sumSq_{du}\ x = destroy\ h_{sum}\ ((\lambda next\ s \to unfoldr\ (h_{map}\ next)\ s)\ h_{eft}\ 1)$$

Performing a beta reduction then yields

$$sumSq_{du}\, x = destroy\, h_{sum}\, (unfoldr\, (h_{map}\, h_{eft})\, 1)$$

This provides another opportunity to apply the rewrite rule:

$$sumSq_{du}\, x = h_{sum}\, (h_{map}\, h_{eft})\, 1$$

Now, h_{map}, as a nonrecursive function, can be inlined to

$$\begin{aligned} sumSq_{du}\, x = h_{sum}\, (\lambda s \to &\mathbf{case}\, h_{eft}\, s\, \mathbf{of} \\ &Nothing \to Nothing \\ &Just\, (x, ys) \to Just\, (x \uparrow 2, ys))\, 1 \end{aligned}$$

And likewise with h_{eft}

$$\begin{aligned} sumSq_{du}\, x = h_{sum}\, (\lambda s \to &\mathbf{case}\, (\mathbf{if}\, s > x \\ &\qquad\qquad\qquad\qquad \mathbf{then}\, Nothing \\ &\qquad\qquad\qquad\qquad \mathbf{else}\, Just\, (s, s+1))\, \mathbf{of} \\ &Nothing \to Nothing \\ &Just\, (x, ys) \to Just\, (x \uparrow 2, ys))\, 1 \end{aligned}$$

The **if** statement is then desugared into a **case** statement:

$$\begin{aligned} sumSq_{du}\, x = h_{sum}\, (\lambda s \to &\mathbf{case}\, (\mathbf{case}\, s > x\, \mathbf{of} \\ &True \to Nothing \\ &False \to Just\, (s, s+1))\, \mathbf{of} \\ &Nothing \to Nothing \\ &Just\, (x, ys) \to Just\, (x \uparrow 2, ys))\, 1 \end{aligned}$$

This leaves a nested **case** expression. The inner statement can be pushed through the outer using by the **case-of-case** transformation:[1]

$$\begin{aligned} sumSq_{du}\, x = h_{sum}\, (\lambda s \to &\mathbf{case}\, s > x\, \mathbf{of} \\ &True \to \mathbf{case}\, Nothing\, \mathbf{of} \\ &\qquad Nothing \to Nothing \\ &\qquad Just\, (x, ys) \to Just\, (x \uparrow 2, ys) \\ &False \to \mathbf{case}\, Just\, (s, s+1)\, \mathbf{of} \\ &\qquad Nothing \to Nothing \\ &\qquad Just\, (x, ys) \to Just\, (x \uparrow 2, ys))\, 1 \end{aligned}$$

[1]The transformations mentioned here are discussed in detail in Chapter 5

Now, there are two **case** statements where the expression being matched is known. This can be optimised using the **case-of-***known* transformation:

$$\begin{aligned}&sumSq_{du}\, x = h_{sum}\, (\lambda s \to \mathbf{case}\, s > x\, \mathbf{of}\\ &\quad True \to Nothing\\ &\quad False \to Just\, (s \uparrow 2, s + 1))\, 1\end{aligned}$$

The details of these transformations can be found in Peyton Jones and Santos [1998]. The h_{sum} function, being recursive, cannot be inlined. However, its definition is such that it will use its arguments to generate each member of the enumeration and then immediately consume it without creating any intermediate lists.

Like *foldr/build*, the correctness of *destroy/unfoldr* depends on a free theorem. The *destroy* function has type

$$destroy :: (\forall b.(b \to Maybe\, (a, b)) \to b \to c) \to [\, a] \to c$$

The rank-2 type of *destroy* demands that its first argument be a polymorphic function. Specifically, this function will take *any* stepper function and seed, as long as their types agree with each other. As it does not depend on the type of this seed, it is guaranteed to use the stepper function and seed in order to create a list though it may then do something with the resulting values. If *unfoldr* is used to create the data structure instead, it would do the same thing, collecting the elements in a list instead, however. Therefore, an intermediate data structure is removed by applying *destroy*'s first argument directly to the stepper function and seed. The free theorem and corresponding proof parallels those in *foldr/build*.

Although *destroy/unfoldr* is dual to *foldr/build*, it actually appears, at first glance, more expressive. Figure 2.4 shows not only *foldl* and *zip*, where *foldr/build* fusion fails to produce fully optimised programs, but also definitions of *foldr* and *filter*. While the definition of *foldr* can indeed produce efficient programs, the *filter* program, although defined in terms of *destroy* and *unfoldr*, will not fuse. As mentioned before, the key to being able to inline and optimise these functions is that they are *nonrecursive*. The h_{filter} function, however, must be recursive in order to correctly implement *filter*. This is because, just as *zip* is inherently an unfold, *filter* is inherently a fold. A 'cheat' using recursion allows us to achieve an unfold version, but it is not fusible using the transformation machinery that has been covered so far. Although *destroy/unfoldr* allows fusion of functions that *foldr/build* cannot fuse, it failed to deliver on the promise of being able to subsume *foldr/build* in terms of expressivity.

2.1.3 Stream fusion

The results of *foldr/build* and *destroy/unfoldr* are both a bit disappointing, because they both exclude some common list functions. At first glance, *destroy/unfoldr* has no such issues, because it is possible to define *filter* in terms of *destroy* and *unfoldr*, but the recursiveness that this introduces undoes all the effort to make these functions fusible in the first place.

Stream fusion addresses the issue of *filter*-like functions present in *destroy/unfoldr* fusion [Coutts et al., 2007a]. First, it introduces a new datatype

$$\begin{aligned}
&\textbf{data}\ \mathit{Stream}\ a = \forall s.\ \mathit{Stream}\ (s \to \mathit{Step}\ s\ a)\ s \\
&\textbf{data}\ \mathit{Step}\ s\ a = \mathit{Done} \\
&\qquad\qquad\quad \mid\ \mathit{Skip}\ s \\
&\qquad\qquad\quad \mid\ \mathit{Yield}\ a\ s
\end{aligned}$$

Note that this is not the usual use of the term 'stream', which often denotes type of definitely infinite lists. Instead, a *Stream* consists of a stepper function and an initial seed for that stepper function. The use of the existential type[2] expresses that, while *Stream*s are parameterised by the type of the elements that they produce, the seed type is independent of the type *Stream* produced. The type of the seed must, however, match the type that the stepper function expects.

So far, this looks rather like the stepper functions of *destroy/unfoldr*. The key difference is in the type of *Step*. In regular list unfolds, the stepper function has two cases, *Nothing*, signalling that no more elements will be yielded (corresponding to []), and a pair wrapped in a *Just*, signalling that an element has been yielded and there is more work to do. The constructors *Done* and *Yield* also cover these two cases, respectively. The new case, *Skip*, however, is crucial to fixing the *filter* problem, as will be shown.

The use of an explicit datatype makes for a cleaner presentation than in *foldr/build* and *destroy/unfoldr* by introducing some modularity; functions are written over streams, and then are composed with conversion functions to form a corresponding list function. To convert from a list to a stream, the function

$$\begin{aligned}
&\mathit{stream} :: [a] \to \mathit{Stream}\ a \\
&\mathit{stream}\ xs = \mathit{Stream}\ \mathit{next}\ xs \\
&\quad \textbf{where}
\end{aligned}$$

[2]Both existentially and universally quantified types are expressed using the *forall* keyword in GHC, and the typesetting matches this convention.

```
next [] = Done
next (x : xs) = Yield x xs
```

is used. This creates a *stream* whose stepper function breaks up a list into its head, which is the yielded element, and its tail, which is new seed. An empty list corresponds to a stream with no more elements left, so it is *Done*. A stream can be converted back into a list with

```
unstream :: Stream a → [a]
unstream (Stream next s) = unfold s
  where
    unfold s = case next s of
      Done      → []
      Skip s'   → unfold s'
      Yield a s' → a : unfold s'
```

The function *unfold* uses the stepper function and seed from the *Stream* to unfold a list, similarly to *unfoldr*, except the stepper function's return type is different. The cases *Done* and *Yield* are similar to their *unfoldr* counterparts, but in the *Skip* case, no value is yielded, only a seed. This seed can then be recursively unfolded. If a list is converted into a stream and back again, the original list results. In such a case, no *Skip*s appear. So where do they come from?

Recall that the problem with *filter* in *destroy/unfoldr* is that its stepper function had a recursive definition. This prevents it from being inlined, which means that it cannot be fused with other functions. It is possible to define the analogous over streams:

```
filter'_s :: (a → Bool) → Stream a → Stream a   -- Bad definition, does not fuse!
filter'_s p (Stream next s) = Stream next' s
  where
    next' s = case next s of
      Done   → Done
      Skip s' → Skip s'
      Yield a s' → if p a then Yield a s' else next' s'
```

This definition shows the general method of defining stream transformations. A new stepper function is defined that calls the old one, and then performs some transformation on the output. The *Skip* case is dealt with by simply passing the seeds on

unchanged, as they have no elements to transform. As in *destroy/unfoldr*, discarding elements is dealt with by a recursive call on the new seed, which stops any fusion from taking place. However, there is now a way discard an element: *Skip*. If recursive call is replaced with a *Skip* constructor

```
filter_s :: (a → Bool) → Stream a → Stream a   -- Correct definition.
filter_s p (Stream next s) = Stream next s
  where
    next' s = case next s of
      Done   → Done
      Skip s' → Skip s'
      Yield a s' → if p a then Yield a s' else Skip s'
```

Voilà! A nonrecursive definition of *filter*. The value is discarded, but productivity is maintained by using *Skip* s' to keep the seed even though the associated value is no longer wanted. A *Skip* constructor still results in a recursive call when the *Stream* is converted back into a list, but this recursion has been delayed until the list is unfolded in *unstream*, which is already responsible for recursively unfolding the *Stream* into a list.

Writing the list version *filter* is now simply a matter of composing the conversion functions with the *Stream* version to get the types right:

```
filter :: (a → Bool) → [a] → [a]
filter p = unstream ∘ filter'_s p ∘ stream
```

Generally speaking, a list producer will consist of a stream function followed by *unstream*. A list consumer will convert the list using *stream*, which is followed by a stream function. Transformers are both producers and consumers and therefore use both conversion functions. Since *unstream* signals list production, and *stream* signals list consumption, an intermediate data structure is created whenever *stream* ∘ *unstream* appears, leading to the rule

```
∀s. stream (unstream s) = s   -- Not sound
```

With the *stream* ∘ *unstream* pairs removed, the remaining stream functions can be fused, as they are nonrecursive.

Fusion of stream functions proceeds along the same lines as in *destroy/unfoldr*; functions are inlined to spot case-of-case and case-of-known transformations. The only

difference between the two approaches is presentation (by using an explicit datatype) and the introduction of an additional *Skip* case. While this allows for nonrecursive definitions of previously recursive functions, it has no impact on the mechanics of the low-level transformations.

Proving the correctness of stream fusion is, however, more complicated. Previous stream fusion systems relied on free theorems for correctness. Such an approach has its own caveats, which are discussed in Chapter 4. For stream fusion, the rewrite rule assumes that $stream \circ unstream = id$. This is obviously not true, as $stream\,(unstream\,\bot) = stream\,\bot = Stream\,next\,\bot \not\equiv \bot$. Furthermore, *Stream*s and lists are not isomorphic, as, due to *Skip*s, multiple *Stream*s correspond to the same list. The rewrite rule can be considered correct, through, for certain *Streams* and functions over them. The solution to this was simple: a library using stream fusion should export (correct) functions over *Stream*s as well as conversion functions, but prevent end users from constructing their own *Stream*s by only using the *Stream* constructors internally and not exporting them. Such a rather informal justification of correctness is rather unsatisfactory, and the correctness of stream fusion is further discussed in Chapters 3 and 4. Alternative approaches to proving shortcut fusion correctness are also discussed as related work in Chapter 6.

Having introduced the three fusion techniques that will be explored in this thesis, it is worth defining more precisely what it means for fusion to be 'shortcut'. Originally, shortcut fusion referred to *foldr/build* fusion, where the 'shortcut' taken had two characteristics. First, the problem was brought into a more tractable realm by fusing only specific functions of a certain, structured form whose recursive pattern was encapsulated in two combinators. The difficult work, dealing with recursive functions, was then reduced to simple, algebraic transformations, with the rest a matter of leaving the compiler to do its usual work. Second, the proof that this transformation was correct came 'for free' through the types of the combinators that encapsulated the structured recursion. With the proliferation of similar techniques, however, the name has come to refer to a class of fusion techniques, including those just discussed. While both aspects of the original shortcut also apply to *destroy/unfoldr* fusion, only the former applies to stream fusion. It is this mechanical similarity, i.e. fusing a form of structured recursion, using rewrite rules for the difficult part, and letting the compiler do the rest, that we use as the defining feature of shortcut fusion for the rest of this thesis. The goal for the rest of the thesis is to consider how these techniques are related and can be generalised to create a framework for describing shortcut fusion,

regardless. First, however, some theoretical concepts must be introduced that will form the toolkit used for this effort.

2.2 Folds and unfolds

As illustrated in the preceding sections, folds and unfolds provide a common pattern for defining functions over lists. These concepts also generalise to other recursively defined datatypes. One of the useful aspects of folds and unfolds is the extra reasoning power that comes with them, and category theory provides a useful setting for this reasoning. This will be leveraged in this thesis to provide a setting for reasoning about the fusion transformations discussed, which are based on these concepts. Therefore, this section provides an overview of folds as initial algebras and unfolds as final coalgebras along with Haskell code that parallels these developments.[3] In this overview, a basic knowledge of categories and functors is assumed.

2.2.1 Initial algebras and folds

In category theory, folds can be modelled using the concept of an *initial algebra*. An *algebra* for a category $\mathbb{C}$ and a functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$, is a pair (A, a), where A is an object in $\mathbb{C}$, called the *carrier* of the algebra, and $a : \mathsf{F}\,A \to A$, called the *action*. For another algebra (B, b), an *algebra homomorphism* $(A, a) \to (B, b)$ between them is an arrow $h : A \to B$ that makes the following diagram commute:

$$\begin{array}{ccccc} \mathsf{F}\,A & & \mathsf{F}\,A & \xrightarrow{\mathsf{F}\,h} & \mathsf{F}\,B \\ \downarrow{\scriptstyle a} & & \downarrow{\scriptstyle a} & & \downarrow{\scriptstyle b} \\ A & & A & \xrightarrow[h]{} & B \end{array} \qquad \begin{array}{c} \mathsf{F}\,B \\ \downarrow{\scriptstyle b} \\ B \end{array}$$

Together, the algebras and algebra homomorphisms of a functor F form a category $\mathbf{Alg}(\mathsf{F})$.

In Haskell, a functor is modelled as a datatype whose constructors describe the action of the functor on objects (i.e. Haskell values). The action on arrows is defined separately by making the datatype an instance of the *Functor* type class:

[3] When developing Haskell code examples, we make an assumption about the Haskell programs being written: It is assumed that they are total. This is not a particularly unusual assumption to make when reasoning about Haskell programs, but should be kept in mind when evaluating the applicability of our results.

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

As with functors in category theory, an instance of *fmap* should be implemented such that it preserves identity and composition, but this is not guaranteed by the types and therefore must be proven by the programmer. It is possible then to express an F-algebra as a type A and a function $a :: \mathsf{F}\,A \to A$. An algebra homomorphism $h : (A, a) \to (B, b)$ is a Haskell function $h :: A \to B$ that fulfils the required side condition for algebra homomorphisms.

If $\mathbf{Alg}(\mathsf{F})$ has an initial object, it is called the *initial algebra*, denoted $(\mu\mathsf{F}, in)$. As the initial object, it has the property that there is a *unique* arrow from it to any algebra. For the algebra (A, a), this arrow is called *fold a*, denoted $(\!|a|\!)$. As an algebra homomorphism, it makes the following diagram commute:

$$\begin{array}{ccccc}
\mathsf{F}\,(\mu\mathsf{F}) & & \mathsf{F}\,(\mu\mathsf{F}) & \xrightarrow{\mathsf{F}\,(\!|a|\!)} & \mathsf{F}\,A \\
\Big\downarrow{\scriptstyle \mathbf{in}} & & \Big\downarrow{\scriptstyle in} & & \Big\downarrow{\scriptstyle a} \\
\mu\mathsf{F} & & \mu\mathsf{F} & \xrightarrow[(\!|a|\!)]{} & A
\end{array}$$

The initiality of $\mu\mathsf{F}$, and therefore the uniqueness of $(\!|a|\!)$, is captured by the *universal property of folds*

$$h = (\!|a|\!) \quad \Longleftrightarrow \quad h \circ in = a \circ \mathsf{F}\,h \tag{2.1}$$

Initial algebras provide a semantics for recursive data structures. As an example, consider the functor $\mathsf{L}\,A\,B = 1 + A \times B$. The functor $\mathsf{L}\,A$ is called the *base functor* for lists with elements of type A. In Haskell, we can define the base functor as

```
data List a b = Nil | Cons a b
instance Functor (List a) where
  fmap f Nil         = Nil
  fmap f (Cons a b) = Cons a (f b)
```

where the type constructors define the action on objects, and *fmap* the action on arrows. (The notation of underlining datatypes that are used as base functors will be used throughout the thesis). This is similar to a datatype declaration for lists, but the recursive 'call' has been replaced by an additional type parameter. When expressing base functors in Haskell, sum types correspond to case analysis, where each summand is represented by a different data constructor, and product types dictate the arity of

each constructor, with one field per product operand. The initial algebra for this functor is the type $List\ a$, which is obtained by replacing the argument of the functor with a recursive call:

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

The corresponding action, in

$$\begin{aligned} &in :: \underline{List}\ a\ (List\ a) \to List\ a \\ &in\ \underline{Nil} &&= Nil \\ &in\ (\underline{Cons}\ x\ xs) &&= Cons\ x\ xs \end{aligned}$$

specifies how to construct a $List$ either from the final object $\underline{Nil}$, or from a pair consisting of an element and an already-constructed $List$.

The universal property (2.1) can be used to reason about a datatype that can modelled in this way. In addition to its raw form, it gives rise to three additional laws, which are called the computation, reflection, and fusion laws.

Computation Law Applying the left hand condition of the universal property to the right by setting $h := (\!|a|\!)$ yields the *computation law*

$$(\!|a|\!) \circ in = a \circ \mathsf{F}\ (\!|a|\!) \tag{2.2}$$

which can be seen as a definition for $(\!|a|\!)$. This law is also evident in the commutative diagram above. The algebra a is used to combine the head of a list with a recursive call over the tail of the list, the placement of which is determined by the action of the functors on arrows or, in Haskell, the definition of $fmap$:

$$\begin{aligned} &fold :: (\underline{List}\ a\ b \to b) \to List\ a \to b \\ &fold\ a\ Nil &&= (a \circ fmap\ (fold\ a))\ \underline{Nil} \\ &fold\ a\ (Cons\ x\ xs) &&= (a \circ fmap\ (fold\ a))\ (\underline{Cons}\ x\ xs) \end{aligned}$$

Thus, the base functor determines the shape of a data structure by its action on objects (Haskell values), and the recursion pattern over it by its action on arrows (Haskell functions). This recursion pattern provides a method of combining the elements of a data structure using an algebra, which is just a function of type $\underline{List}\ a\ b \to b$ where a is the type of the elements of the list and b is the type of value obtained from combining the elements.

While this definition illustrates the relationship between the base functor and the recursive pattern clearly, a more idiomatically Haskell style is to place the recursive call explicitly:

$$
\begin{aligned}
&\mathit{fold'} :: (\underline{\mathit{List}}\ a\ b \to b) \to \mathit{List}\ a \to b \\
&\mathit{fold'}\ a\ \mathit{Nil} &&= a\ \underline{\mathit{Nil}} \\
&\mathit{fold'}\ a\ (\mathit{Cons}\ x\ xs) &&= a\ (\underline{\mathit{Cons}}\ x\ (\mathit{fold'}\ a\ xs))
\end{aligned}
$$

The algebra contains a case for each constructor. The traditional definition of *foldr* handles this using separate arguments, one for each constructor being replaced by the fold. The relationship between these two versions can be demonstrated by defining *foldr* in terms of *fold′*:

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to \mathit{List}\ a \to b \\
&\mathit{foldr}\ c\ n\ xs = \mathit{fold'}\ a\ xs \\
&\quad \textbf{where} \\
&\qquad a\ (\underline{\mathit{Cons}}\ x\ y) = c\ x\ y \\
&\qquad a\ \underline{\mathit{Nil}} \qquad\quad = n
\end{aligned}
$$

Reflection Law Setting $h := id$ and $a := in$ yields an extensionality property, which states that folding the initial algebra over a data structure results in the same structure since the data constructors are being replaced with themselves. This property is called the *reflection law*

$$(\!|in|\!) = id \tag{2.3}$$

The correspondence between the reflection law and the universal property can be more clearly demonstrated diagrammatically:

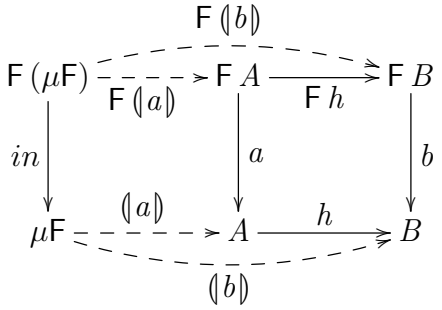
$$
\begin{array}{ccc}
\mathsf{F}\,(\mu\mathsf{F}) & \xrightarrow{\mathsf{F}\,(\!|in|\!)} & \mathsf{F}\,(\mu\mathsf{F}) \\
{\scriptstyle in}\downarrow & & \downarrow{\scriptstyle in} \\
\mu\mathsf{F} & \xrightarrow[(\!|in|\!)]{id} & \mu\mathsf{F}
\end{array}
$$

Fusion Law Finally, folds come with a *fusion law*. As its name hints, this law describes a similar transformation to those being discussed in this thesis. Specifically,

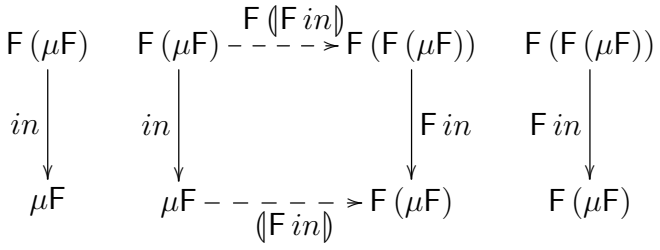
it allows a fold to be combined with another by 'absorbing' a function that follows it under certain conditions:

$$h \circ (\!|a|\!) = (\!|b|\!) \quad \Longleftarrow \quad h \circ a = b \circ Fh \tag{2.4}$$

This allows a program that consumes a data structure using a fold, but then performs some transformation h over the resulting value, to be replaced with a program consisting of a single fold, assuming that the transformation h satisfies the side condition. Expressed diagrammatically, this side condition can be seen in the right subdiagram below, where h must be an algebra homomorphism from A to B, where the latter is the carrier of the algebra used in the new fold:



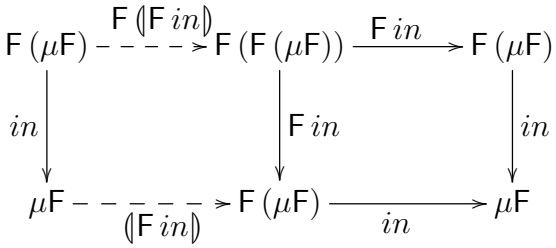
The notation $\mu\mathsf{F}$ is used for the carrier of an initial algebra because it is the least fixed point of F, i.e. it is the 'least' solution of the equation $\mathsf{F}(X) \cong X$.[4] This is known as Lambek's Lemma [Lambek, 1968]. One direction of this isomorphism is given by in, and the other by the inverse $(\!|\mathsf{F}\, in|\!)$. This can be demonstrated by leveraging the fact that $(\mathsf{F}(\mu\mathsf{F}), \mathsf{F}\, in)$ is an F-algebra:



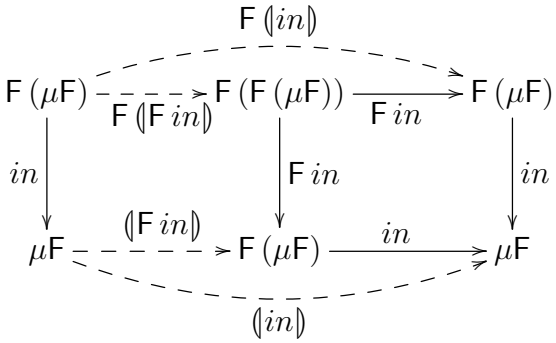
This means that there is a unique arrow from $\mu\mathsf{F}$ to $\mathsf{F}(\mu\mathsf{F})$, but does not prove that it is a witness to the isomorphism. There is, however, also an algebra homomorphism

[4]The term 'least' implies an ordering, and is interpreted differently in different settings. In *Set*, for example, 'least' refers to subset ordering.

from $(\mathsf{F}\,(\mu\mathsf{F}), \mathsf{F}\,in)$ to $(\mu\mathsf{F}, in)$, which is in itself:



Furthermore, due to the initiality of $(\mu\mathsf{F}, in)$, there also exists a unique arrow directly from $\mu\mathsf{F}$ to $\mu\mathsf{F}$, which is $(\!|in|\!) = id$. This is an instance of the fusion law, where in fulfils the necessary side condition.



The bottom of this diagram gives the equations $in \circ (\!|\mathsf{F}\,in|\!) = (\!|in|\!)$, and due to the uniqueness of $(\!|in|\!)$, it must be equal to id (as stated by the reflection law). The left square gives the other direction $(\!|\mathsf{F}\,in|\!) \circ in = \mathsf{F}\,in \circ \mathsf{F}\,(\!|\mathsf{F}\,in|\!) = \mathsf{F}\,(in \circ (\!|\mathsf{F}\,in|\!)) = \mathsf{F}\,id = id$.

Lambek's Lemma allows us to give an even more general view of recursive datatypes and folds in Haskell by using the least fixed point operator

$$\textbf{newtype}\,\mu f = In\,\{in^\circ :: f\,(\mu f)\}$$

The In constructor, having type $f\,(\mu f) \to \mu f$, constructs a recursive data structure, as its name would suggest. The in° destructor is the inverse, breaking up a recursive data structure into its constituent parts. Together, they represent the witness of the isomorphism $\mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F}$. It is now possible to define lists, for example, as the least fixed point of the $\underline{List}\;a$ functor:

$$\textbf{type}\,List\;a = \mu(\underline{List}\;a)$$

In this case, In works similarly to the function defined above, taking either $\underline{Nil}$, or an element and a list wrapped up in a $\underline{Cons}$. In the latter case, it enforces the recursiveness of the structure by requiring that the second argument of the constructor is

another $\mu\underline{List}\ a$, whereas this was accomplished in the previous definition by swapping from $\underline{Cons}$ to $Cons$ constructors. It is also possible to give a general definition of fold for recursive datatypes defined in this manner:

$$(\!|-|\!) :: Functor\, f \Rightarrow (f\ a \to a) \to \mu f \to a$$
$$(\!|a|\!) = a \circ fmap\, (\!|a|\!) \circ in^\circ$$

This definition more transparently corresponds to the definition given by the computation law, except the in on the left hand side of the equation is swapped for in° on the right hand side. Lambek's Lemma also possesses a dual, which will be revealed in Section 2.2.2 and will be utilised in Section 2.3.

2.2.2 Final coalgebras and unfolds

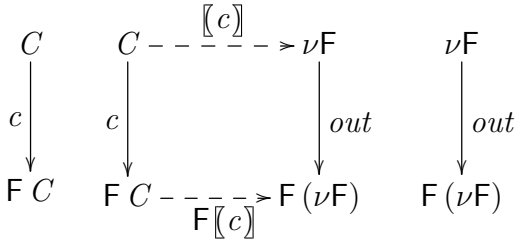
In the previous section, initial algebras were revealed as a way to model folds. Dual to initial algebras are *final coalgebras*, which give another semantics by modelling unfolds. A coalgebra for a category $\mathbb{C}$ and functor $\mathsf{F}{:}\mathbb{C} \to \mathbb{C}$ is a pair (C, c) consisting of an object C in $\mathbb{C}$ and an arrow $c : C \to \mathsf{F}\, C$. Given another coalgebra (D, d), a *coalgebra homomorphism* $(C, c) \to (D, d)$ is an arrow $h : C \to D$ that makes the following diagram commute:

$$\begin{array}{ccc} C & \xrightarrow{h} & D \\ \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle d} \\ \mathsf{F}\, C & \xrightarrow[\mathsf{F}\, h]{} & \mathsf{F}\, D \end{array} \qquad \begin{array}{c} D \\ \downarrow{\scriptstyle d} \\ \mathsf{F}\, D \end{array}$$

The coalgebras and coalgebra homomorphisms of a functor F also form a category called $\mathbf{Coalg}(\mathsf{F})$. Similarly to initial algebras, an F-coalgebra can be expressed in Haskell as a type C and an accompanying function $c :: \mathsf{F}\, C \to C$. A coalgebra homomorphism $h : (C, c) \to (D, d)$ is then a function $h :: C \to D$ satisfying the aforementioned condition.

If it exists, the final object of $\mathbf{Coalg}(\mathsf{F})$ is called the *final coalgebra* and is denoted $(out, \nu\mathsf{F})$. Finality is the dual property of initiality, i.e. that there exists a unique arrow *to* it from every coalgebra. For a coalgebra (C, c), this arrow is called *unfold*

c, denoted $[\![c]\!]$, and makes the following diagram commute:



The finality of $(out, \nu\mathsf{F})$ is captured in the *universal property of unfolds*:

$$h = [\![c]\!] \quad \Longleftrightarrow \quad out \circ h = \mathsf{F}\, h \circ c \tag{2.5}$$

Unfolds can be used to model functions that produce a recursive datatype. As an example, the $\underline{List}\ a$ base functor can be used again, and the carrier of the final coalgebra for this functor is $List\ a$. The corresponding function, however, describes how to *destruct* a list instead of construct it

$$\begin{aligned} &out :: List\ a \to \underline{List}\ a\ (List\ a) \\ &out\ Nil \qquad\quad = \underline{Nil} \\ &out\ (Cons\ x\ xs) = \underline{Cons}\ x\ xs \end{aligned}$$

According to this definition, the empty list is deconstructed into the corresponding constructor $\underline{Nil}$, while a non-empty list is broken up into a pair consisting of the head of the list x and the tail xs.

From the universal property of unfolds come the same three laws as those for folds.

Computation Law The computation law can be seen as a datatype-generic definition of unfold

$$out \circ [\![c]\!] = \mathsf{F}[\![c]\!] \circ c \tag{2.6}$$

As seen on the right hand side of the equation, a coalgebra takes a 'seed' value and produces an F-shaped structure containing data and one or more new seeds of type b over which the recursive call is mapped. In the case of the $\underline{List}$ functor, a coalgebra $c : b \to \underline{List}\ a\ b$ takes a seed value and yields an element of the list and a new seed value, which can be used to produce the tail of the list. For $List\ a$, this has the definition

$$\begin{aligned} &unfold :: (s \to \underline{List}\ a\ s) \to s \to List\ a \\ &unfold\ c\ s = \mathbf{case}\ c\ s\ \mathbf{of} \end{aligned}$$

$$\underline{Nil} \qquad \to Nil$$
$$\underline{Cons}\ x\ s' \to Cons\ x\ (unfold\ c\ s')$$

Here, the recursive call is already placed explicitly. To build a list out of the successively yielded elements, the results of the unfold are collected in a *List a*.

Reflection Law Substituting $h := id$ and $c := out$ shows that successively yielding elements of a data structure and collecting the results yields the same data structure again. Therefore, the reflection law for data unfolds is

$$[\![out]\!] = id \tag{2.7}$$

or, diagrammatically

$$\begin{array}{ccc} \nu\mathsf{F} & \xrightarrow[id]{[\![out]\!]} & \nu\mathsf{F} \\ {\scriptstyle out}\downarrow & & \downarrow{\scriptstyle out} \\ \mathsf{F}\,(\nu\mathsf{F}) & \xrightarrow[\mathsf{F}\,[\![out]\!]]{\mathsf{F}\,id} & \mathsf{F}\,(\nu\mathsf{F}) \end{array}$$

Fusion Law There is also a fusion law for unfolds, which allows it to absorb a function to the right of it:

$$[\![c]\!] = [\![d]\!] \circ h \Longleftarrow \mathsf{F}\,h \circ c = d \circ h \tag{2.8}$$

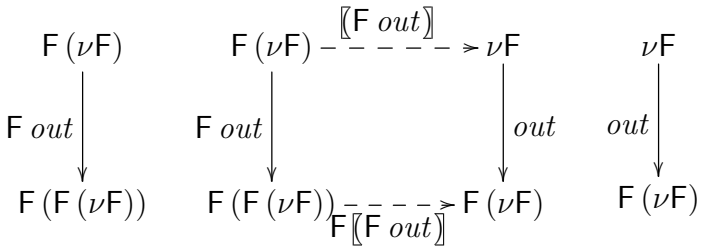
Dual to the fusion law for folds, this allows a program that unfolds a structure using a seed transformed by h to be replaced by a new program consisting of single unfold, assuming h fulfils the appropriate side condition. As shown by the diagram below, this requires that h be a coalgebra homomorphism from (C, c) to (D, d):

$$\begin{array}{ccccc} & & \mathsf{F}\,[\![c]\!] & & \\ & \overset{}{\dashrightarrow} & & \overset{}{\dashrightarrow} & \\ C & \xrightarrow[h]{} & D & \xrightarrow[[\![d]\!]]{} & \nu\mathsf{F} \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle out} \\ \mathsf{F}\,C & \xrightarrow{\mathsf{F}\,h} & \mathsf{F}\,D & \xrightarrow{\mathsf{F}\,[\![d]\!]} & \mathsf{F}\,(\nu\mathsf{F}) \\ & \underset{}{\dashrightarrow} & & \underset{}{\dashrightarrow} & \\ & & [\![c]\!] & & \end{array}$$

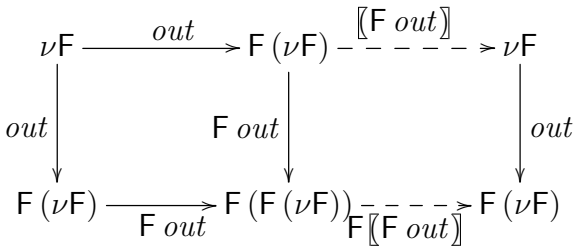
Dual to Lambek's Lemma, the notation $\nu\mathsf{F}$ denotes that $\nu\mathsf{F}$ is also a fixed point of the equation $\mathsf{F}\,(X) \cong X$. In this instance, $\nu\mathsf{F}$ is the *greatest* fixed point.[5] The

[5] In the same manner as the least fixed point, the interpretation of 'greatest' is dependent upon setting.

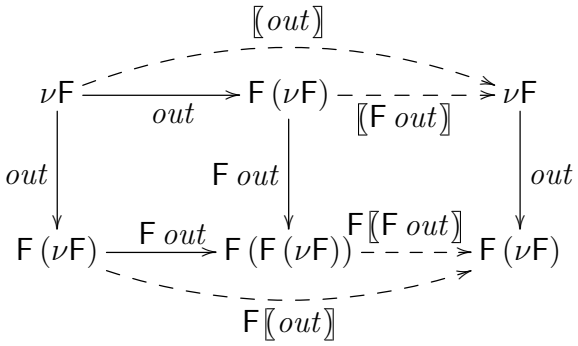
coalgebra *out* provides half of the witness of this isomorphism. The other half, in accordance with the duality of algebras and coalgebras, begins with establishing that $(\mathsf{F}\,(\nu\mathsf{F}), \mathsf{F}\,out)$ is a coalgebra



There also exists a coalgebra homomorphism $(out, \nu\mathsf{F}) \to (\mathsf{F}\,out, \mathsf{F}\,(\nu\mathsf{F}))$



The finality of $\nu\mathsf{F}$ guarantees that there is then a unique arrow $\nu\mathsf{F}$ to itself, $[\![out]\!]$:



The isomorphism is then again obvious, with the top line of the diagram showing one direction and the right square showing the other.

The greatest fixed point functor can similarly be defined in Haskell:

$$\mathbf{newtype}\,\nu f = Out^{\circ}\,\{\,out :: f\,(\nu f)\,\}$$

along with a generalised definition of unfolds:

$$[\![-]\!] :: Functor\,f \Rightarrow (c \to f\;c) \to c \to \nu f$$
$$[\![c]\!] = Out^{\circ} \circ fmap\,[\![c]\!] \circ c$$

Note that the definition of $\nu-$ is the same as that for $\mu-$, but the view is different, which is reflected in the names. Here, *out* corresponds to the destructor of the final coalgebra, while Out° is the other direction of the isomorphism.

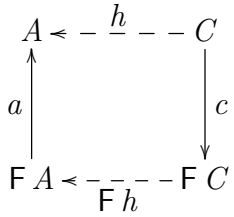
In Haskell, the distinction between these two objects, $\mu\mathsf{F}$ and $\nu\mathsf{F}$, is not obvious, because Haskell has the property of being *algebraically compact* [Freyd, 1992], meaning that $\mu\mathsf{F} \cong \nu\mathsf{F}$. This is why the programmer can freely mix folds and unfolds over the same data structure in Haskell. As demonstrated in Section 2.1, however, this distinction comes to the fore when a fusion technique requires all functions to be either folds or unfolds. In general, the data structures given by greatest fixed points and least fixed points do not necessarily coincide. For example, in *Set*, the category of sets and total functions, the least fixed point $\mu\underline{List}\ a$ is the set of all *finite* lists of elements of type a, while $\nu\underline{List}\ a$ also includes *infinite* lists. This means that folds and unfolds are generally incompatible and must be modelled in two different settings. The following section, however, will present an overview of a setting in which fold and unfolds can coexist.

2.3 Recursive coalgebras and hylomorphisms

In the previous sections, initial algebras and final coalgebras were introduced as useful concepts for modelling folds and unfolds, respectively. Unfortunately, this requires that dealing with folds and unfolds in different settings, because the carriers of initial algebras and final coalgebras do not necessarily coincide. This limits the comparisons that be can be made between the fusion techniques discussed, despite the fact that they are closely related. This section will introduce *recursive* coalgebras, a setting in which folds and unfolds can coexist. In subsequent chapters, this setting will be used bring the fusion transformations discussed in Section 2.1 together 'under the same roof'. This section follows the work of Capretta et al. [2006], who motivate the use of hylomorphisms based on recursive coalgebras as a structured recursion scheme. Where appropriate, these developments will be accompanied by examples in Haskell.

A coalgebra (C, c) is called *recursive* if for *every* algebra (A, a) the equation in the unknown $h : A \leftarrow C$,

$$h \;=\; a \circ \mathsf{F}\, h \circ c \;, \tag{2.9}$$



has a *unique* solution. The equation captures the *divide-and-conquer* pattern of computation: a problem is divided into subproblems by (c), the subproblems are solved recursively $(\mathsf{F}\,h)$, and finally the subsolutions are combined into a single solution (a). The uniquely defined function h is called a *hylomorphism* or *hylo* for short and is written $(\!|a \leftarrow c|\!)_{\mathsf{F}} : A \leftarrow C$.[6] The notation is meant to capture the idea that h is a coalgebra-to-algebra morphism, with the reversed arrow in the signature $A \leftarrow C$ mirroring this notation. Such notation will also be used for algebra and coalgebra homomorphisms where appropriate. The subscripted functor name is omitted if it is obvious from the context. The uniqueness of h is captured by the following property:

$$h = (\!|a \leftarrow c|\!) \quad \Longleftrightarrow \quad h = a \circ \mathsf{F}\,h \circ c \tag{2.10}$$

This can be interpreted as a definition of $(\!|- \leftarrow -|\!)$ in Haskell, where it becomes a function that takes an algebra and a recursive coalgebra as arguments and returns resulting hylo:

$$\begin{aligned}&(\!|- \leftarrow -|\!) :: (\mathit{Functor}\,f) \Rightarrow (f\ a \to a) \to (c \to f\ c) \to (c \to a)\\ &(\!|a \leftarrow c|\!) = a \circ \mathit{fmap}\,(\!|a \leftarrow c|\!) \circ c\end{aligned}$$

Note that the type of this function does not guarantee that c is a *recursive* coalgebra and therefore does not guarantee that the resulting hylo has a unique solution; the programmer needs to discharge this obligation by some other means.

The category of recursive coalgebras and coalgebra homomorphisms forms a full subcategory of $\mathbf{Coalg}(\mathsf{F})$, called $\mathbf{Rec}(\mathsf{F})$. If $\mathbf{Rec}(\mathsf{F})$ has a final object $(\mathit{out}, \mathsf{F})$, then there is a unique arrow from any other *recursive* coalgebra (C, c) to $(\mathit{out}, \mathsf{F})$. By abuse of notation, this arrow is called *unfold*, written $[\!(c)\!] : c \to \mathit{out}$, as in $\mathbf{Coalg}(\mathsf{F})$. Finality is captured by the following uniqueness property:

$$h = [\!(c)\!] \quad \Longleftrightarrow \quad h : c \to \mathit{out} \quad \Longleftrightarrow \quad \mathsf{F}\,h \circ c = \mathit{out} \circ h \tag{2.11}$$

(The notation $h : a \to b$ denotes a (co)algebra homomorphism, where a and b are (co)algebras i.e. the carriers of those algebras have been elided). This is the usual

[6] In this thesis, 'hylo' refers to unique solutions to (2.9), as opposed to Meijer's use when he introduced the scheme [Meijer et al., 1991], where it referred to canonical solutions, which were assumed to exist in functional programming languages.

property of unfolds, as shown in Section 2.2.2, although this time for $\mathbf{Rec}(\mathsf{F})$ instead of $\mathbf{Coalg}(\mathsf{F})$. Similarly, the development of the computation, reflection, and fusion laws from Section 2.2.2 also follow from this law in the same manner, but in this case restricted to recursive coalgebras. The same applies to the corresponding Haskell code, which remains unchanged. This is because the recursiveness of a coalgebra is not encapsulated by the definition of $[\![-]\!]$, nor in the representation of coalgebras as functions. Now, however, the structure of $\mathbf{Rec}(\mathsf{F})$ will be explored, which will reveal its usefulness as a setting in this thesis.

It was shown in Section 2.2.1 that *in* has an inverse according Lambek's Lemma, and that this dualises to unfolds in Section 2.2.2. The invertibility of *out* can also be proven for final recursive coalgebras.

Lemma 2.3.1. *A recursive coalgebra is final if and only if it is invertible: (1) If* (out, F) *is the final recursive coalgebra, then out is invertible with* $out^\circ = [\![\mathsf{F}\ out]\!]$. *(2) If* (C, c) *is a recursive coalgebra and* c *is invertible, then* (C, c) *is final. Furthermore,* $[\![d]\!] = (\!|c^\circ \leftarrow d|\!)$.

The proof progresses along the same lines as in $\mathbf{Coalg}(\mathsf{F})$. Therefore, it must first be proved that:

Lemma 2.3.2. *If* (C, c) *is recursive* F*-coalgebra, then* $(\mathsf{F}\ c, \mathsf{F}\ C)$ *is as well.*

Proof. This means that $h = a \circ \mathsf{F}\ h \circ \mathsf{F}\ c$ must have a solution, and furthermore, it must be unique. These two requirements can be proved concurrently:

$$
\begin{aligned}
& h = a \circ \mathsf{F}\ h \circ \mathsf{F}\ c \\
\iff & \quad \{\ \mathsf{F} \text{ functor } \} \\
& h = a \circ \mathsf{F}\ (h \circ c) \\
\iff & \quad \{\ \text{logic and } f = g \implies f \circ c = g \circ c\ \} \\
& h = a \circ \mathsf{F}\ (h \circ c) \quad \text{and} \quad h \circ c = a \circ \mathsf{F}\ (h \circ c) \circ c \\
\iff & \quad \{\ \text{uniqueness property (2.9) and assumption: } c \text{ is recursive } \} \\
& h = a \circ \mathsf{F}\ (h \circ c) \quad \text{and} \quad h \circ c = (\!|a \leftarrow c|\!) \\
\iff & \quad \{\ \text{Leibniz } \} \\
& h = a \circ \mathsf{F}\ (\!|a \leftarrow c|\!) \quad \text{and} \quad h \circ c = (\!|a \leftarrow c|\!) \\
\iff & \quad \{\ \text{computation (2.13)} \ \} \\
& h = a \circ \mathsf{F}\ (\!|a \leftarrow c|\!) \quad \text{and} \quad h \circ c = a \circ \mathsf{F}\ (\!|a \leftarrow c|\!) \circ c \\
\iff & \quad \{\ \text{logic and } f = g \implies f \circ c = g \circ c\ \} \\
& h = a \circ \mathsf{F}\ (\!|a \leftarrow c|\!) \qquad \square
\end{aligned}
$$

□

Proof of Lemma 2.3.1. 1. First of all, $[\![\mathsf{F}\ out]\!]$ is well-defined, since $\mathsf{F}\ out$ is recursive, by Lemma 2.3.2. To establish the isomorphism, we must show that $id = [\![\mathsf{F}\ out]\!] \circ out$.

$$\begin{array}{ll} & id = [\![\mathsf{F}\ out]\!] \circ out \\ \iff & \{\ \text{unfold reflection}\ \} \\ & [\![out]\!] = [\![\mathsf{F}\ out]\!] \circ out \\ \Longleftarrow & \{\ \text{unfold fusion}\ \} \\ & out : out \to \mathsf{F}\ out\ \ , \end{array}$$

and $id = out \circ [\![\mathsf{F}\ out]\!]$,

$$\begin{array}{ll} & out \circ [\![\mathsf{F}\ out]\!] \\ = & \{\ \text{unfold computation}\ \} \\ & \mathsf{F}\,[\![\mathsf{F}\ out]\!] \circ \mathsf{F}\ out \\ = & \{\ \mathsf{F}\ \text{functor}\ \} \\ & \mathsf{F}\,([\![\mathsf{F}\ out]\!] \circ out) \\ = & \{\ \text{see above}\ \} \\ & \mathsf{F}\ id \\ = & \{\ \mathsf{F}\ \text{functor}\ \} \\ & id\ \ . \end{array}$$

2. We have to prove the uniqueness property of unfolds (2.11) with $out := c$ and $[\![d]\!] := (\!|c^\circ \leftarrow d|\!)$.

$$\begin{array}{ll} & h = (\!|c^\circ \leftarrow d|\!) \iff \mathsf{F}\ h \circ d = c \circ h \\ \iff & \{\ \text{inverses}\ \} \\ & h = (\!|c^\circ \leftarrow d|\!) \iff c^\circ \circ \mathsf{F}\ h \circ d = h \end{array}$$

The latter equivalence is an instance of the uniqueness property of hylos (2.10).
□

The definition of a hylomorphism does not assume that the initial F-algebra exists. The powerset functor, for instance, admits no fixed points, yet we may want to divide a problem into a *set* of sub-problems. However, if the initial algebra exists, then

it coincides with the final recursive coalgebra and, furthermore, folds and unfolds emerge as special cases of hylos. We can state this more formally:

Theorem 2.3.3. *Initial* F*-algebras and final recursive* F*-coalgebras coincide:*

(1) If (out, C) *is the final recursive* F*-coalgebra, then* (C, out°) *is the initial* F*-algebra. Furthermore,* $(\!|a|\!) = (\!|a \leftarrow out|\!)$*.*

(2) If (A, in) *is the initial* F*-algebra, then* (in°, A) *is the final recursive* F*-coalgebra. Furthermore,* $[\![c]\!] = (\!|in \leftarrow c|\!)$*.*

Proof of Theorem 2.3.3. 1. By Lemma 2.3.1–(1) *out* has an inverse. We have to prove the uniqueness property of folds (2.1) with $in := out^\circ$ and $(\!|a|\!) := (\!|a \leftarrow out|\!)$.

$$\begin{aligned} & h = (\!|a \leftarrow out|\!) \iff h \circ out^\circ = a \circ \mathsf{F}\, h \\ \iff & \quad \{ \text{ inverses } \} \\ & h = (\!|a \leftarrow out|\!) \iff h = a \circ \mathsf{F}\, h \circ out \end{aligned}$$

The latter equivalence is an instance of the uniqueness property of hylos (2.10).

2. We first have to show that (in°, A) is recursive.

$$\begin{aligned} & h = a \circ \mathsf{F}\, h \circ in^\circ \\ \iff & \quad \{ \text{ inverses } \} \\ & h \circ in = a \circ \mathsf{F}\, h \\ \iff & \quad \{ \text{ uniqueness property of folds (2.1) } \} \\ & h = (\!|a|\!) \end{aligned}$$

The statement then follows from Lemma 2.3.1–(2) with $c := in^\circ$.

□

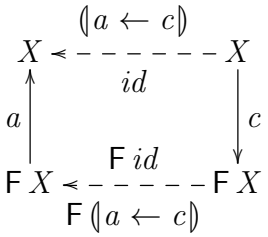
Theorem 2.3.3 allows us to treat folds and unfolds in the same setting—note that an unfold produces an element of an initial algebra! This allows for reasoning without depending on algebraic compactness of the ambient category. Although this is the usual approach, the downside is that the hylo equation (2.9) only has a canonical, least solution, not a unique solution, so (2.10) does not hold.

2.3.1 Calculational Properties

Having introduced the setting of recursive coalgebras, this section will now show the calculational properties that emerge in such a setting. The reasoning power given by these properties will be used in subsequent chapters. The uniqueness property of hylos has already been mentioned. As a consequence of it, hylos also have an identity law and a computation law, in addition to *three* fusion laws.

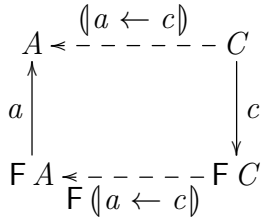
Identity Law Setting $h := id$ in the universal property (2.10), yields the *identity law*

$$(\!|a \leftarrow c|\!) = id \quad \Longleftrightarrow \quad a \circ c = id \ . \tag{2.12}$$



Computation Law Substituting the left-hand side of the uniqueness property into the right-hand side, yields the *computation law*:

$$(\!|a \leftarrow c|\!) \ = \ a \circ \mathsf{F}\,(\!|a \leftarrow c|\!) \circ c \ . \tag{2.13}$$



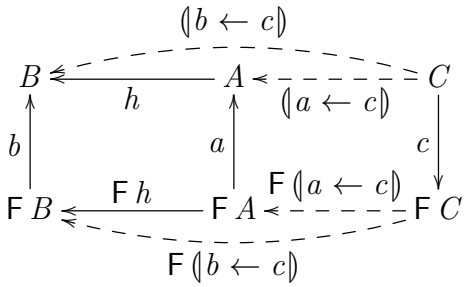
The Haskell code given earlier corresponds directly to this definition.

Hylomorphisms also come with *three* fusion laws: algebra fusion, coalgebra fusion, and composition.

Algebra fusion An algebra homomorphism after a hylo can be absorbed to form a single hylo:

$$h \circ (\!|a \leftarrow c|\!) = (\!|b \leftarrow c|\!) \quad \Longleftarrow \quad h : a \to b \tag{2.14}$$

The side condition is the same as that for the fold fusion law, and the the corresponding diagram is very similar:



For the proof we appeal to the uniqueness property.

$$
\begin{aligned}
& h \circ (\!|\, a \leftarrow c \,|\!) = (\!|\, b \leftarrow c \,|\!) \\
\iff \quad & \{ \text{ uniqueness property of hylos (2.10) } \} \\
& h \circ (\!|\, a \leftarrow c \,|\!) = b \circ \mathsf{F}\,(h \circ (\!|\, a \leftarrow c \,|\!)) \circ c
\end{aligned}
$$

The obligation is discharged as follows:

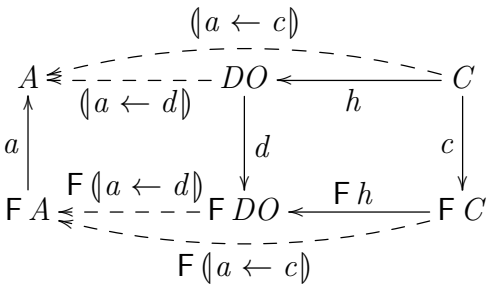
$$
\begin{aligned}
& h \circ (\!|\, a \leftarrow c \,|\!) \\
= \quad & \{ \text{ hylo computation (2.13) } \} \\
& h \circ a \circ \mathsf{F}\,(\!|\, a \leftarrow c \,|\!) \circ c \\
= \quad & \{ \text{ assumption: } h : a \to b \ \} \\
& b \circ \mathsf{F}\, h \circ \mathsf{F}\,(\!|\, a \leftarrow c \,|\!) \circ c \\
= \quad & \{ \ \mathsf{F} \text{ functor } \} \\
& b \circ \mathsf{F}\,(h \circ (\!|\, a \leftarrow c \,|\!)) \circ c \ .
\end{aligned}
$$

As an aside, in the calculation the coalgebra c is totally passive.

Coalgebra fusion Dually, a coalgebra homomorphism *before* a hylo can be absorbed to form a single hylo:

$$
(\!|\, a \leftarrow c \,|\!) = (\!|\, a \leftarrow d \,|\!) \circ h \quad \Longleftarrow \quad h : c \to d \tag{2.15}
$$

Again, the diagram is very similar to that of the unfold fusion law:

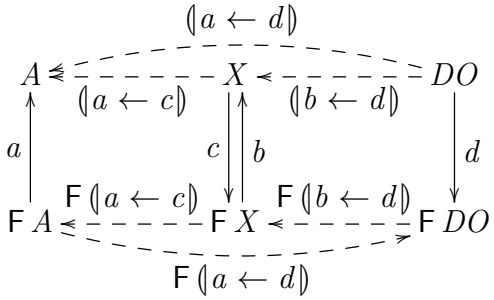


Like the law, the proof is the dual of that for algebra fusion.

Composition Law A composition of hylos can be merged into a single one if the coalgebra of the hylo on the left inverts the algebra of the right hylo:

$$(\!|a \leftarrow c|\!) \circ (\!|b \leftarrow d|\!) = (\!|a \leftarrow d|\!) \quad \Longleftarrow \quad c \circ b = id \tag{2.16}$$

The following diagram illustrates the composition visually, as it is the composition of two hylo diagrams:



Composition is, in fact, a simple consequence of algebra fusion as the hylomorphism $(\!|a \leftarrow c|\!) : b \to a$ is simultaneously an F-algebra homomorphism:

$$\begin{array}{ll} & (\!|a \leftarrow c|\!) \circ b \\ = & \quad \{ \text{ hylo computation (2.13) } \} \\ & a \circ \mathsf{F}\,(\!|a \leftarrow c|\!) \circ c \circ b \\ = & \quad \{ \text{ assumption: } c \circ b = id \ \} \\ & a \circ \mathsf{F}\,(\!|a \leftarrow c|\!) \end{array}$$

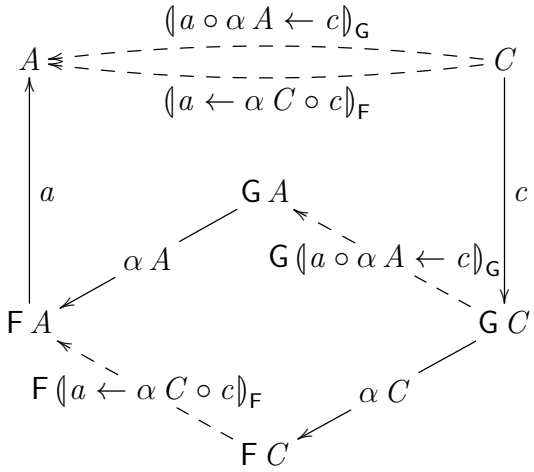
Alternatively, the law can be derived from coalgebra fusion by showing that $(\!|b \leftarrow d|\!) : d \to c$ is an F-coalgebra homomorphism.

Hylo-shift Law or Base Change Law For a natural transformation $\alpha : \mathsf{G} \dot{\to} \mathsf{F}$:

$$(\!|a \circ \alpha\, A \leftarrow c|\!)_{\mathsf{G}} \;=\; (\!|a \leftarrow \alpha\, C \circ c|\!)_{\mathsf{F}} \tag{2.17}$$

Like the diagram for the composition law, the following diagram also comprises two hylo diagrams. However, in this case they are intertwined with the natural transfor-

mation α. (The naturality square has been turned into a naturality diamond.)



In fact, the statement can be strengthened: if c is recursive, then $\alpha\, C \circ c$ is recursive, as well.

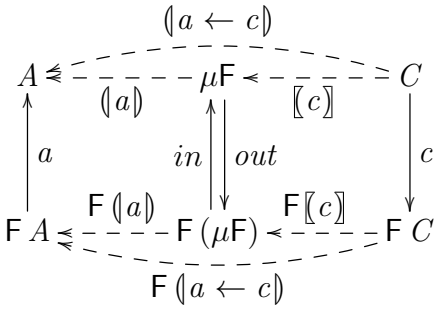
$$
\begin{array}{ll}
 & h = a \circ \mathsf{F}\, h \circ \alpha\, C \circ c \\
\iff & \quad \{\ \alpha \text{ natural}\ \} \\
 & h = a \circ \alpha\, A \circ \mathsf{G}\, h \circ c \\
\iff & \quad \{\ \text{uniqueness property of hylos (2.10)}\ \} \\
 & h = (\!|a \circ \alpha\, A \leftarrow c|\!)_{\mathsf{G}}
\end{array}
$$

It is worth pointing out that the laws stated thus far are independent of the existence of initial algebras. Only the following law makes this assumption.

Fold/Unfold Law A fold after an unfold can be fused together to form a hylo.

$$(\!|a|\!) \circ [\![c]\!] \;=\; (\!|a \leftarrow c|\!) \tag{2.18}$$

In the following diagrams, a fold and an unfold diagram are juxtaposed to form a hylo diagram.



From left to right, this represents fusing two functions together and deforesting an intermediate data structure. From right to left, this turns a control structure into a data structure. The *fold/unfold law* is a direct consequence of Theorem 2.3.3 and any of the fusion laws.

Now that the preliminaries have been established, recursive coalgebras will be used as a starting point to model the shortcut fusion techniques discussed. The ability to bring fold-based and unfold-based fusion into the same setting allows for some interesting comparisons. From there, the thesis will progress from a platonic domain to the practical concerns of implementation to, finally, the pragmatics of fusion during the compilation process.

Chapter 3

Modelling shortcut fusion transformations

In Chapter 2, three shortcut fusion techniques were discussed, as was a definition of shortcut fusion that encompasses these techniques: they share the common characteristic of standardising the way data structures are recursively consumed and produced, and algebraic transformations rewrite syntactically explicit instances of data structure consumption immediately followed by production. This approach has the benefit of allowing the programmer to focus on fusing a targeted set of functions without affecting other parts of the program. Where shortcut fusion techniques differ is in their choice of recursion scheme. The treatment of shortcut fusion techniques given so far has been specialised in imitation of their original presentations. The 'specialisation' is a result of presenting them in their original habitats, i.e. targeting lists with specialised combinators in Haskell. As has been shown, however, the underlying recursions schemes are more general. Folds and unfolds were also discussed in a categorical setting, along with some calculational properties of these and how they relate to recursion schemes. A similar treatment was given to recursive coalgebras, which allow folds and unfolds, incompatible in general, to be brought together in the same setting under certain conditions.

In this chapter, the calculational properties introduced in Section 2.3.1 will be used to formalise the shortcut fusion techniques discussed. In a more formal setting, the syntactic noise of specialisation can be filtered out, and the underlying foundations become more apparent. The idea behind using recursive coalgebras is that they bring together folds and unfolds, meaning the program transformations can be laid out in the same categorical setting. Recursive coalgebras therefore provide a general way to examine the relationships among these fusion approaches, which are not readily apparent when examining their individual implementations. In this chapter, as in

the previous one, it should be noted that reasoning applied to programs assumes that functions are written in total language, a not-uncommon assumption. Therefore, although the discussion is Haskell-based, it is, in fact, a subset of Haskell programs that our reasoning applies to, even though we do not explicitly depend on any particular ambient category with respect to recursive coalgebras.

The material presented in this chapter is based on the research published in Hinze et al. [2011]. Credit goes to Ralf Hinze for the choice of setting and the establishment of the preliminaries presented in Section 3.1, the connection to the acid rain theorem and its dual in Sections 3.3 and 3.4, and the observation that stream fusion's *Skip* turns a coalgebra into a natural transformation in Section 3.6. The presentation in the original paper was largely joint work between the author of this thesis and Daniel James, the latter of whom also contributed the parallel cata-hylo rule (3.19) in Section 3.4.

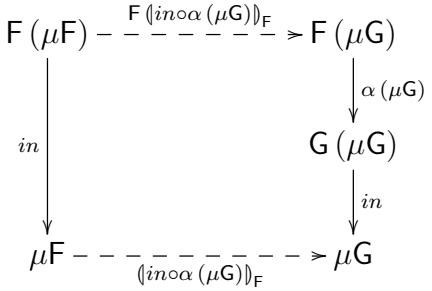
The author claims particular credit for the Haskell developments that run through this chapter, as well as the work on Church and Cochurch Encodings (Section 3.5), the data abstraction property for stream fusion (Section 3.6), and the discussion about recursiveness as it relates to functional programming languages (Section 3.7).

3.1 Preliminaries: Functor Fusion

One of the salient features of the recursion schemes discussed in Chapter 2 is that they come with fusion laws that allow them to absorb surrounding functions that satisfy certain properties. In the context of programs phrased as recursive functions, these represent program transformations that can result in more efficient programs, which is one of the reasons that modelling functions using these recursion schemes is useful. In order to continue, it is necessary first to establish one more fusion law. To begin, the least fixed point operator $\mu-$ must be turned into a functor of type $(\mathbb{C} \to \mathbb{C}) \to \mathbb{C}$. That is, it is a *higher-order functor* that takes an endofunctor to an object in the underlying category. The action on objects maps the endofunctor to its least fixed point i.e. its initial algebra, if it has one. Because the objects in question are functors, the arrows are natural transformations. Therefore, $\mu-$ turns a natural transformation $\alpha : \mathsf{F} \dot{\to} \mathsf{G} : \mathbb{C}^{\mathbb{C}}$ into an arrow $\mu\alpha : \mu\mathsf{F} \to \mu\mathsf{G} : \mathbb{C}$. It is defined as

$$\mu\alpha \quad = \quad (\!|in \circ \alpha\,(\mu\mathsf{G})|\!) \tag{3.1}$$

This definition of $\mu-$ as a higher-order functor can be shown with the following diagram:



To reduce clutter, the object argument of the natural transformation α will be omitted if it can be deduced from context. Based on the fact that $\mu\alpha$ is a fold, the *functor fusion law* can be derived:

$$(\!|b|\!)_{\mathsf{G}} \circ \mu\alpha \;=\; (\!|b \circ \alpha|\!)_{\mathsf{F}} \tag{3.2}$$

The functor fusion law states that a fold after a map can be fused into a single fold—the map $\mu\alpha$ can be seen as a 'base changer'. To prove this, it is necessary to note that, if (A, a) is a G-algebra and $\alpha :: \mathsf{F} \dot{\to} \mathsf{G}$, then $(A, a \circ \alpha\, A)$ is an F-algebra. Furthermore, homomorphisms in $\mathbf{Alg}(\mathsf{G})$ are also homomorphisms in $\mathbf{Alg}(\mathsf{F})$:

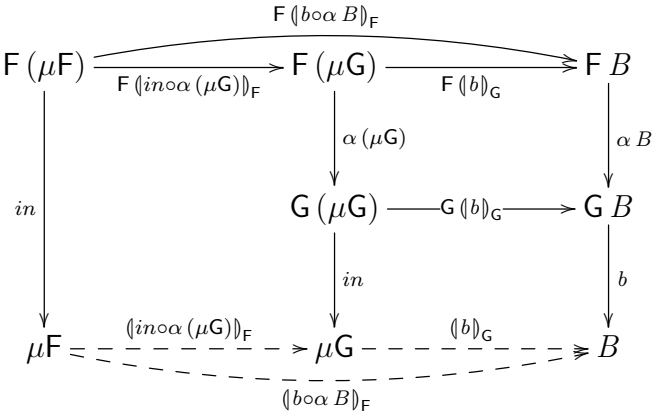
$$h : a \circ \alpha\, A \to b \circ \alpha\, B : \mathbf{Alg}(\mathsf{F}) \quad \Longleftarrow \quad h : a \to b : \mathbf{Alg}(\mathsf{G}) \tag{3.3}$$

$$\begin{aligned}
& h \circ a \circ \alpha\, A \\
= \quad & \{ \text{ assumption: } h : a \to b : \mathbf{Alg}(\mathsf{G}) \ \} \\
& b \circ \mathsf{G}\, h \circ \alpha\, A \\
= \quad & \{ \ \alpha \text{ is natural: } \mathsf{G}\, h \circ \alpha\, A = \alpha\, B \circ \mathsf{F}\, h \ \} \\
& b \circ \alpha\, B \circ \mathsf{F}\, h
\end{aligned}$$

Now, functor fusion (3.2) can be proved both diagrammatically and equationally.

Diagrammatic proof The diagram of the functor fusion law appears to be complex, but it is composed of three simple subdiagrams: the left hand pentagon is the diagram defined above for $\mu-$ as a functor; the top right square is the naturality of α; and the

bottom right square is the fold of the algebra b.



The diagram commutes. The arcing arrows come from the algebra $b \circ \alpha\, B$ on the right-hand side of the diagram. By uniqueness, we have the functor fusion law.

Equational proof To establish functor fusion equationally, the properties shown in each of the subdiagrams above are invoked:

$$
\begin{array}{ll}
 & (\!|b|\!)_{\mathsf{G}} \circ \mu\alpha = (\!|b \circ \alpha|\!)_{\mathsf{F}} \\
\iff & \quad \{ \text{ definition of } \mu- \text{ (3.1) } \} \\
 & (\!|b|\!)_{\mathsf{G}} \circ (\!|in \circ \alpha|\!)_{\mathsf{F}} = (\!|b \circ \alpha|\!)_{\mathsf{F}} \\
\Longleftarrow & \quad \{ \text{ fold fusion (2.4) } \} \\
 & (\!|b|\!)_{\mathsf{G}} : in \circ \alpha \to b \circ \alpha : \mathbf{Alg}(\mathsf{F}) \\
\Longleftarrow & \quad \{ \text{ naturality of } \alpha \text{ and (3.3) } \} \\
 & (\!|b|\!)_{\mathsf{G}} : in \to b : \mathbf{Alg}(\mathsf{G})
\end{array}
$$

With this law established, it is straightforward to show that $\mu-$ preserves identity

$$
\begin{array}{ll}
 & \mu id \\
= & \quad \{ \text{ definition of } \mu- \text{ (3.1) } \} \\
 & (\!|in \circ id|\!) \\
= & \quad \{ \text{ identity and reflection (2.3) } \} \\
 & id
\end{array}
$$

and composition

$$
\begin{aligned}
& \mu\beta \circ \mu\alpha \\
= & \quad \{ \text{ definition of } \mu- \text{ (3.1) } \} \\
& (\!|in \circ \beta|\!) \circ \mu\alpha \\
= & \quad \{ \text{ functor fusion (3.2) } \} \\
& (\!|in \circ \beta \circ \alpha|\!) \\
= & \quad \{ \text{ definition of } \mu- \text{ (3.1) } \} \\
& \mu(\beta \circ \alpha)
\end{aligned}
$$

It is also possible to continue with parallel developments in Haskell by giving a definition of $\mu-$ as a functor. The action on data is given by its datatype declaration. The action on functions is given by the following, which is simply a transcription of its definition (3.1) into Haskell:

$$
\begin{aligned}
& \mu- :: (\mathit{Functor}\, f) \Rightarrow (\forall a.f\ a \to g\ a) \to (\mu f \to \mu g) \\
& \mu\alpha = (\!|in \circ \alpha|\!)
\end{aligned}
$$

Note that the rank-2 polymorphic type is used to express the idea that $\mu-$ maps a natural transformation, modelled as a polymorphic function from $f\ a$ to $g\ a$ for all a, to a function between their fixpoints.

The definition of $\mu\alpha$ defines it as a fold. However, using Theorem 2.3.3 and the hylo shift law (2.17), it can actually be expressed as a fold, an unfold or a hylo.

$$\mu\alpha \;=\; (\!|in \circ \alpha|\!) \;=\; (\!|in \circ \alpha \leftarrow out|\!) \;=\; (\!|in \leftarrow \alpha \circ out|\!) \;=\; [\!(\alpha \circ out)]\!] \qquad (3.4)$$

(Note that the coalgebra of is a recursive coalgebra.) In Section 3.6, this equivalence will play a key role in modelling stream fusion.

3.2 Warm-up: Simple Fusion

Before digging in to the program transformations described in the background, a simpler example of fusion can be readily expressed to get into the right frame of mind. As a warm up, consider the program *sum* ∘ *between*, which creates a list of integers and then sums them together. The function *sum* can be expressed as a fold with algebra

$$
\begin{aligned}
&s :: \underline{List}\ Nat\ Nat \to Nat \\
&s\ \underline{Nil} &&= 0 \\
&s\ (\underline{Cons}\ x\ y) &&= x + y
\end{aligned}
$$

and *between* as unfold with the *recursive* (we omit a proof of this fact) coalgebra

$$
\begin{aligned}
&b :: (Nat, Nat) \to \underline{List}\ Nat\ (Nat, Nat) \\
&b\ (m, n) = \mathbf{if}\ m > n \\
&\qquad\qquad \mathbf{then}\ \underline{Nil} \\
&\qquad\qquad \mathbf{else}\ \underline{Cons}\ m\ (m + 1, n)
\end{aligned}
$$

The program $sum \circ between$ can therefore be expressed as $(\!|s|\!) \circ [\!(b)\!]$. Expressed thus, an intermediate list is used to build the sequence of numbers and then consumed to calculate the sum. It would instead be more efficient to write a program that sums the numbers as they are produced. The existence of such a transformation is given by the *fold/unfold law* (2.18) which can be applied to obtain $(\!|s|\!) \circ [\!(b)\!] = (\!|s \leftarrow b|\!)$. The definition that corresponds to this hylo can be derived as follows:

$$
\begin{aligned}
& (\!|s \leftarrow b|\!)\ (m, n) \\
= \quad & \{\ \text{definition of } (\!|- \leftarrow -|\!)\ \} \\
& s\ (fmap\ (\!|s \leftarrow b|\!)\ (b\ (m, n))) \\
= \quad & \{\ \text{definition of } b\ \} \\
& \mathbf{if}\ m > n\ \mathbf{then}\ s\ (fmap\ (\!|s \leftarrow b|\!)\ \underline{Nil}) \\
& \mathbf{else}\ s\ (fmap\ (\!|s \leftarrow b|\!)\ (\underline{Cons}\ m\ (m + 1, n))) \\
= \quad & \{\ \text{definition } fmap \text{ for } \underline{List}\ \} \\
& \mathbf{if}\ m > n\ \mathbf{then}\ s\ \underline{Nil} \\
& \mathbf{else}\ s\ (\underline{Cons}\ m\ ((\!|s \leftarrow b|\!)\ (m + 1, n))) \\
= \quad & \{\ \text{definition of } s\ \} \\
& \mathbf{if}\ m > n\ \mathbf{then}\ 0\ \mathbf{else}\ m + (\!|s \leftarrow b|\!)\ (m + 1, n).
\end{aligned}
$$

Therefore, the hylo corresponds to the function

$$
\begin{aligned}
&gauss :: (Nat, Nat) \to Nat \\
&gauss\ (m, n) = \mathbf{if}\ m > n \\
&\qquad\qquad \mathbf{then}\ 0 \\
&\qquad\qquad \mathbf{else}\ m + gauss\ (m + 1, n)
\end{aligned}
$$

This example is one of many cases where the laws of algebras and recursive coalgebras correspond to program transformations in functional programming.

3.3 Generalised *foldr/build* Fusion

It is now possible to move on to the main target of this setting: shortcut fusion. The original shortcut fusion technique, *foldr/build*, is fold-centric, and is furthermore specialised to folds over lists. In this section, the foundations of this technique will be exposed and generalised.

The mother of all fusion rules is algebra fusion (2.14). It allows a hylo followed by an algebra homomorphism to be combined into a single hylo. Similarly to fold fusion (2.4), however, the new hylo must be *constructed* such that it satisfies the precondition. To illustrate, the list pipeline $sum \circ filter\ odd$ can be expressed as a composition of two folds: $(\!|s|\!) \circ (\!|f|\!)$. The algebra s is the one of Section 3.2, and the algebra f is given by

$$\begin{aligned} f &:: \underline{List}\ Nat\ (\mu\underline{List}\ Nat) \to \mu\underline{List}\ Nat \\ f\ \underline{Nil} &= in\ \underline{Nil} \\ f\ (\underline{Cons}\ x\ y) &= \mathbf{if}\ odd\ x \\ &\quad\ \mathbf{then}\ in\ (\underline{Cons}\ x\ y) \\ &\quad\ \mathbf{else}\ y \end{aligned}$$

The intention is to fuse $(\!|s|\!) \circ (\!|f|\!)$ using algebra fusion. As a reminder, the algebra fusion law is

$$h \circ (\!|a \leftarrow c|\!) = (\!|b \leftarrow c|\!) \quad \Longleftarrow \quad h : a \to b$$

The left side of the law is instantiated with $h := (\!|s|\!)$ and $(\!|a \leftarrow c|\!) := (\!|f|\!)$, where (following Theorem 2.3.3) $a := f$ and $c := out$. To be able to apply algebra fusion, it is first necessary to prove that $(\!|s|\!)$ is an algebra homomorphism from f to some unknown algebra sf—the instantiation of $h : a \to b$, where $b := sf$. By hand, it is not hard to derive sf so that $(\!|s|\!) \circ f = sf \circ \mathsf{F}\,(\!|s|\!)$:

$$\begin{aligned} sf &:: \underline{List}\ Nat\ Nat \to Nat \\ sf\ \underline{Nil} &= 0 \\ sf\ (\underline{Cons}\ x\ y) &= \mathbf{if}\ odd\ x \\ &\quad\ \mathbf{then}\ x + y \\ &\quad\ \mathbf{else}\ y \end{aligned}$$

Since $(\!|s|\!)$ replaces in by s, we simply have to replace the occurrences of in in f by s (in the above example, the result of this replacement has been inlined to give the fully fused algebra). While this is an easy task to perform by hand, it is potentially difficult to mechanise, since it requires analysis of the body of f. This transformation

is therefore not purely syntactic, but also involves some further analysis of the source program. This is the same condition identified in Section 2.1.1, where attempting to fuse arbitrary arguments of *foldr* is not possible using algebraic transformations, and further analysis of these arguments is exactly what shortcut fusion strives to avoid.

The central idea of *foldr/build* fusion is to expose *in* so that replacing it by the algebra a is simple to implement via an algebraic transformation. Consider fold fusion (2.4) again:

$$h \circ (\!|a|\!) = (\!|b|\!) \quad \Longleftarrow \quad h : a \to b$$

The function $(\!|-|\!)$ is a transformation that turns an algebra into a homomorphism. Assume that there is another such transformation, β, that satisfies

$$h \circ \beta\, a = \beta\, b \quad \Longleftarrow \quad h : a \to b \tag{3.5}$$

Such a transformation can be created by abstracting away the constructors (i.e. *in*) found in an algebra that creates a new recursive data structure as its result.

The generalisation of *foldr/build* from lists to arbitrary datatypes, the so-called *acid rain rule* [Takano and Meijer, 1995], is then

$$(\!|a|\!) \circ \beta\, in \;=\; \beta\, a \tag{3.6}$$

Using β exposes *in* so that it can be replaced by a simply by replacing β's argument. Instead of building a structure and then folding over it, the *in* is eliminated and a is passed directly to β. The proof of correctness is painless:

$$\begin{array}{ll} & (\!|a|\!) \circ \beta\, in = \beta\, a \\ \Longleftarrow & \quad \{\ \text{assumption (3.5)}\ \} \\ & (\!|a|\!) : in \to a \end{array}$$

However, in order to apply (3.6), one must prove (3.5). Folds satisfy this property, but this instance of (3.6) is trivial: $(\!|a|\!) \circ (\!|in|\!) = (\!|a|\!)$, which can be satisfied by the fusion law 2.4. It turns out, though, that in a *relationally parametric* programming language [Reynolds, 1983], the proof obligation (3.5) amounts to the *free theorem* [Wadler, 1989] of the polymorphic type

$$\beta : \forall A.(\mathsf{F}\, A \to A) \to (B \to A) \tag{3.7}$$

for a fixed type B. An instance of the free theorem for this type is

$$\forall a.(\!|a|\!) \circ \beta\, in = \beta\, a \tag{3.8}$$

which, of course, is precisely the condition required. Therefore, in a relationally parametric language, the proof obligation can be discharged by the type checker. This is a generalisation of the proof presented in Section 2.1.1, which used the same technique but was specialised to lists.

Returning to the example, *in* can be abstracted out from *filter odd*, yielding $(\lambda a \to (\!|f'\ a|\!))\ in$ where

$$\begin{aligned} &f' :: (\underline{List}\ Nat\ b \to b) \to (\underline{List}\ Nat\ b \to b) \\ &f'\ a\ \underline{Nil} &&= a\ \underline{Nil} \\ &f'\ a\ (\underline{Cons}\ x\ y) &&= \mathbf{if}\ odd\ x \\ &&&\quad \mathbf{then}\ a\ (\underline{Cons}\ x\ y) \\ &&&\quad \mathbf{else}\ y \end{aligned}$$

We can then invoke the acid rain rule (3.6) to obtain

$$(\!|s|\!) \circ (\lambda a \to (\!|f'\ a|\!))\ in = (\lambda a \to (\!|f'\ a|\!))\ s = (\!|f'\ s|\!)$$

The example also shows that the *acid rain* rule is somewhat unstructured; a hylo is hidden inside the abstraction λa. Without performing an additional beta-reduction, the rule can be applied only once, as evidenced by *foldr/build* fusion's reliance on beta reduction to expose fusion opportunities. It is possible, however, to obtain a more structured rule if the abstraction is shifted to the algebra to achieve *cata-hylo fusion:* If τ is a functor $\mathbf{Alg}(\mathsf{F}) \to \mathbf{Alg}(\mathsf{G})$

$$h : \tau\ a \to \tau\ b : \mathbf{Alg}(\mathsf{G}) \quad \Longleftarrow \quad h : a \to b : \mathbf{Alg}(\mathsf{F}) \tag{3.9}$$

then

$$(\!|a|\!)_{\mathsf{F}} \circ (\!|\tau\ in \leftarrow c|\!)_{\mathsf{G}} \ = \ (\!|\tau\ a \leftarrow c|\!)_{\mathsf{G}} \tag{3.10}$$

If τ is $\lambda a \to a$, then this is just the fold/unfold law (2.18): $(\!|a|\!) \circ (\!|in \leftarrow c|\!) = (\!|a|\!) \circ [\![c]\!] = (\!|a \leftarrow c|\!)$. For $\tau\ a = a \circ \alpha$, this is essentially functor fusion (3.2).

Note that (3.3) is an example of (3.9). The proof of correctness is straightforward.

$$\begin{aligned} & (\!|a|\!)_{\mathsf{F}} \circ (\!|\tau\ in \leftarrow c|\!)_{\mathsf{G}} = (\!|\tau\ a \leftarrow c|\!)_{\mathsf{G}} \\ = \quad & \{\ \text{algebra fusion (2.14)}\ \} \\ & (\!|a|\!)_{\mathsf{F}} : \tau\ in \to \tau\ a : \mathbf{Alg}(\mathsf{G}) \\ = \quad & \{\ \text{assumption (3.9)}\ \} \\ & (\!|a|\!)_{\mathsf{F}} : in \to a : \mathbf{Alg}(\mathsf{F}) \end{aligned}$$

The proof obligation (3.9) once again amounts to a theorem for free, this time of the polymorphic type[1]

$$\tau :: \forall A.(\mathsf{F}\ A \to A) \to (\mathsf{G}\ A \to A)$$

Using cata-hylo fusion, the running example simplifies to

$$(\!|s|\!) \circ (\!|f'\ in|\!) = (\!|f'\ s|\!)$$

Now, a composition of folds can be fused all at once:

$$(\!|a|\!) \circ (\!|\tau_1\ in|\!) \circ \cdots \circ (\!|\tau_n\ in|\!) \circ [\![c]\!] = (\!|(\tau_n \circ \cdots \circ \tau_1)\ a \leftarrow c|\!)$$

It is now clearer how the rewrite rule results in a fused pipeline. Although the pragmatics of this method of fusion rely on beta reduction and inlining, this is down to implementation rather than the underlying foundations. On the left hand side, the incoming data structure is processed by a series of transformations that use *in* to build intermediate data structures, but *in* is exposed; on the right hand side, they are removed, and the τ's, representing the abstracted algebras, can be composed together to result in a fused pipeline, where the single-steps of each are performed all in one go, and the results combined by a.

3.4 Generalised *destroy*/*unfoldr* Fusion

The *foldr/build* brand of shortcut fusion, and its generalisation to algebraic datatypes, is *fold-centric*. This limits the kind of functions that can be fused, because some functions, such as *zip* or *take*, are not naturally written as folds. In many cases, forcing them into a fold is possible, but such contortion causes a loss of efficiency, defeating the purpose of fusion. It is possible to dualise *foldr/build* fusion to achieve an *unfold-centric* version. To illustrate, consider the simple pipeline $take\ 5 \circ between$, where $take\ n$ takes n elements (if available) from a list. It can be written as an unfold after an initialisation step: $take\ n = [\![t]\!] \circ start\ n$ where the coalgebra t and the initialisation function *start* are given by

$$\begin{aligned}
&\mathbf{type}\ State\ a = (Nat, a)\\
&start :: Nat \to a \to State\ a\\
&start\ n\ l = (n, l)\\
&t :: State\ (\mu\underline{List}\ a) \to \underline{List}\ a\ (State\ (\mu\underline{List}\ a))
\end{aligned}$$

[1] The original formulation of cata-hylo fusion, by Takano and Meijer [Takano and Meijer, 1995], unnecessarily requires F and G to be the same.

$$
\begin{aligned}
t\,(0,x) &= \underline{Nil} \\
t\,(n,x) &= \mathbf{case}\ out\ x\ \mathbf{of} \\
&\qquad \underline{Nil} \qquad\ \to \underline{Nil} \\
&\qquad \underline{Cons}\ a\ y \to \underline{Cons}\ a\,(n-1,y)
\end{aligned}
$$

Here, the notion that an unfold models a stateful computation is made explicit. The coalgebra takes a state as an argument and uses it to produce a value and a new state. In this example, the state type pairs the input list with a natural number that tracks the overall number of values produced. The number of elements to take, paired with the list where the values are to be taken from, forms the initial state.

The *acid rain rule* can be dualised to fuse the pipeline. Generally, if δ is a transformation that satisfies

$$\delta\ c = \delta\ d \circ h \quad \Longleftarrow \quad h : c \to d \tag{3.11}$$

then

$$\delta\ c \;=\; \delta\ out \circ [\![c]\!] \tag{3.12}$$

Previously, *in* was exposed, now *out* is. To apply the dual of acid rain *take n* is redefined as $(\lambda c \to [\![t'\ c]\!] \circ start\ n)\ out$, where

$$
\begin{aligned}
&t' :: (c \to \underline{List}\ a\ c) \to (State\ c \to \underline{List}\ a\ (State\ c)) \\
&t'\ c\,(0,x) = \underline{Nil} \\
&t'\ c\,(n,x) = \mathbf{case}\ c\ x\ \mathbf{of} \\
&\qquad \underline{Nil} \qquad\ \to \underline{Nil} \\
&\qquad \underline{Cons}\ a\ y \to \underline{Cons}\ a\,(n-1,y)
\end{aligned}
$$

The transformation t' is derived from t by abstracting away from *out*, instead using the argument c to obtain a new value. The example can now be tackled:

$$(\lambda c \to [\![t'\ c]\!] \circ start\ 5)\ out \circ [\![b]\!] = (\lambda c \to [\![t'\ c]\!] \circ start\ 5)\ b = [\![t'\ b]\!] \circ start\ 5$$

The proof obligation (3.11) corresponds to the free theorem of

$$\delta : \forall C.(C \to \mathsf{F}\ C) \to (C \to D) \tag{3.13}$$

where D is fixed. And, indeed, $\lambda c \to [\![t'\ c]\!] \circ start\ 5$ has the required type.

Similarly, *cata-hylo fusion* can be dualised to achieve the more structured *hylo-ana fusion:* If τ is a transformation that takes recursive F-coalgebras to recursive G-coalgebras satisfying

$$h : \tau\ c \to \tau\ d :: \mathbf{Rec}(\mathsf{G}) \quad \Longleftarrow \quad h : c \to d :: \mathbf{Rec}(\mathsf{F}) \tag{3.14}$$

then

$$(\!|a \leftarrow \tau\ c|\!)_{\mathsf{G}} \quad = \quad (\!|a \leftarrow \tau\ out|\!)_{\mathsf{G}} \circ [\![c]\!]_{\mathsf{F}} \tag{3.15}$$

The proof of correctness follows similarly to the proof of cata-hylo fusion.

$$\begin{aligned}
& (\!|a \leftarrow \tau\ c|\!)_{\mathsf{G}} = (\!|a \leftarrow \tau\ out|\!)_{\mathsf{G}} \circ [\![c]\!]_{\mathsf{F}} \\
= \quad & \{\ \text{coalgebra fusion (2.15)}\ \} \\
& [\![c]\!]_{\mathsf{F}} :: \tau\ c \to \tau\ out :: \mathbf{Rec}(\mathsf{G}) \\
= \quad & \{\ \text{assumption (3.14)}\ \} \\
& [\![c]\!]_{\mathsf{F}} :: c \to out :: \mathbf{Rec}(\mathsf{F})
\end{aligned}$$

This time the proof obligation (3.14) cannot be discharged by the type checker alone; τ must transform a *recursive* coalgebra into a *recursive* coalgebra. The intricacies of discharging such obligations are discussed in Section 3.7. This rule can also not immediately handle the state initialisation example. Phrased in terms of hylomorphisms, this program becomes

$$(\!|in \leftarrow \tau\ out|\!) \circ start\ 5 \circ [\![b]\!]$$

The initialisation function is stuck in between the hylo and the unfold.

This example focussed on fusing an input that consisted of single recursive type. As shown in this example, the state can actually be more complex. The aforementioned *zip* is an example of a function where state consists of two recursive data structures, and therefore has the potential to fuse both of these inputs. Consider the program $zip \circ (between \times between)$ as another example that can be written in terms of unfolds: $[\![z]\!] \circ ([\![b]\!] \times [\![b]\!])$. The coalgebra z is given by

$$\begin{aligned}
& z :: (\mu\underline{List}\ a, \mu\underline{List}\ b) \to \underline{List}\ (a, b)\ (\mu\underline{List}\ a, \mu\underline{List}\ b) \\
& z\ (as, bs) = \mathbf{case}\ (out\ as, out\ bs)\ \mathbf{of} \\
& \qquad\qquad (\underline{Cons}\ a\ as', \underline{Cons}\ b\ bs') \to \underline{Cons}\ (a, b)\ (as', bs') \\
& \qquad\qquad _ \qquad\qquad\qquad\qquad \to \underline{Nil}
\end{aligned}$$

Neither the dual of the acid rain rule (3.12) nor its more structured form (3.15) is applicable, as there are *two* producers preceding *zip*, one for each argument. To fuse such a function, it is necessary to employ *parallel hylo-ana fusion:* If τ satisfies,

$$\begin{aligned}
& h_1 \times h_2 : \tau\ (c_1, c_2) \to \tau\ (d_1, d_2) : \mathbf{Rec}(\mathsf{G}) \\
& \qquad\qquad \Longleftarrow \quad h_1 : c_1 \to d_1 : \mathbf{Rec}(\mathsf{F}_1)\ \wedge\ h_2 : c_2 \to d_2 : \mathbf{Rec}(\mathsf{F}_2)
\end{aligned} \tag{3.16}$$

then

$$(\!|\, a \leftarrow \tau\,(c_1, c_2) \,|\!)_{\mathsf{G}} \;=\; (\!|\, a \leftarrow \tau\,(out, out) \,|\!)_{\mathsf{G}} \circ ([\![c_1]\!]_{\mathsf{F}_1} \times [\![c_2]\!]_{\mathsf{F}_2}) \tag{3.17}$$

The proof of correctness the follows the same steps as for cata-hylo and hylo-ana.

$$\begin{aligned}
& (\!|\, a \leftarrow \tau\,(c_1, c_2) \,|\!)_{\mathsf{G}} = (\!|\, a \leftarrow \tau\,(out, out) \,|\!)_{\mathsf{G}} \circ ([\![c_1]\!]_{\mathsf{F}_1} \times [\![c_2]\!]_{\mathsf{F}_1}) \\
= \quad & \{ \text{ coalgebra fusion (2.15) } \} \\
& ([\![c_1]\!]_{\mathsf{F}_1} \times [\![c_2]\!]_{\mathsf{F}_1}) : \tau\,(c_1, c_2) \to \tau\,(out, out) : \mathbf{Rec}(\mathsf{G}) \\
= \quad & \{ \text{ assumption (3.16) } \} \\
& [\![c_1]\!]_{\mathsf{F}_1} : c_1 \to out : \mathbf{Rec}(\mathsf{F}_1) \;\wedge\; [\![c_2]\!]_{\mathsf{F}_2} : c_2 \to out : \mathbf{Rec}(\mathsf{F}_2)
\end{aligned}$$

Using the parallel hylo-ana rule, it is now possible to fuse the *zip* example:

$$[\![z'\,(out, out)]\!] \circ ([\![b]\!] \times [\![b]\!]) = [\![z'\,(b, b)]\!]$$

where the transformation z' is defined as

$$\begin{aligned}
& z' :: (s_1 \to \underline{List}\; a\; s_1, s_2 \to \underline{List}\; b\; s_2) \to (s_1, s_2) \to \underline{List}\,(a, b)\,(s_1, s_2) \\
& z'\,(c_1, c_2)\,(s_1, s_2) = \mathbf{case}\,(c_1\; s_1, c_2\; s_2)\;\mathbf{of} \\
& \qquad\qquad (\underline{Cons}\; a\; s_1, \underline{Cons}\; b\; s_2) \to \underline{Cons}\,(a, b)\,(s_1, s_2) \\
& \qquad\qquad _ \qquad\qquad\qquad\quad \to \underline{Nil}
\end{aligned}$$

Note that $\mathsf{F}_1 := \underline{List}\; A_1$, $\mathsf{F}_2 := \underline{List}\; A_2$ and $\mathsf{G} := \underline{List}\,(A_1 \times A_2)$.

Now is a good point to review the structured fusion rules presented so far. The cata-hylo rule absorbs a fold on the left of a hylo, and thus absorbs the consumer of a hylo into a new hylo. Dually, the hylo-ana rule absorbs an unfold on the right of a hylo, and thus absorbs a producer preceding a hylo into a new hylo. Functions such as *zip* consume two data structures, and the parallel hylo-ana rule was introduced to deal with this situation. In this case, two producers on the right of hylo are absorbed into a hylo. Given the apparent symmetry between the cata-hylo and hylo-ana rules, one might naturally wonder about a parallel cata-hylo rule. Such a rule would cover the case where a function produces a pair of data structures, which are consumed by two folds. The Haskell function $partition :: (a \to Bool) \to [a] \to ([a], [a])$ is a good example of this scenario. It takes a predicate and a list, and partitions the list based on the predicate. Let $partition\; even = (\!|pt|\!)$, where the algebra pt is given by,

$$\begin{aligned}
& pt :: \underline{List}\; Nat\; (\mu\underline{List}\; Nat, \mu\underline{List}\; Nat) \to (\mu\underline{List}\; Nat, \mu\underline{List}\; Nat) \\
& pt\; \underline{Nil} \qquad\qquad = (in\; \underline{Nil}, in\; \underline{Nil}) \\
& pt\,(\underline{Cons}\; n\,(x, y)) = \mathbf{if}\; even\; n
\end{aligned}$$

$$\begin{array}{l}\textbf{then}\ (in\ (\underline{Cons}\ n\ x), y)\\ \textbf{else}\ \ (x, in\ (\underline{Cons}\ n\ y))\end{array}$$

The example that will be used is $(\langle\!|s|\!\rangle \times \langle\!|s|\!\rangle) \circ \langle\!|pt|\!\rangle$; a list of natural numbers is partitioned into the odds and evens, and then these two lists are independently summed.

Now, parallel hylo-ana fusion can be dualised to present *parallel cata-hylo fusion:* If τ satisfies,

$$\begin{array}{l} h_1 \times h_2 : \tau\,(a_1, a_2) \to \tau\,(b_1, b_2) : \mathbf{Alg}(\mathsf{G}) \\ \qquad\qquad \Longleftarrow \quad h_1 : a_1 \to b_1 : \mathbf{Alg}(\mathsf{F}_1)\ \wedge\ h_2 : a_2 \to b_2 : \mathbf{Alg}(\mathsf{F}_2) \end{array} \tag{3.18}$$

then

$$(\langle\!|a_1|\!\rangle_{\mathsf{F}_1} \times \langle\!|a_2|\!\rangle_{\mathsf{F}_2}) \circ \langle\!|\tau\,(in, in) \leftarrow c|\!\rangle_{\mathsf{G}} \ = \ \langle\!|\tau\,(a_1, a_2) \leftarrow c|\!\rangle_{\mathsf{G}} \tag{3.19}$$

To apply parallel cata-hylo fusion to the *partition* example, it is first necessary to redefine pt as ψ so that it satisfies the form of τ.

$$\begin{array}{l} \psi :: (\underline{List}\ Nat\ x \to x, \underline{List}\ Nat\ y \to y) \to (\underline{List}\ Nat\ (x, y) \to (x, y)) \\ \psi\,(a_1, a_2)\ \underline{Nil} \qquad\qquad\quad = (a_1\ \underline{Nil}, a_2\ \underline{Nil}) \\ \psi\,(a_1, a_2)\ (\underline{Cons}\ n\ (x, y)) = \textbf{if}\ even\ n \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ (a_1\ (\underline{Cons}\ n\ x), y) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ (x, a_2\ (\underline{Cons}\ n\ y)) \end{array}$$

Now parallel cata-hylo fusion can be employed: $(\langle\!|s|\!\rangle \times \langle\!|s|\!\rangle) \circ \langle\!|\psi\,(in, in)|\!\rangle = \langle\!|\psi\,(s, s)|\!\rangle$

3.5 Church and Cochurch Encodings

In the two previous sections, *foldr/build* and *destroy/unfoldr* fusion were generalised and their foundations exposed. Central to this were β and δ, which allow the algebras of folds and coalgebras of unfolds to expose *in* and *out*, respectively, so that they can be removed via algebraic transformations. Furthermore, the proofs of correctness of these transformations are justified by their polymorphic types. These two functions deserve a second look that will provide a fresh perspective on recursive datatypes.

Consider again the polymorphic type of β (3.7) repeated below on the left hand side.

$$\forall A.(\mathsf{F}\,A \to A) \to (B \to A) \ \cong \ B \to (\forall A.(\mathsf{F}\,A \to A) \to A)$$

On the right, the type has been massaged to bring B to the front. The universally quantified type on the right is known as the *Church encoding* of $\mu\mathsf{F}$, so called because

the constructors have been abstracted away as function arguments in the manner used to embed data structures in the lambda calculus. Church encodings themselves are well-known, but their relationship to the abstractions used for fusion often goes unstated. This correspondence was, however, made explicit and utilised by Ghani et al. in generalising and modelling *foldr/build* fusion [Ghani and Uustalu, 2004], and by Johann and Ghani when they extended initial algebra semantics, and also *foldr/build* fusion, to nested datatypes [Johann and Ghani, 2007].

The type is quite remarkable, as it encodes a recursive type without using recursion, but is isomorphic to it. One direction of this isomorphism is given by the acid rain rule (3.6). The following derivation, which infers the isomorphisms, makes this explicit. The initial equation is (3.6) with the arguments of β swapped.

$$
\begin{aligned}
& \forall a.(\!|a|\!)\ (\beta\ b\ in) = \beta\ b\ a \\
\iff & \quad \{\ \text{change of variables } \beta\ b = t'\ \} \\
& \forall a.(\!|a|\!)\ (t'\ in) = t'\ a \\
\iff & \quad \{\ \text{extensionality}\ \} \\
& \lambda a \to (\!|a|\!)\ (t'\ in) = t' \\
\iff & \quad \{\ \text{define } toChurch\ x = \lambda a \to (\!|a|\!)\ x\ \} \\
& toChurch\ (t'\ in) = t' \\
\iff & \quad \{\ \text{define } fromChurch\ t' = t'\ in\ \} \\
& toChurch\ (fromChurch\ t') = t'
\end{aligned}
$$

The isomorphism *toChurch* creates a function whose argument is an algebra and folds that algebra over the given data structure. Its inverse, *fromChurch*, analogous to *build*, applies this function to the *in* algebra. Going the other direction yields the original structure: $fromChurch\ (toChurch\ s) = fromChurch\ (\lambda a \to (\!|a|\!)\ s) = (\lambda a \to (\!|a|\!)\ s)\ in = (\!|in|\!)\ s = s$. This is the other part of the isomorphism, which follows directly from fold reflection.

The Church encoding can be readily implemented in Haskell:

$$
\begin{aligned}
& \textbf{data}\ Church\ f = C\ \{\ app :: \forall a.(f\ a \to a) \to a\ \} \\
& toChurch :: (Functor\ f) \Rightarrow \mu f \to Church\ f \\
& toChurch\ x = C\ (\lambda a \to (\!|a|\!)\ x) \\
& fromChurch :: Church\ f \to \mu f \\
& fromChurch\ t' = app\ t'\ in
\end{aligned}
$$

As to be expected, everything dualises nicely. The polymorphic type of δ (3.13) gives rise to the *Cochurch encoding*:

$$\forall C.(C \to \mathsf{F}\ C) \to (C \to D) \ \cong \ (\exists\, C.(C \to \mathsf{F}\ C) \times C) \to D$$

The Cochurch encoding $\exists\, C.(C \to \mathsf{F}\ C) \times C$ can be thought of as the type of state machines encapsulating a transition function $C \to \mathsf{F}\ C$ and the current state C. Dually to the previous situation, this existentially quantified type is known in full as the Cochurch encoding of $\nu\mathsf{F}$. Again, it encodes a recursive type without using recursion, however, but now the one given by the greatest fixpoint of F. The Cochurch encoding can also be readily implemented in Haskell:

$$\begin{aligned}
&\mathbf{data}\ \mathit{CoChurch}\ f = \exists\, s.\mathit{CC}\ (s \to f\ s)\ s \\
&\mathit{toCoChurch} :: (\mathit{Functor}\ f) \Rightarrow \nu f \to \mathit{CoChurch}\ f \\
&\mathit{toCoChurch}\ x = \mathit{CC}\ \mathit{out}\ x \\
&\mathit{fromCoChurch} :: (\mathit{Functor}\ f) \Rightarrow \mathit{CoChurch}\ f \to \nu f \\
&\mathit{fromCoChurch}\ (\mathit{CC}\ \mathit{trans}\ \mathit{state}) = [\![\mathit{trans}]\!]\ \mathit{state}
\end{aligned}$$

The datatype declarations *Church* and *CoChurch* make explicit the underlying conversions that are central to the concept of shortcut fusion. These representation changes convert a recursive data structure to one with the recursion scheme 'built-in', and therefore transformations over them can be written as nonrecursively-defined (co-)algebras. Unlike recursive programs, compositions of these (co-)algebras can be inlined and optimised by the compiler. All that remains is for the programmer to instruct the compiler to remove any unnecessary conversions, i.e. cases of $\mathit{toChurch}.\mathit{fromChurch}$ and $\mathit{toCoChurch} \circ \mathit{fromCoChurch}$. Removing these transformations preserves the semantics of the program because we can prove the isomorphism between these representations. Such encoding also underlies the original formulation of stream fusion, which will be considered next.

3.6 Stream Fusion

The *foldr/build* flavour of fusion is fold-centric, in that it requires all functions that are intended to be fusible to be written as folds; similarly, *destroy/unfoldr* is unfold-centric. The boundaries of these world views are fuzzy, in that many unfolds can be written as folds (and vice versa). Such contortion of the recursion scheme, however, can have performance impacts, which defeat the purpose of the entire exercise. A

zip, for example, can be written as a fold, but only one of the two inputs can be fused [Gill et al., 1993, Section 9]. Along a similar vein, the *filter* for odd natural numbers, written before as a fold, can also be written as an unfold $[\![fs]\!]$ where

$$
\begin{aligned}
&fs :: \mu\underline{List}\ Nat \to \underline{List}\ Nat\ (\mu\underline{List}\ Nat)\\
&fs\ x = \mathbf{case}\ out\ x\ \mathbf{of}\\
&\qquad \underline{Nil} \qquad\ \to \underline{Nil}\\
&\qquad \underline{Cons}\ x\ y \to \mathbf{if}\ odd\ x\\
&\qquad\qquad\qquad\ \ \mathbf{then}\ \underline{Cons}\ x\ y\\
&\qquad\qquad\qquad\ \ \mathbf{else}\ fs\ y
\end{aligned}
$$

The coalgebra *fs* is a recursive coalgebra and therefore causes no issues in this setting, but it is also has a recursive definition, and this is a practical problem. A coalgebra must be nonrecursively defined for it to be inlined and fused. The use of shortcut fusion now seems particularly restrictive, because a program cannot have both *zip* and *filter* without intermediate data structures between them. The programmer is forced to choose between one of two worlds without the ability to move between them.

This restriction cannot be totally cast off, but it can be loosened. In the case of *filter*, there is nothing theoretically wrong with the definition above. The goal is then to remove the recursion. In doing so, things fall into place. The idea is to use a different base functor that has the ability to express the notion that an element is being discarded, but that computation continues to be productive. Recall the *Step* type from Section 2.1.3

$$
\begin{aligned}
&\mathbf{data}\ Step\ a\ s = Done\\
&\qquad\qquad\quad \mid\ Skip\ s\\
&\qquad\qquad\quad \mid\ Yield\ a\ s
\end{aligned}
$$

which can be seen as a base functor, with following *Functor* instance:

$$
\begin{aligned}
&\mathbf{instance}\ Functor\ (Step\ a)\ \mathbf{where}\\
&\quad fmap\ f\ Done \qquad\ = Done\\
&\quad fmap\ f\ (Skip\ s) \quad = Skip\ (f\ s)\\
&\quad fmap\ f\ (Yield\ a\ s) = Yield\ a\ (f\ s)
\end{aligned}
$$

Note that the type arguments have been swapped to allow for this definition, it is otherwise the same. With *Skip*, an element is discarded, but the value needed to carry on is kept. The consequence of this is that a coalgebra can yield a step in the

computation *without yielding a value*. Therefore the *fs* coalgebra can now be written as a composition of *out* with

$$
\begin{aligned}
&fs' :: Step\ Nat\ s \to Step\ Nat\ s \\
&fs'\ Done &&= Done \\
&fs'\ (Skip\ s) &&= Skip\ s \\
&fs'\ (Yield\ x\ s) &&= \mathbf{if}\ odd\ x\ \mathbf{then}\ Yield\ x\ s\ \mathbf{else}\ Skip\ s
\end{aligned}
$$

Now, $filter = [\![fs' \circ out]\!]$. Something interesting has happened: fs' is natural in the type of seed s. Since fs' is a natural transformation, it is also possible to define $filter = (\!|in \circ fs'|\!)$. The barrier has now been partially breached; *filter* is both a fold *and* an unfold of this new base functor! Moreover, it is an application of a mapping function: $filter = \mu fs'$. (Although we are really talking about a specialised *filter*, filtering for odd numbers, this generalises to function that takes a predicate as an argument instead.)

As another example, consider *map* for streams. Like fs', it simply passes on any *Skip*s it encounters, but otherwise behaves as it would over a list:

$$
\begin{aligned}
&ms :: (a \to b) \to (Step\ a\ s \to Step\ b\ s) \\
&ms\ f\ Done &&= Done \\
&ms\ f\ (Skip\ b) &&= Skip\ b \\
&ms\ f\ (Yield\ a\ b) &&= Yield\ (f\ a)\ b
\end{aligned}
$$

Given a function $f :: a \to b$, $ms\ f$ is natural transformation, meaning that it can be stated as both a fold $(\!|in \circ ms\ f|\!)$ and an unfold $[\![ms\ f \circ out]\!]$.

In general, consumers are folds, transformers are maps, and producers are unfolds. An entire pipeline of these can be fused into a single hylo:

$$(\!|a|\!) \circ \mu\alpha_1 \circ \cdots \circ \mu\alpha_n \circ [\![c]\!] = (\!|a \circ \alpha_1 \circ \cdots \circ \alpha_n \leftarrow c|\!)$$

Inspecting the types, the rule is clear:

$$A \xleftarrow{(\!|a|\!)} \mu\mathsf{F}_0 \xleftarrow{\mu\alpha_1} \mu\mathsf{F}_1 \qquad \cdots \qquad \mu\mathsf{F}_{n-1} \xleftarrow{\mu\alpha_n} \mu\mathsf{F}_n \xleftarrow{[\![c]\!]} C$$

In a sense, the introduction of *Skip* keeps the recursion in sync. Each transformation consumes a token and produces a token. Before, *filter* had to possibly consume several tokens before producing one.

The *Skip* trick is not unique to lists; for a base functor $\mathsf{F}\ A$ (where, as with $\underline{List}$, A is the element type), a *Skip*ping version $\mathsf{F}_s\ A$ can be defined as $\mathsf{F}_s\ A\ X = X + \mathsf{F}\ A\ X$, or in Haskell as:

$$\mathbf{data}\,\mathsf{F}_s\ a\ b = \mathit{Skip}\ b \mid \mathit{Base}\ (\mathsf{F}\ a\ b)$$

With *foldr/build* and *destroy/unfoldr* fusion, the correctness is justified because of the isomorphism between a (Co)Church encoding and its underlying datatype, which allows for unnecessary conversions to be removed while preserving the semantics of the program. This is not the case for stream fusion, however, as F and F_s are not isomorphic. The goal is nevertheless to be able to use *Skip* versions to model computations over the underlying datatype and achieve the same result. To accomplish this, the first order of business is to define conversion functions *toS* and *fromS* to convert to and from the skipping structure, respectively. They can be defined as an algebra and a coalgebra, so that a skipping structure is consumed using a fold, and is produced using an unfold with the underlying structure as the seed value.

$$\begin{aligned}
&\mathit{fromS} :: \mathsf{F}_s\ a\ (\mu\mathsf{F}\ a) \to (\mu\mathsf{F}\ a)\\
&\mathit{fromS}\ (\mathit{Skip}\ xs)\ \ = xs\\
&\mathit{fromS}\ (\mathit{Base}\ ys) = \mathit{in}_{\mathsf{F}}\ ys\\
&\mathit{toS} :: \mu\mathsf{F}\ a \to \mathsf{F}_s\ a\ (\mu\mathsf{F}\ a)\\
&\mathit{toS}\ xs = \mathit{Base}\ (\mathit{out}_{\mathsf{F}}\ xs)
\end{aligned}$$

Converting from the underlying data structure to a skipping version with $[\![\mathit{toS}]\!]$ does not introduce any *Skip*s, and so simply yields the data structure itself using the *out* of the underlying datatype, subscripted here for clarity. Likewise, $(\!|\mathit{fromS}|\!)$ simply reassembles these elements, assuming no *Skip*s appear. Therefore, $(\!|\mathit{fromS}|\!) \circ [\![\mathit{toS}]\!] = \mathit{id}$. Going the other way is not possible, however, since $(\!|\mathit{fromS}|\!)$ removes *Skip*s, and so $[\![\mathit{toS}]\!] \circ (\!|\mathit{fromS}|\!)$ can possibly yield a different F_s-structure than the original.

There is still hope, however. The idea is to depend not on the isomorphism between the two representations, but instead on the fact that skipping implementations, together with these conversion functions, fulfil the same specification as the analogous list functions over functions over $\mu\mathsf{F}\ a$ (*cf.* Lemma 1 and Theorem 3 in [Wang et al., 2010]). This is called the *data abstraction property*. In this setting, this obligation is expressed as a simple equality between a conventional function definition and its associated skipping version composed with the conversion functions. For example, for a function t that, like *filter*, both consumes and yields a data structure, it must be proved that

$$t = (\!|\mathit{fromS}|\!) \circ \mu\alpha \circ [\![\mathit{toS}]\!]$$

Because these functions can be phrased as folds, unfolds, and natural transformations, the proof is straightforward using the laws set out previously. Recall that $(\!|fromS|\!) \circ [\!(toS)\!] = id$. This fact can be used to simplify the obligation:

$$
\begin{aligned}
& t = (\!|fromS|\!) \circ \mu\alpha \circ [\!(toS)\!] \\
\iff & \quad \{\ (\!|fromS|\!) \circ [\!(toS)\!] = id\ \} \\
& t \circ (\!|fromS|\!) \circ [\!(toS)\!] = (\!|fromS|\!) \circ \mu\alpha \circ [\!(toS)\!] \\
\Longleftarrow & \quad \{\ \text{composition}\ \} \\
& t \circ (\!|fromS|\!) = (\!|fromS|\!) \circ \mu\alpha \\
\iff & \quad \{\ \text{functor fusion}\ \} \\
& t \circ (\!|fromS|\!) = (\!|fromS \circ \alpha|\!) \\
\Longleftarrow & \quad \{\ \text{fold fusion}\ \} \\
& t \circ fromS = fromS \circ \alpha \circ fmap\ (t)
\end{aligned}
$$

The obligation has been simplified to a requirement about the equivalence of a single step in each representation. The proof of this is easily accomplished by case analysis. This kind of function, which consumes a data structure to yield a new one of the same type, is a transformer. In this case, they are defined by converting to the skipping version, transforming it, and then converting back to the original datatype.

In addition to transformers, there are functions which either only consume a data structure or only produce one from a value. Their definitions do not fit the mold established by t, so their proof obligations are different. Consumers do not produce a new, fusible data structure, their definition only contains the conversion $[\!(toS)\!]$:

$$c = c_s \circ [\!(toS)\!]$$

The proof obligation can then once again be simplified:

$$
\begin{aligned}
& c = c_s \circ [\!(toS)\!] \\
\iff & \quad \{\ (\!|fromS|\!) \circ [\!(toS)\!] = id\ \} \\
& c \circ (\!|fromS|\!) \circ [\!(toS)\!] = c_s \circ [\!(toS)\!] \\
\Longleftarrow & \quad \{\ \text{composition}\ \} \\
& c \circ (\!|fromS|\!) = c_s \\
\Longleftarrow & \quad \{\ \text{fold fusion}\ \} \\
& c \circ fromS = c_s \circ fmap\ c
\end{aligned}
$$

This obligation can then be once again satisfied by case analysis on individual consumers. Dually, producers start from a non-fusible value to produce a stream, which can then be converted into a list, so they have the form

$$prod = (\!|fromS|\!) \circ p_s$$

For producers, the situation becomes more complex. It is not possible to rely on the fact that $(\!|fromS|\!) \circ [\![toS]\!] = id$ to simplify the obligation. This makes sense, because in the other cases, it was clear that the incoming data structure was being produced by $[\![toS]\!]$, which provides some information. In this case, p_s is opaque, so there no information to be gleaned from it. However, in the setting of recursive coalgebras, it is expected that programs are produced as unfolds. If this is the case, some progress can be made:

$$\begin{aligned}
& [\![p]\!] = (\!|fromS|\!) \circ [\![p_s]\!] \\
\iff & \quad \{ \text{ fold/unfold law } \} \\
& [\![p]\!] = (\!|fromS \leftarrow p_s|\!) \\
\iff & \quad \{ \text{ uniqueness property of hylomorphisms } \} \\
& [\![p]\!] = fromS \circ fmap\, [\![p]\!] \circ p_s
\end{aligned}$$

By relying on the fact that p is a recursive coalgebra, and therefore any hylo has a unique solution, it is possible to reduce the obligation once again, this time to a proof about the recursive coalgebra producing the data structure.

Although stream fusion is the first to make use of this a *Skip*-like augmentation in this manner, it is closely related to Capretta's representation of general recursion in type theory [Capretta, 2005], which proposes adding a 'computation step' constructor to coinductive types to maintain productivity. It is also similar to the resumption monad, a manner of modelling concurrent computations, which uses a constructor to represent a pause in a computation and stores the 'rest' of the computation for when it resumes [Harrison, 2006]. As demonstrated, *Skip* can easily be used with other datatypes by adding a *Skip* to their base functor. However, the usefulness of this augmentation for other datatypes for the purpose of fusion is unclear and has not yet been explored.

3.7 Recursiveness

Throughout this chapter, proofs of recursiveness of coalgebras have been elided. The reason for this is that the developments have not depended upon which ambient

category is chosen, as long as it has the characteristics mentioned. This is not the case, however, when it comes to proving recursiveness, and doing so would have broken this generality by selecting a specific category in which to prove recursiveness. This section will discuss recursiveness, tactics for proving it in some relevant settings, and the implications of recursiveness.

By definition, a recursive coalgebra of an endofunctor F is one for which, for all F-algebras, there is exists unique solution to the hylo diagram. The implications of this fact are not necessarily the same for all ambient categories, and therefore there is no general strategy for proving recursiveness. Previous developments of recursive coalgebras have focussed on proving recursiveness in *Set*. In this setting, it has been found that a coalgebra is recursive if and only if it defines a well founded relation [Taylor, 1996], meaning that recursiveness can be proven indirectly. It was also proven that, for finitary endofunctors in *Set*, a coalgebra is recursive if it has the halting property, i.e. if it describes a system that, given a fixed state, terminates in finitely many steps [Adámek et al., 2007]. Therefore, in *Set*, termination implies recursiveness, and proving termination can be used to prove recursiveness.

Using *Set* as the ambient category ensures that the results of this chapter hold if we assume that functions are total and that the recursive data structures are finite. The semantics of lazy functional programming languages (in this case, Haskell), however, are often modelled using other categories, notably $\mathbf{Cpo}_{\perp}$. One method of dealing with hylos in this setting is to rely on algebraic compactness, i.e. depending on the fact that least and greatest fixed points coincide. In such a situation, the hylo diagram has a canonical, but not a unique solution, and this is the approach taken by Meijer et al. [1991]. Doing so, however, sacrifices the reasoning power granted by uniqueness and results in a less general setting.

Capretta et al. [2006] informally discuss the issue of recursiveness in lazy functional programming languages. In terms of proof methods, they give examples of known recursive coalgebras, and then methods of constructing new algebras out of them. Some of these constructions are given in Section 2.3. In terms of the implication of recursiveness, Capretta argues that the uniqueness 'follows from the arguments of the recursive call always being strictly shorter than that of the main call'. This is a property of the coalgebra, as it is the coalgebra that prepares the arguments for the recursive call. The functions used in this chapter certainly satisfy this informal notion of breaking down the problem for the recursive call. This helps provide an intuition about recursiveness, but fails to provide any rigorous explanation, and remains an open problem.

The power of recursive coalgebras is the reasoning they provide *without* depending on a specific category. The tradeoff for this generality in reasoning is that recursiveness is still not well understood for all underlying categories. Nevertheless, as will be shown in subsequent chapters, this setting is a useful laboratory that has provided valuable insight into the underlying mechanics of shortcut fusion. Such results *can* be transferred to the Haskell world by suggesting starting points more direct reasoning, as will be shown. More direct applicability of category-theoretic work in modelling lazy functional semantics may also be strengthened by future work in this area.

3.8 Conclusions

In this chapter, the goal was to use a formal setting to model the fusion techniques discussed in Chapter 2. Hylomorphisms were chosen because allowed folds and unfolds to be brought together. Using recursive coalgebras provided a setting that was less dependent on the choice of the ambient category, as well as providing the extra reasoning power of uniqueness. In one aspect, the results are interesting because they provide another application of recursive coalgebras, which are relatively new to the scene. It is also a useful setting for gaining a more general view of shortcut fusion, as the individual fusion techniques could be laid out side-by-side and examined. This allowed for an examination of the dual nature of the fusion techniques and exposed their underlying foundations. Of particular note is the formalisation of stream fusion and the resultant understanding of how and why the *Skip* constructor, whose roots were rather more pragmatic than theoretical, expands the expressibility of a recursion scheme in a way that enables fusion of a greater variety of functions. The importance of Church and Cochurch encodings, and, more generally, the role of representation change, was also exposed. It is this concept in particular that will be used in subsequent chapters, where the notion shortcut fusion will be generalised to provide a framework that describes it without reference to a specific recursion scheme.

Chapter 4

A generalised framework for shortcut fusion

In the previous chapter, three shortcut fusion techniques were formally modelled in a category-theoretic setting. Among the results gleaned from this model was the fact that the techniques modelled rely on *representation change* to create more efficient programs: recursive datatypes are converted to a representation in which functions over them are nonrecursive, unnecessary conversions between representations are removed by simple syntactic rewrites, and the resulting pipelines of functions can be readily fused by the compiler.

This unifying feature suggests that these techniques are actually instantiations of a general program transformation. In this chapter, a framework is established that describes shortcut fusion as such a general program transformation. The result is that the transformation, and associated proof obligations, can be described without reference to a specific underlying representation. The framework can then be instantiated to a specific representation and collection of functions over it. The underlying representations of the techniques covered fit into such a framework and are instantiated for example data structures.

As in the previous chapter, certain assumptions are made in reasoning about programs. In particular, the use of properties of folds and unfolds applies specifically to total programs. However, discussions also delve further into the effect of non-totality on these programs and how this can affects the use of this reasoning. Furthermore, the use of hylomorphisms is introduced later, which assumes a setting in which they are usable. In summary, a subset of Haskell in which programs are total is assumed.

This chapter is based on material published in Harper [2011], for which the author of this thesis was the sole author.

4.1 Shortcut fusion as data abstraction

Most generally, the term *fusion* is used here to refer to a transformation that takes the composition of a series of functions

$$f = f_n \circ \cdots \circ f_1$$

defined over one or more recursive datatypes and converts it into a single recursive pass f' such that $f = f'$, where the difference is that f' removes the intermediate data structures that are passed from f_i to f_{i+1}. As stated previously, the challenges posed by optimising recursive functions in general means that removing these intermediate data structures is beyond the scope of most compilers' standard arsenal of optimisations.

The shortcut fusion approach is to convert values of recursive datatypes to a different representation where fusion becomes possible by the compiler's already-implemented optimisations. Assume that the goal is to convert values of the recursive datatype A to values of another type C. The idea is that C can faithfully represent values of A, but composed functions over C can be fused automatically. Instead of writing functions directly over A, they are defined in terms of functions over C along with conversion functions that convert between A and C. These conversion functions will be called $con : A \to C$ and $abs : C \to A$.

This setup can be seen as an instance of *data abstraction* [Hoare, 1972]. The type A is the *abstract* datatype over which an interface is defined and the behaviour of that interface specified, and the type C is the *concrete* datatype over which the interface is implemented. In order for C to be a faithful representation of A, *con* and *abs* must have the property

$$abs \circ con = id_A \tag{4.1}$$

The import of this property is that the type C must be capable of uniquely representing all values of A, and that *con* maps distinct values of A to distinct values of C, from which the original A values can be recovered. Note that it is not required that the opposite i.e. $con \circ abs = id_C$ be true.

4.1.1 Defining fusible interfaces

Once a suitable representation is established, the goal is to write functions over the abstract A in terms of a function over C. A function using C can be viewed as a refinement of a function that is implemented directly over A that achieves the same

result. Therefore, the aim is to ensure that this refinement accurately implements the original function. For example, consider a function $f : A \to A$, which transforms a recursive data structure into another one (e.g. a map or a filter). To define it over C using *abs* and *con*, a function $f_C : C \to C$ is required such that the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{con} & C \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f_C} \\ A & \xleftarrow[abs]{} & C \end{array}$$

If such a function is implemented, then f can be redefined as $f' = abs \circ f_C \circ con$. Using the property of *abs* and *con* (4.1), it is possible to identify the condition that f_C must fulfil to be used in this manner:

$$\begin{aligned} & f = abs \circ f_C \circ con \\ \iff & \quad \{ \;\; abs \circ con = id_{\mu\mathsf{F}} \;\; \} \\ & f \circ abs \circ con = abs \circ f_C \circ con \\ \impliedby & \quad \{ \;\; \text{Composition} \;\; \} \\ & f \circ abs = abs \circ f_C \end{aligned}$$

This means that f_C is considered to implement f when converting a data structure from the concrete representation and applying f, the function over the abstract type, has the same result as first using f_C to accomplish the transformation and then performing the conversion. As will be shown, this property is necessary to justify program transformations when fusible functions are composed.

In isolation, such a function definition does not necessarily yield any performance improvement. In fact, the cost of converting between A and C may even make this function *less* efficient than one that is defined directly over A. Suppose, however, that there are functions $g : A \to A$ and $g_C : C \to C$ defined likewise, and f and g are composed:

$$\begin{array}{ccc} A & \xrightarrow{con} & C \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f_C} \\ A & \xleftarrow[abs]{} & C \\ & \searrow^{con} & \\ {\scriptstyle g}\downarrow & & C \\ & & \downarrow{\scriptstyle g_C} \\ A & \xleftarrow[abs]{} & C \end{array}$$

Down the left hand side of this diagram, the program can be seen as being written directly over A, in which f passes results to g using an intermediate data structure. The goal is to be able to travel down the right side, however, despite gaps in this route, in order to yield the program

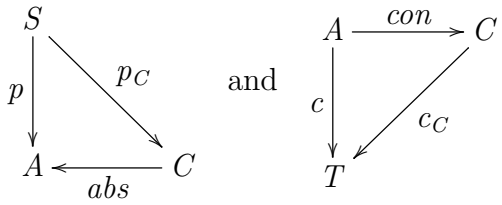
$$abs \circ g_C \circ f_C \circ con$$

which, because it contains a composition of functions over C, is fusible. Such a program is only correct, however, if the conversion $con \circ abs$ in the middle can be removed. The simplest situation is the one in which these gaps can be filled with id, which means that it can be removed unconditionally. The argument that this is the case is, in fact, what underlies the claim that *foldr/build* and *destroy/unfoldr* are correct, i.e. that the implicit representation change is fully reversible. This is an extremely strong condition, demanding not only that C faithfully represent values of A, but also vice versa, i.e. that $A \cong C$. Such a condition would disqualify other useful types, such as stream fusion's *Stream*s, and is therefore not a suitable condition in general.

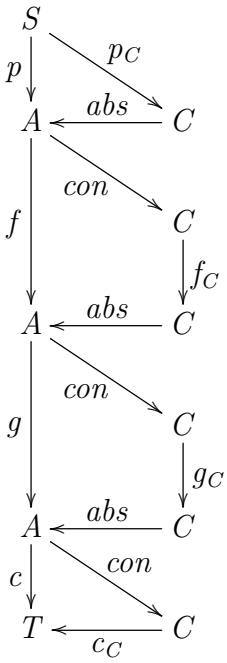
This is not the only way to justify the program transformation, however. Instead, it turns out that it suffices that, if f_C and g_C are refinements of f and g, i.e. if they satisfy one of the properties stated above, then the fusible program preserves the meaning of the original.

$$\begin{aligned}
& abs \circ f_C \circ g_C \circ con \\
= \quad & \{ \text{ assumption: } f \circ abs = abs \circ f_C \ \} \\
& f \circ abs \circ g_C \circ con \\
= \quad & \{ \text{ assumption: } g \circ abs = abs \circ g_C \ \} \\
& f \circ g \circ abs \circ con \\
= \quad & \{ \ abs \circ con = id_A \ \} \\
& f \circ g
\end{aligned}$$

In Chapter 3, stream fusion functions were classified based on whether they consumed, produced, or transformed a data structure, and this determined the correctness property they had to satisfy. This same classification applies here; the functions f and g are examples of *transformers*. In addition to such functions, there are *consumers*, which only consume a data structure in order to produce a value, and *producers*, which create a data structure out of a seed value. In this framework, a producer $p : S \to A$ and a consumer $c : A \to T$ will have fusible counterparts $p_C : S \to C$ and $c_C : C \to T$, respectively, such that



commute. This yields the obligations $p = abs \circ p_C$ and $c = c_C \circ con$. Like the property for transformations, these conditions are sufficient to allow us to remove the unnecessary conversions. To tie it all together, consider a pipeline $c \circ f \circ g \circ p$, whose functions have types as above and meet the necessary obligations for their fusible counterparts. Diagrammatically, this pipeline can be rendered



By removing the $con \circ abs$ conversions, a pipeline is obtained in which there are no longer any nonfusible functions over the recursive datatype. Instead, the program $c_C \circ f_C \circ g_C \circ p_C$, when optimised by the compiler, will take a seed of type S and use it to produce elements that it transforms and consumes without producing any intermediate data structures in order to produce a single value, like previous fused examples.

By taking a step back from the specific representations used in shortcut fusion, a more general setup has emerged. This has centred around idea of representation change and how to use this to obtain fusible programs. In the next section, representations that can be used for fusion will be discussed and instantiated for this framework.

4.2 Fusible Representations

As mentioned before, the issue at hand is that the compiler cannot fuse pipelines composed of recursive functions. It will, however, inline nonrecursive functions and remove intermediate data structures from the resulting program. For shortcut fusion, a datatype is needed that faithfully represents a recursive datatype but, paradoxically, allows us to write functions with nonrecursive definitions over it. Previously, it was revealed that the fusion techniques discussed in Chapter 3 were based on the Church encodings and Cochurch encodings of recursive datatypes, which allow us to do exactly what is needed. Examples of such encodings will be instantiated within the framework just presented using Haskell.

4.2.1 Instantiating Church encodings

In Section 3.5, it was shown that Church and Cochurch encodings are implicitly used in *foldr/build* and *destroy/unfoldr* fusion, respectively. In this section, Church encodings encodings will be instantiated for a specific datatype: leaf trees. A representation will be declared for use with this datatype, and then a selection of interface functions will be defined, along with discussions about how these satisfy the proof obligations established in previous sections. As with the proof obligations, the development will be broken down into three types of functions: consumers, producers, and transformers. This development will then be paralleled for Cochurch encodings in the following section.

Recall that, for a base functor F, the Church encoding is a function of type

$$\mathbf{data}\ \mathit{Church}\ \mathsf{F} = C\ (\forall A.(\mathsf{F}\ A \to A) \to A)$$

which takes an algebra and folds it over an encoded data structure. The rank-2 polymorphic type guarantees that the result actually depends on the algebra passed in i.e. that it obtains its results of type A using the algebra that is passed in. Church encodings represent a recursive datatype such that the recursion scheme is 'built-in'. Functions that transform Church encodings do not describe the recursion scheme itself, only a single step in the form of an algebra. If this algebra has a nonrecursive definition, a pipeline of such functions can be inlined and fused by the usual complement of compiler optimisations. To demonstrate the details of implementing functions in this way, leaf trees will be used as a running example. Leaf trees can be defined as

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Leaf}\ a \mid \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Such a tree can be used as an alternative to lists as a representation of linear sequences. The base functor of this datatype is

$$\begin{aligned}
&\textbf{data}\ \underline{\mathit{Tree}}\ a\ b = \underline{\mathit{Empty}} \mid \underline{\mathit{Leaf}}\ a \mid \underline{\mathit{Branch}}\ b\ b \\
&\textbf{instance}\ \mathit{Functor}\ (\underline{\mathit{Tree}}\ a)\ \textbf{where} \\
&\quad \mathit{fmap}\ f\ (\underline{\mathit{Empty}}) \quad\ = \underline{\mathit{Empty}} \\
&\quad \mathit{fmap}\ f\ (\underline{\mathit{Leaf}}\ x) \quad\ = \underline{\mathit{Leaf}}\ x \\
&\quad \mathit{fmap}\ f\ (\underline{\mathit{Branch}}\ l\ r) = (\underline{\mathit{Branch}}\ (f\ l)\ (f\ r))
\end{aligned}$$

Therefore, the corresponding Church encoding for this datatype is

$$\textbf{data}\ \mathit{Tree}^{\dagger}\ a = \mathit{Tree}^{\dagger}\ (\forall b.(\underline{\mathit{Tree}}\ a\ b \to b) \to b)$$

Unlike in previous chapters, datatypes will not be represented as fixed points of their base functors here. The more common style for implementing this representation change is to convert between the explicitly recursive datatype its Church encoding directly. To use a Church encoding as a representation, the conversion functions toC and fromC are instantiated for con and abs. A recursive data structure $\mu\mathsf{F}$ is converted to its Church encoding by defining a function that takes an algebra and folds that algebra over the data structure:

$$\begin{aligned}
&\mathit{toC} :: \mu f \to \mathit{Church}\ f \\
&\mathit{toC}\ x = C\ (\lambda a \to (\!|\, a \,|\!)\ x)
\end{aligned}$$

Here, as in *foldr/build* fusion, the algebra being applied has been abstracted away. Unlike in *foldr/build*, this conversion is made explicit.

To get the data structure back, the Church encoding is applied to the initial algebra $\mathit{in} :: f\ (\mu f) \to f$

$$\begin{aligned}
&\mathit{fromC} :: \mathit{Church}\ f \to \mu f \\
&\mathit{fromC}\ (C\ g) = g\ \mathit{in}
\end{aligned}$$

which defines how to construct the datatype. This is analogous to *build*. These definitions can be specialised for the Church encoding of *Tree*:

$$\begin{aligned}
&\mathit{toTree}^{\dagger} :: \mathit{Tree}\ a \to \mathit{Tree}^{\dagger}\ a \\
&\mathit{toTree}^{\dagger}\ x = \mathit{Tree}^{\dagger}\ (\lambda a \to \mathit{fold}\ a\ x)
\end{aligned}$$

$$
\begin{aligned}
&\mathit{fromTree}^{\dagger} :: \mathit{Tree}^{\dagger}\ a \to \mathit{Tree}\ a \\
&\mathit{fromTree}^{\dagger}\ (\mathit{Tree}^{\dagger}\ g) = g\ \mathit{in}
\end{aligned}
$$

Definitions must now be provided for *fold* and *in*. First, *fold* takes an algebra $\underline{\mathit{Tree}}\ a\ b \to b$ and folds it over the *Tree*:

$$
\begin{aligned}
&\mathit{fold} :: (\underline{\mathit{Tree}}\ a\ b \to b) \to \mathit{Tree}\ a \to b \\
&\mathit{fold}\ a\ \mathit{Empty} &&= a\ \underline{\mathit{Empty}} \\
&\mathit{fold}\ a\ (\mathit{Leaf}\ x) &&= a\ (\underline{\mathit{Leaf}}\ x) \\
&\mathit{fold}\ a\ (\mathit{Branch}\ l\ r) &&= a\ (\underline{\mathit{Branch}}\ (\mathit{fold}\ a\ l)\ (\mathit{fold}\ a\ r))
\end{aligned}
$$

In order to apply the algebra to each of the constructors, they have to be swapped from those of the recursive datatype to those of the base functor. This constructor swap does incur some overhead performance-wise, but it is meant to be fused away. The reverse swapping of constructors appears in *in*, when a *Tree* is constructed:

$$
\begin{aligned}
&\mathit{in} :: \underline{\mathit{Tree}}\ a\ (\mathit{Tree}\ a) \to \mathit{Tree}\ a \\
&\mathit{in}\ \underline{\mathit{Empty}} &&= \mathit{Empty} \\
&\mathit{in}\ (\underline{\mathit{Leaf}}\ x) &&= \mathit{Leaf}\ x \\
&\mathit{in}\ (\underline{\mathit{Branch}}\ l\ r) &&= \mathit{Branch}\ l\ r
\end{aligned}
$$

To prove that Church encodings faithfully represent their underlying datatype, it must be proved that $\mathit{fromC} \circ \mathit{toC} = \mathit{id}$. This is simply a consequence of the reflection law for folds:

$$
\begin{aligned}
& \mathit{fromC}\ (\mathit{toC}\ x) \\
= & \quad \{\ \text{definition of } \mathit{toC}\ \} \\
& \mathit{fromC}\ (C\ (\lambda a \to (\!|a|\!)\ x)) \\
= & \quad \{\ \text{definition of } \mathit{fromC}\ \} \\
& (\lambda a \to (\!|a|\!)\ x)\ \mathit{in} \\
= & \quad \{\ \text{function application}\ \} \\
& (\!|\mathit{in}|\!)\ x \\
= & \quad \{\ \text{fold reflection law (2.3)}\ \} \\
& x
\end{aligned}
$$

At this point, prior proofs of correctness for *foldr/build* have rested on proving the isomorphism between Church encodings and their underlying datatype, i.e. that

$toC \circ fromC = id$:

$$
\begin{aligned}
& toC\,(fromC\,(C\,g)) = C\,g \\
\iff & \quad \{\ \text{definition of } fromC\ \} \\
& toC\,(g\,in) = (C\,g) \\
\iff & \quad \{\ \text{definition of } toC\ \} \\
& C\,(\lambda a \to (\!|a|\!)\,(g\,in)) = (C\,g) \\
\iff & \quad \{\ \text{extensionality}\ \} \\
& \forall a.(\!|a|\!)\,(g\,in) = g\,a
\end{aligned}
$$

It is here that such proofs rest on the free theorem of g in previous chapters. To prove this isomorphism, it must be shown that applying a Church-encoded datatype to an algebra is equivalent to constructing the data structure using *in* and then folding the same algebra over it. We avoid the need for this proof by shifting the obligation to the conversion and interface functions that manipulate the abstract and concrete datatypes.

The traditional obligation of isomorphism and the means of proving it, free theorems, can be problematic, particularly if we break away from the assumption that all Haskell programs are total. First, free theorems depend on the fact that information about the behaviour of a function can be gleaned from its type, in particular those that have universally-quantified type parameters, but there are certain caveats to their use in Haskell. In particular, free theorems can fail in the presence of non-total functions, as well as Haskell's *seq* function. This function allows for the evaluation of arbitrary expressions that do not have to appear in the result nor in the type of the function. This circumvents the guarantees made by free theorems by hiding information from the typechecker. This issue has been specifically examined in proofs of correctness for program transformations Johann and Voigtländer [2004], in which 'correct' transformations have been shown to change a non-terminating program into a terminating one and vice versa. The use of free theorems has been recovered in Haskell by imposing preconditions on their use, but this results in a loss of the generality and simplicity sought by the proof technique by placing strictness conditions on various parts of the program, meaning that the isomorphism proof sought no longer completely holds.

The second reason is that free theorem-based proofs might not be suitable for all program transformations. Stream fusion represents a case in which there is no isomorphism between the concrete and abstract datatypes (because of *Skip*), and as

a result the transformation is not correct in general, only for those cases described above. Therefore, although free theorems may provide a quick and easy route to proving correctness, the condition they prove is stronger than required for shortcut fusion and cannot be used in all situations. As a consequence, this thesis explores the alternative route of placing proof obligations on the fusible function definitions (as indicated above) rather than on concrete datatype being used, as this allows for a more general description of proof obligations.

Some generality can nevertheless be maintained. As outlined in Section 2.1, the transformation must be justified by proving that fusible functions constitute implementations of the functions that they are replacing. By using Church encodings, it is given that the recursion scheme that is built-in to the datatype is a fold. This information can be used to simplify the proof conditions. As the proof obligations are addressed in this section, we will show how the use of properties of folds (and, in the next section, unfolds) can be used to simplify these obligations when Church and Cochurch representations, respectively, are used.

For consumers, the obligation is rather simple; a consumer applies the Church-encoded value to an algebra, so $c_C = \lambda(C\ g) \to g\ b$ for some algebra $b : \mathsf{F}\ B \to B$. The goal is to prove that this is equal to folding b over the underlying data structure, i.e. that $(\lambda(C\ g) \to g\ b)\ (toC\ x) = (\!|b|\!)\ x$:

$$
\begin{aligned}
& (\lambda(C\ g) \to g\ b)\ (toC\ x) \\
= \quad & \{\ \text{definition of } toC\ \} \\
& (\lambda(C\ g) \to g\ b)\ (C\ (\lambda a \to (\!|a|\!)\ x)) \\
= \quad & \{\ \text{function application}\ \} \\
& (\lambda a \to (\!|a|\!)\ x)\ b \\
= \quad & \{\ \text{function application}\ \} \\
& (\!|b|\!)\ x
\end{aligned}
$$

An example of such a consumer is *sum*:

$$
\begin{aligned}
& sum :: Tree\ Int \to Int \\
& sum\ Empty \qquad\quad = 0 \\
& sum\ (Leaf\ x) \qquad = x \\
& sum\ (Branch\ x\ y) = sum\ x + sum\ y
\end{aligned}
$$

Such a function can be defined as a fold (this can be justified using the universal property of folds) with the algebra

$$
\begin{aligned}
&s :: \underline{Tree}\ Int\ Int \to Int \\
&s\ \underline{Empty} &&= 0 \\
&s\ (\underline{Leaf}\ x) &&= x \\
&s\ (\underline{Branch}\ x\ y) &&= x + y
\end{aligned}
$$

This allows for a definition of *sum* over $Tree^{\dagger}$:

$$
\begin{aligned}
&sum^{\dagger} :: Tree^{\dagger}\ Int \to Int \\
&sum^{\dagger}\ (Tree^{\dagger}\ g) = g\ s
\end{aligned}
$$

To create a function over *Tree*s, this function merely needs to be composed with $toTree^{\dagger}$:

$$
\begin{aligned}
&sum' :: Tree\ Int \to Int \\
&sum' = sum^{\dagger} \circ toTree^{\dagger}
\end{aligned}
$$

satisfying the proof of correctness is simply a matter of inlining the definitions of $sum^{\dagger}$ and $toTree^{\dagger}$, which show that this is equal to $fold\ s$.

For producers, the situation is less straightforward. Unlike consumers, these do not obviously conform to a specific structured recursion scheme. Instead, they construct a data structure recursively. Although the goal is generally to avoid recursion, it is acceptable here. This is because, lacking a fold to build in the recursion scheme, the recursion must be performed by the producer itself. This recursive definition, however, abstracts away the list constructors. Therefore, a producer has the form $(\lambda x \to C\ (\lambda a \to p_C\ a\ x))$ where p_C recursively creates elements, and folds them by putting a where the constructors belong. If the encoded data structure is actually constructed, using $fromC$, it turns out in is passed to p_C:

$$
\begin{aligned}
& fromC\ ((\lambda x \to C\ (\lambda a \to p_C\ a\ x))\ s) \\
= &\quad \{\ \text{function application}\ \} \\
& fromC\ (C\ (\lambda a \to p_C\ a\ s)) \\
= &\quad \{\ \text{definition of } fromC\ \} \\
& (\lambda a \to p_C\ a\ s)\ in \\
= &\quad \{\ \text{function application}\ \} \\
& p_C\ in\ s
\end{aligned}
$$

The proof of correctness proceeds by comparing the recursive function that creates the abstract datatype directly to the one that creates a Church encoding and verifying

that, when passed *in*, the latter is equal to the former. For example, take a function that, given a pair of integers, generates the enumeration from the first to the second, inclusive (this is analogous to the Prelude function *enumFromTo* over lists):

$$
\begin{aligned}
&\mathit{enum} :: (\mathit{Int}, \mathit{Int}) \to \mathit{Tree}\ \mathit{Int} \\
&\mathit{enum}\ (x, y) \\
&\quad \mid x > y \ = \mathit{Empty} \\
&\quad \mid x \equiv y = \mathit{Leaf}\ x \\
&\quad \mid x < y \ = \mathit{Branch}\ (\mathit{enum}\ (x, \mathit{mid}))\ (\mathit{enum}\ (\mathit{mid} + 1, y)) \\
&\quad \textbf{where} \\
&\qquad \mathit{mid} = (x + y)\ `\mathit{div}`\ 2
\end{aligned}
$$

This version divides the enumeration into two sub-enumerations for the recursive call, yielding a leaf when an enumeration has length 1. To create a Church-encoded version, the function is defined similarly, but abstracts the constructors away as an extra argument and uses it instead of the constructors:

$$
\begin{aligned}
&\mathit{enum}^{\dagger} :: (\mathit{Int}, \mathit{Int}) \to \mathit{Tree}^{\dagger}\ \mathit{Int} \\
&\mathit{enum}^{\dagger}\ (x, y) = \mathit{Tree}^{\dagger}\ (\lambda a \to \mathit{go}\ a\ (x, y)) \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ a\ (x, y) \\
&\qquad\quad \mid x > y \ = a\ \underline{\mathit{Empty}} \\
&\qquad\quad \mid x \equiv y = a\ (\underline{\mathit{Leaf}}\ x) \\
&\qquad\quad \mid x < y \ = a\ (\underline{\mathit{Branch}}\ (\mathit{go}\ a\ (x, \mathit{mid})) \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{go}\ a\ (\mathit{mid} + 1, y))) \\
&\qquad\quad \textbf{where} \\
&\qquad\qquad \mathit{mid} = (x + y)\ `\mathit{div}`\ 2 \\
&\mathit{enum}' :: (\mathit{Int}, \mathit{Int}) \to \mathit{Tree}\ \mathit{Int} \\
&\mathit{enum}' = \mathit{fromTree}^{\dagger} \circ \mathit{enum}^{\dagger}
\end{aligned}
$$

Here, the recursion is handled by *go*, which has the same body as *enum* but applies the abstracted algebra instead of the constructors. Although the obligation above provides a general roadmap for proving the correctness of producers, it does not provide any clear characterisation about what sort of functions satisfy it. Some insight can be gleaned from the fact that, in addition to being defined recursively, *enum*, in this case, does conform to a structured recursion scheme. Specifically, it is an unfold with the coalgebra

$$
\begin{aligned}
&b :: (Int, Int) \to \underline{Tree}\ Int\ (Int, Int)\\
&b\ (x, y)\\
&\quad \mid x > y = \underline{Empty}\\
&\quad \mid x \equiv y = \underline{Leaf}\ x\\
&\quad \mid x < y = \underline{Branch}\ (x, mid)\ (mid + 1, y)\\
&\quad \textbf{where}\\
&\qquad mid = (x + y)\ `div`\ 2
\end{aligned}
$$

Therefore, it could be written $enum^{\dagger}\ (x, y) = Tree^{\dagger}\ (\lambda a \to (fold\ a \circ unfold\ b)\ (x, y))$. This version creates an intermediate data structure, but it could be fused, according to the fold/unfold rule (2.18) given in Chapter 3, resulting in a hylomorphism. The function *go* is this hylomorphism, although the algebra portion has been abstracted away into an argument. Such reasoning requires working in a setting where hylomorphisms are valid, however. This can be justified by proving that b is a recursive coalgebra, and admitting hylomorphisms in that manner. Alternatively, it suffices to note that hylomorphisms can be used in Haskell because it is algebraically compact, and the carrier of the initial algebra and final coalgebra coincide. This admits unfolds as producers in Church-encoding-based fusion, but it does not necessarily imply that *only* unfolds can be used. It simply means there is a general proof that applies to all unfolds in a setting where hylomorphisms are a valid recursion scheme.

Transformations are both producers and consumers. They consume a Church encoding by applying it to an algebra, but the result is a new Church-encoded data structure. Over the abstract data type, such a function is a fold where the algebra uses *in* to construct the new structure. In the Church-encoded version, *in* is abstracted away in the algebra. It therefore has the form $(\lambda(C\ g) \to C\ (\lambda a \to g\ (t_C\ a)))$, where t_C takes an algebra and produces a new one, which is then passed to the Church-encoded datatype. To prove the equality, then, it must be proved that $t_C\ in$ is equal to the algebra used by the transformer over the abstract type:

$$
\begin{aligned}
&\quad (\!|t|\!)\ (fromC\ xs) = fromC\ ((\lambda(C\ g) \to C\ (\lambda a \to g\ (t_C\ a)))\ (C\ xs))\\
\iff &\quad \{\ \text{function application}\ \}\\
&\quad (\!|t|\!)\ (fromC\ xs) = fromC\ (C\ (\lambda a \to xs\ (t_C\ a)))\\
\iff &\quad \{\ \text{definition of } fromC\ \}\\
&\quad (\!|t|\!)\ (fromC\ xs) = (\lambda a \to xs\ (t_C\ a))\ in\\
\iff &\quad \{\ \text{function application}\ \}\\
&\quad (\!|t|\!)\ (fromC\ xs) = xs\ (t_C\ in)
\end{aligned}
$$

There is a roadblock here, because the definition of xs is obscured. Usually, the free theorem provides the necessary next step. The 'promise' provided by the free theorem is that xs folds the algebra that is passed to it over an underlying data structure. If it is assumed that $xs = (\lambda a \to (\!|a|\!)\ x)$ for some underlying data structure x, then the proof proceeds as:

$$
\begin{aligned}
& (\!|t|\!)\ (\mathit{fromC}\ (\lambda a \to (\!|a|\!)\ x)) = (\lambda a \to (\!|a|\!)\ x)\ (t_C\ \mathit{in}) \\
\iff & \quad \{\ \text{function application}\ \} \\
& (\!|t|\!)\ (\mathit{fromC}\ (\lambda a \to (\!|a|\!)\ x)) = (\!|t_C\ \mathit{in}|\!)\ x \\
\iff & \quad \{\ \text{definition of } \mathit{fromC} \text{ and application}\ \} \\
& (\!|t|\!)\ (\!|\mathit{in}|\!)\ x = (\!|t_C\ \mathit{in}|\!)\ x \\
\iff & \quad \{\ \text{reflection}\ \} \\
& (\!|t|\!)\ x = (\!|t_C\ \mathit{in}|\!)\ x
\end{aligned}
$$

Such an assumption is justifiable, because both toC and correct producers satisfy this assumption with the Church encodings they create. Furthermore, transformers that satisfy this obligation preserve the correctness condition for any transformer or producer that follows after, because they preserve the underlying recursion scheme and merely abstract away the relevant occurrences of in, creating placeholders for the next algebra. It may seem somewhat unsatisfactory that the types involved are not leveraged in the proof. Specifically, xs is no longer considered as providing any guarantee. The fact of the matter is that this is only possible because of the isomorphism between Church encodings and their underlying datatype. In the case of other representations, where the concrete type is more expressive than abstract, there may be some values in C for which these functions are not equivalent. In this case, the proof shows that if the underlying recursion scheme is indeed a fold, then this transformation holds, and it is assumed this is so because only other functions that satisfy their proof obligations are used in these pipelines.

As an example of a transformer, consider the *filter* function over *Tree*s:

$$
\begin{aligned}
& \mathit{filter} :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Tree}\ a \\
& \mathit{filter}\ p\ \mathit{Empty} &&= \mathit{Empty} \\
& \mathit{filter}\ p\ (\mathit{Leaf}\ a) &&= \textbf{if}\ p\ a\ \textbf{then}\ \mathit{Leaf}\ a\ \textbf{else}\ \mathit{Empty} \\
& \mathit{filter}\ p\ (\mathit{Branch}\ l\ r) &&= \mathit{Branch}\ (\mathit{filter}\ p\ l)\ (\mathit{filter}\ p\ r)
\end{aligned}
$$

In this definition, it is considered acceptable to replace a *Leaf* with an empty tree, meaning that the resulting tree may contain *Empty*s. In some definitions of leaf trees,

this breaks an invariant, which is that non-empty leaf trees do not contain any *Empty* nodes. A solution to this will be proposed later on in Section 4.3. The algebra for this function is

$$
\begin{aligned}
&f :: (\underline{\mathit{Tree}}\ a\ b \to b) \to (a \to \mathit{Bool}) \to \underline{\mathit{Tree}}\ a\ b \to b \\
&f\ a\ p\ \underline{\mathit{Empty}} &&= a\ \underline{\mathit{Empty}} \\
&f\ a\ p\ (\underline{\mathit{Leaf}}\ x) &&= \mathbf{if}\ p\ x\ \mathbf{then}\ a\ (\underline{\mathit{Leaf}}\ x)\ \mathbf{else}\ a\ \underline{\mathit{Empty}} \\
&f\ a\ p\ (\underline{\mathit{Branch}}\ l\ r) &&= a\ (\underline{\mathit{Branch}}\ l\ r)
\end{aligned}
$$

More accurately, this is a function that, given an algebra (and a predicate), creates an algebra. When used to transform $\mathit{Tree}^\dagger$s, it is used thus:

$$
\begin{aligned}
&\mathit{filter}^\dagger :: (a \to \mathit{Bool}) \to \mathit{Tree}^\dagger\ a \to \mathit{Tree}^\dagger\ a \\
&\mathit{filter}^\dagger\ p\ (\mathit{Tree}^\dagger\ g) = \mathit{Tree}^\dagger\ (\lambda a \to g\ (f\ a\ p)) \\
&\mathit{filter}' :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Tree}\ a \\
&\mathit{filter}'\ p = \mathit{fromTree}^\dagger \circ \mathit{filter}^\dagger\ p \circ \mathit{toTree}^\dagger
\end{aligned}
$$

The argument to the new Church tree is the algebra that is passed to f, which is then placed instead of the constructors. If passed in, it will produce the underlying structure in the same way as the original transformation that it is replacing, and this is guaranteed by the proof.

4.2.2 Cochurch Encodings

The concept of Church encodings dualises to Cochurch encodings, which can also be instantiated. This will parallel the development in the previous section, discussing the proof obligations of producers, consumers, and transformers when Cochurch encodings are used. As with folds, properties of unfolds will be used to simplify and satisfy these obligations.

For a base functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$, the type of the Cochurch encoding is

$$\mathbf{data}\ \mathit{CoChurch}\ f = \forall s.\mathit{CC}\ (s \to f\ s)\ s$$

Dual to the universal quantification seen in the Church encoding, Cochurch encodings use existential type quantification to enforce the requirement that the type of the seed and the type of the stepper function match up. Unlike Church encodings, the recursion is not built into the datatype. Instead, the necessary machinery to perform a single step is bundled but not yet executed. Continuing with the running example, the Cochurch encoding of *Tree* can be represented as

$$\textbf{data}\ \mathit{Tree}^{\ddagger}\ a = \forall s.\, \mathit{Tree}^{\ddagger}\ (s \to \underline{\mathit{Tree}}\ a\ s)\ s$$

To convert a data structure to its Cochurch encoding, a pair is created with *out*, which breaks apart a single level of a recursive data structure, as the stepper function and the data structure itself as the seed:

$$\begin{aligned}
&\mathit{toCC} :: \nu f \to \mathit{CoChurch}\ f\\
&\mathit{toCC}\ x = \mathit{CC}\ \mathit{out}\ x
\end{aligned}$$

To get back to the data structure, it is unfolded using the stepper function, recursively applying to successive seeds:

$$\begin{aligned}
&\mathit{fromCC} :: \mathit{CoChurch}\ f \to \nu f\\
&\mathit{fromCC}\ (\mathit{CC}\ h\ x) = [\![h]\!]\ x
\end{aligned}$$

Continuing with the same base functor, *out* and $\mathit{toTree}^{\ddagger}$ for *Tree*s are defined as

$$\begin{aligned}
&\mathit{out} :: \mathit{Tree}\ a \to \underline{\mathit{Tree}}\ a\ (\mathit{Tree}\ a)\\
&\mathit{out}\ \mathit{Empty} \quad\ = \underline{\mathit{Empty}}\\
&\mathit{out}\ (\mathit{Leaf}\ a) \quad = \underline{\mathit{Leaf}}\ a\\
&\mathit{out}\ (\mathit{Branch}\ l\ r) = \underline{\mathit{Branch}}\ l\ r\\
&\mathit{toTree}^{\ddagger} :: \mathit{Tree}\ a \to \mathit{Tree}^{\ddagger}\ a\\
&\mathit{toTree}^{\ddagger}\ t = \mathit{Tree}^{\ddagger}\ \mathit{out}\ t
\end{aligned}$$

To get back to the original structure, *fromCC* and *unfold* are instantiated for *Tree*

$$\begin{aligned}
&\mathit{fromTree}^{\ddagger} :: \mathit{Tree}^{\ddagger}\ a \to \mathit{Tree}\ a\\
&\mathit{fromTree}^{\ddagger}\ (\mathit{Tree}^{\ddagger}\ h\ s) = \mathit{unfold}\ h\ s\\
&\mathit{unfold} :: (s \to \underline{\mathit{Tree}}\ a\ s) \to s \to \mathit{Tree}\ a\\
&\mathit{unfold}\ h\ s = \textbf{case}\ h\ s\ \textbf{of}\\
&\quad \underline{\mathit{Empty}} \quad\ \to \mathit{Empty}\\
&\quad \underline{\mathit{Leaf}}\ a \quad\ \to \mathit{Leaf}\ a\\
&\quad \underline{\mathit{Branch}}\ l\ r \to \mathit{Branch}\ (\mathit{unfold}\ h\ l)\ (\mathit{unfold}\ h\ r)
\end{aligned}$$

Proving the required $\mathit{fromCC} \circ \mathit{toCC}$ proceeds along similar lines to Church encodings, with the conversion to Cochurch encodings and back harnessing the reflection law of

unfolds:

$$
\begin{aligned}
& \mathit{fromCC}\ (\mathit{toCC}\ x) \\
= \quad & \{\ \text{definition of } \mathit{toCC}\ \} \\
& \mathit{fromCC}\ (\mathit{CC}\ \mathit{out}\ x) \\
= \quad & \{\ \text{definition of } \mathit{fromCC}\ \} \\
& [\!(\mathit{out})\!]\ x \\
= \quad & \{\ \text{unfold reflection law (2.8)}\ \} \\
& x
\end{aligned}
$$

As with Church encodings, it is useful to rely on the fact that Cochurch functions will replace functions of a certain form when satisfying the proof obligations. As unfolds are dual to folds, the easiest case this time is the production of a data structure. If producer is phrased as an unfold $[\!(c)\!]$ applied to a seed s, the equivalent Cochurch version simply takes the seed and pairs it with the coalgebra:

$$
\begin{aligned}
& \mathit{fromCC}\ ((\lambda s \rightarrow \mathit{CC}\ c\ s)\ x) \\
= \quad & \{\ \text{function application}\ \} \\
& \mathit{fromCC}\ (\mathit{CC}\ c\ x) \\
= \quad & \{\ \text{definition of } \mathit{fromCC}\ \} \\
& [\!(c)\!]\ x
\end{aligned}
$$

Taking *enum* again as an example, the coalgebra b can be used to define $\mathit{enum}^{\ddagger}$ by pairing it with the initial pair of *Int*s.

$$
\begin{aligned}
& \mathit{enum}^{\ddagger} :: (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{Tree}^{\ddagger}\ \mathit{Int} \\
& \mathit{enum}^{\ddagger}\ (x, y) = \mathit{Tree}^{\ddagger}\ b\ (x, y) \\
& \mathit{enum}'' :: (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{Tree}\ \mathit{Int} \\
& \mathit{enum}'' = \mathit{fromTree}^{\ddagger} \circ \mathit{enum}^{\ddagger}
\end{aligned}
$$

Because both the unfold over the abstract datatype and the Cochurch encoding function use the same coalgebra, they satisfy the proof obligation.

Folds are not naturally producers, and Church producers have to have recursive definitions that build in the recursion as well as placement of the abstracted algebra. Similarly, unfolds are not naturally consumers. This means that a Cochurch consumer will have the form $(\lambda \mathit{CC}\ h\ s \rightarrow c\ h\ s)$, where c is a recursive function that applies the stepper function h to seeds and consumes rather than building a fusible data

structure. To satisfy the proof obligation for consumers, it must be proved that, given *out* as the stepper function, it should consume a data structure of abstract datatype in the same manner as the consumer implemented directly over it:

$$
\begin{aligned}
& (\lambda CC\ c\ s \to f\ c\ s)\ (toCC\ x) \\
= \quad & \{\ \text{definition of } toCC\ \} \\
& (\lambda CC\ c\ s \to f\ c\ s)\ (CC\ out\ x) \\
= \quad & \{\ \text{function application}\ \} \\
& f\ out\ x
\end{aligned}
$$

The example function *sum* can be defined this way:

$$
\begin{aligned}
& sum^{\ddagger} :: Tree^{\ddagger}\ Int \to Int \\
& sum^{\ddagger}\ (Tree^{\ddagger}\ h\ s) = go\ s \\
& \quad \textbf{where} \\
& \quad\quad go\ s = \textbf{case}\ h\ s\ \textbf{of} \\
& \quad\quad\quad \underline{Empty} \quad\quad \to 0 \\
& \quad\quad\quad \underline{Leaf}\ x \quad\quad \to x \\
& \quad\quad\quad \underline{Branch}\ l\ r \to go\ l + go\ r
\end{aligned}
$$

If *out* is passed in for h, and a *Tree* for s, it is an easy proof that this is equal to *sum*; they both return 0 for the empty tree, the value itself of any leaves, and add the results of the recursive calls for the branches. As *sum* has been shown to be a fold, however, it can also be observed that a consumer such as *sum* can be written $sum\ (Tree^{\ddagger}\ h\ s) = fold\ s \circ unfold\ h\ s$. The fold/unfold law again applies, and this becomes a hylomorphism, and *go* has the required form, where it applies the coalgebra to the seed, makes the recursive calls, and then adds the results. Therefore, folds can be considered as valid Cochurch consumers, although again this does not automatically exclude other sorts of functions.

Finally, transformers behave similarly to those in Church encodings in having characteristics of both consumers and producers. As a consumer, they take a coalgebra and a seed that they use to produce values. Like a producer, however, their result is not a value but a new coalgebra and seed pair. This is accomplished by defining a function that takes a coalgebra, applies it a seed, and then performs a transformation does something on the resulting value. This can again be demonstrated by satisfying

the proof obligation for transformers:

$$
\begin{aligned}
& [\![t]\!]\,(\mathit{fromCC}\,(\mathit{CC}\;h\;x)) = \mathit{fromCC}\,((\lambda \mathit{CC}\;h\;s \to \mathit{CC}\,(t_C\;h)\;s)\,(\mathit{CC}\;h\;x)) \\
\iff & \quad \{\ \text{function application}\ \} \\
& [\![t]\!]\,(\mathit{fromCC}\,(\mathit{CC}\;h\;x)) = \mathit{fromCC}\,(\mathit{CC}\,(t_C\;h)\;x) \\
\iff & \quad \{\ \text{definition of } \mathit{fromCC}\ \} \\
& [\![t]\!]\,(\mathit{fromCC}\,(\mathit{CC}\;h\;x)) = [\![t_C\;h]\!]\;x \\
\iff & \quad \{\ \text{definition of } \mathit{fromCC}\ \} \\
& [\![t]\!]\,([\![h]\!]\;x) = [\![t_C\;h]\!]\;x \\
\iff & \quad \{\ \text{extensionality}\ \} \\
& [\![t]\!] \circ [\![h]\!] = [\![t_C\;h]\!] \\
\impliedby & \quad \{\ \text{fusion law}\ \} \\
& \mathit{fmap}\,[\![h]\!] \circ (t_C\;h) = t \circ [\![h]\!]
\end{aligned}
$$

Here, the fusion rule has been invoked in order to prove that the single, fused unfold is equal to the two successive unfolds. This can be proved by case analysis. As an example, *filter* can be defined as an unfold with the algebra:

$$
\begin{aligned}
& f :: (a \to \mathit{Bool}) \to \mathit{Tree}\;a \to \underline{\mathit{Tree}}\;a\,(\mathit{Tree}\;a) \\
& f\;p\;\mathit{Empty} \qquad\quad = \underline{\mathit{Empty}} \\
& f\;p\,(\mathit{Leaf}\;x) \qquad = \mathbf{if}\;p\;x\;\mathbf{then}\,(\underline{\mathit{Leaf}}\;x)\;\mathbf{else}\;\underline{\mathit{Empty}} \\
& f\;p\,(\mathit{Branch}\;l\;r) = \underline{\mathit{Branch}}\;l\;r
\end{aligned}
$$

This function both converts the tree constructors to their base functor and filters the results with the given predicate. For the Cochurch encoding version, the coalgebra is passed the previous coalgebra, which it then applies to the seed to get an element, and then filters it:

$$
\begin{aligned}
& f_C :: (a \to \mathit{Bool}) \to (s \to \underline{\mathit{Tree}}\;a\;s) \to s \to \underline{\mathit{Tree}}\;a\;s \\
& f_C\;p\;h\;s = \mathbf{case}\;h\;s\;\mathbf{of} \\
& \quad \underline{\mathit{Empty}} \qquad \to \underline{\mathit{Empty}} \\
& \quad \underline{\mathit{Leaf}}\;x \qquad \to \mathbf{if}\;p\;x\;\mathbf{then}\;\underline{\mathit{Leaf}}\;x\;\mathbf{else}\;\underline{\mathit{Empty}} \\
& \quad \underline{\mathit{Branch}}\;l\;r \to \underline{\mathit{Branch}}\;l\;r \\
& \mathit{filter}^{\ddagger} :: (a \to \mathit{Bool}) \to \mathit{Tree}^{\ddagger}\;a \to \mathit{Tree}^{\ddagger}\;a \\
& \mathit{filter}^{\ddagger}\;p\,(\mathit{Tree}^{\ddagger}\;h\;s) = \mathit{Tree}^{\ddagger}\,(f_C\;p\;h)\;s \\
& \mathit{filter}'' :: (a \to \mathit{Bool}) \to \mathit{Tree}\;a \to \mathit{Tree}\;a \\
& \mathit{filter}''\;p = \mathit{fromTree}^{\ddagger} \circ \mathit{filter}^{\ddagger}\;p \circ \mathit{toTree}^{\ddagger}
\end{aligned}
$$

Since f and f_C both give the same results for a given case, the result for $\underline{Empty}$ and $\underline{Leaf}\ x$ is the same. Furthermore, they both return the same result their original argument $\underline{Branch}$. On the left hand side of the equation, these branches will contain the unfolded subtrees. On the right hand side, these will contain the seeds to unfold those trees, which the mapping the unfold over the result will also accomplish.

4.3 Streams

In the implementations of *filter*, it was considered acceptable to replace a *Leaf* value with the *Empty* tree. As stated, this violates a common invariant when working with leaf trees, which is that they are either *Empty* or only contain *Branch*es and *Leaf*s. One way to get around this in the Church encoding is to modify *in*, so that any *Empty* trees are discarded:

$$
\begin{aligned}
&inn' :: \underline{Tree}\ a\ (Tree\ a) \to Tree\ a \\
&inn'\ \underline{Empty} &&= Empty \\
&inn'\ (\underline{Leaf}\ x) &&= Leaf\ x \\
&inn'\ (\underline{Branch}\ Empty\ r) &&= r \\
&inn'\ (\underline{Branch}\ l\ Empty) &&= l \\
&inn'\ (\underline{Branch}\ l\ r) &&= Branch\ l\ r
\end{aligned}
$$

When inn' is used to construct *Tree*s, only a truly empty tree will contain *Empty* constructors, and all the rest will be discarded. This breaks the isomorphism between Church encodings and their underlying datatypes, however; the concrete representation is now the Church encoding of leaf trees with possible *Empty* constructors, and the abstract one is leaf trees without them. How does this affect the development? For producers, this is no problem, because they do not need to introduce any empties, as there is no need for this extra expressive power to match a producer over the abstract type. For consumers, the obligation is the same, only now it must match the behaviour of the consumer when passed inn' instead of *in*, which means it must also discard *Empty* branches. For example, this means s must now have patterns

$$
\begin{aligned}
&s\ (\underline{Branch}\ Empty\ r) &&= r \\
&s\ (\underline{Branch}\ l\ Empty) &&= l \\
&s\ (\underline{Branch}\ l\ r) &&= l + r
\end{aligned}
$$

Of course, because $s\ (\underline{Empty})$ equals the neutral value 0 anyway, the original definition yields the same results, but this definition reflects various cases encountered. Unfortunately, this same trick is not possible in the Cochurch encoding, because unfolds

produce trees top-down instead of bottom-up. This modification, however, shows the utility of breaking the isomorphism between an abstract datatype and the concrete, fusible type.

This use of a more expressive underlying datatype is similar to how stream fusion works. Stream fusion also uses Cochurch encodings, but instead of using the Cochurch encoding of a list, it uses the Cochurch of a recursive datatype whose base functor is

$$\begin{aligned}&\textbf{data}\ \underline{\mathit{Step}}\ a\ b = \underline{\mathit{Done}}\\ &\qquad\qquad\quad \mid \underline{\mathit{Skip}}\ b\\ &\qquad\qquad\quad \mid \underline{\mathit{Yield}}\ a\ b\end{aligned}$$

The corresponding recursive datatype is similar to that of list, except that the $\underline{\mathit{Skip}}$ constructors allows for members of the list that have no value, but still point to the tail:

$$\begin{aligned}&\textbf{data}\ \mathit{StepList}\ a = \mathit{Done}\\ &\qquad\qquad\quad \mid \mathit{Skip}\ (\mathit{StepList}\ a)\\ &\qquad\qquad\quad \mid \mathit{Yield}\ a\ (\mathit{StepList}\ a)\end{aligned}$$

To convert to and from lists, the functions *stream* and *unstream* are used, as depicted in Chapter 2. One criticism of stream fusion is that *unstream* would appear to have some unstructured recursion, making it difficult to reason about. Recall the definition of *unstream*, rephrased to use the base functor $\underline{\mathit{Step}}$

$$\begin{aligned}&\mathit{unstream} :: \mathit{Stream}\ a \to [a]\\ &\mathit{unstream}\ (\mathit{Stream}\ h\ s) = \mathit{go}\ s\\ &\quad \textbf{where}\\ &\qquad \mathit{go}\ s = \textbf{case}\ h\ s\ \textbf{of}\\ &\qquad\quad \underline{\mathit{Yield}}\ x\ s' \to x : \mathit{go}\ s'\\ &\qquad\quad \underline{\mathit{Skip}}\ s' \quad\ \to \mathit{go}\ s'\\ &\qquad\quad \underline{\mathit{Done}} \quad\ \to [\,]\end{aligned}$$

The recursive call for $\underline{\mathit{Skip}}$ would appear to obscure any reasoning about this function in a structured way, because this is hidden away instead of being handled by the recursion of the unfold. The unfold for *StepList* has the definition

$$\begin{aligned}&\mathit{unfold}_s :: (s \to \underline{\mathit{Step}}\ a\ s) \to s \to \mathit{StepList}\ a\\ &\mathit{unfold}_s\ h\ s = \textbf{case}\ h\ s\ \textbf{of}\\ &\quad \underline{\mathit{Yield}}\ x\ s' \to\ \mathit{Yield}\ x\ (\mathit{unfold}_s\ h\ s')\end{aligned}$$

$$\begin{aligned}
\underline{Skip}\ s' &\to Skip\ (unfold_s\ h\ s') \\
\underline{Done} &\to Done
\end{aligned}$$

The definitions are *almost* the same, except that the recursive call for $\underline{Skip}$ now wraps that in the corresponding constructor for *StepList*, and notably does so for *Skip*s. From here, however, it is possible to write a function that converts a *StepList* into a normal list:

$$\begin{aligned}
fromSL\ (Yield\ x\ xs) &= x : fromSL\ xs \\
fromSL\ (Skip\ xs) &= fromSL\ xs \\
fromSL\ Done &= [\,]
\end{aligned}$$

Such a function is a *StepList* fold with the algebra

$$\begin{aligned}
sl :: \underline{Step}\ a\ [a] &\to [a] \\
sl\ (\underline{Yield}\ x\ xs) &= x : xs \\
sl\ (\underline{Skip}\ xs) &= xs \\
sl\ \underline{Done} &= [\,]
\end{aligned}$$

This is the same conversion algebra as shown in Chapter 3, and in fact demonstrates the connection between that theoretical model and the typical implementation of stream fusion. The conversion of *Stream*s, or the Cochurch encoding of *StepList*s, to lists, is actually an unfold followed by a fold. If deforested into a hylomorphism, it has the form of *unstream*.

Having grafted some structure onto the conversion from *Stream* to list, it has become clear that not one, but *two* representation changes occur. The first converts lists to and from *StepLists*, and the second converts *StepList*s to Cochurch encodings and back. The fusion of these conversions has previously obscured this idea. Now, the proof obligation becomes a two-step process, but each step still fits into this approach.

First, it must be proved that $fromSL \circ toSL = id$, where $toSL :: [a] \to StepList\ a$ is the corresponding function that converts a list into a *StepList*:

$$\begin{aligned}
toSL :: [a] &\to StepList\ a \\
toSL\ [\,] &= Done \\
toSL\ (x : xs) &= Yield\ x\ (toSL\ xs)
\end{aligned}$$

This function simply swaps in *Yield* for (:) and [ ] for *Done*. Since there are no *Skip*s for such a list, the conversion back simply swaps them back. Second, the relationship

between the Cochurch encoding and *StepList*s can be proven using the techniques above.

The process is the same for functions. If the goal is to prove that $t \circ \mathit{unstream} = \mathit{unstream} \circ t_S$, where t_S, is a stream transformer, the transformation can be bridged by proving that $t_{SL} \circ \mathit{fromS} = \mathit{fromS} \circ t_S$ and $t \circ \mathit{fromSL} = \mathit{fromSL} \circ t_{SL}$. This 'bridging' function is similar to t itself but preserves the *Skip*s. These conditions are sufficient to prove $t \circ \mathit{unstream} = \mathit{unstream} \circ t_S$, where $\mathit{unstream} = \mathit{fromSL} \circ \mathit{fromS}$:

$$
\begin{aligned}
& t \circ \mathit{unstream} \\
= & \quad \{ \;\; \mathit{unstream} = \mathit{fromSL} \circ \mathit{fromS} \;\; \} \\
& t \circ \mathit{fromSL} \circ \mathit{fromS} \\
= & \quad \{ \;\; t \circ \mathit{fromSL} = \mathit{fromSL} \circ t_{SL} \;\; \} \\
& \mathit{fromSL} \circ t_{SL} \circ \mathit{fromS} \\
= & \quad \{ \;\; t_{SL} \circ \mathit{fromS} = \mathit{fromS} \circ t_S \;\; \} \\
& \mathit{fromSL} \circ \mathit{fromS} \circ t_S \\
= & \quad \{ \;\; \mathit{unstream} = \mathit{fromSL} \circ \mathit{fromS} \;\; \} \\
& \mathit{unstream} \circ t_S
\end{aligned}
$$

Likewise, this two-step process can be applied to producers and consumers. Furthermore, however, the relationship between *StepList*s and lists refers back to the model given in Chapter 3, which demonstrates how the relationship between *StepList*s and lists can be modelled and how proofs obligations for interface functions can be satisfied.

4.4 Conclusions

In the previous chapter, the shortcut fusion techniques laid out in Chapter 2 were modelled in a category-theoretic setting. The result was that it was possible to directly compare them without having to filter out the syntactic noise that accompanied their original implementations. The insight gained from this effort was that, although the recursion schemes used in these fusion techniques differed, they possessed a unifying feature, which was the notion of representation change. With the exception of stream fusion, this was accomplished implicitly rather than explicitly. In this chapter, a fusion framework was created with the notion that representation change is central to

the concept shortcut fusion. This allowed for a recursion-scheme-agnostic characterisation of shortcut fusion that formally defined it as a general program transformation technique of which the others are instantiations.

One of the benefits of this is that the proof obligations could be stated generally. It also revealed that, while the 'shortcut' of free theorems seen in *foldr/build* and *destroy/unfoldr* provide one method for justifying correctness, these are special cases in this thesis's definition rather than a fundamental feature. Furthermore, it was shown that these techniques could be justified *without* the use of free theorems, which, while useful, do come with certain caveats. Instead of relying on the underlying representation to provide the necessary assurances, this onus has been moved onto the interface functions themselves. By instantiating this framework with the techniques discussed, a connection was demonstrated between the theoretical model in Chapter 3 and the practical aspects of using shortcut fusion in Haskell.

Now that this new setup for implementing shortcut fusion has been created, it paves the way for new applications of shortcut fusion that reach out beyond the representations discussed here. It is possible that other program transformations fit into this framework as well as provide inspiration for new ones. Of particular interest are those cases where, like stream fusion, the concrete representation is not isomorphic to but still faithfully represents the abstract datatype. For example, stream fusion has been use as a 'concrete' representation for transformations over arrays and encoded data.

Chapter 5

A declarative infrastructure for implementing shortcut fusion

In the previous chapter, a general framework was given for shortcut fusion as a program transformation strategy, along with examples of its instantiation. Currently, such fusible libraries are usually implemented with the Glasgow Haskell Compiler (GHC) in mind [Peyton Jones and Marlow, 2012]. GHC already has an existing infrastructure for implementing shortcut fusion; namely, it has a way to specify the algebraic transformations, and its optimisation phase already incorporates the necessary low level transformations to fuse nonrecursive portions of programs. This was the compiler in which shortcut fusion was first implemented, with *foldr/build* relying on GHC to fuse programs without any major modification to the compiler itself.[1] This meant that the complexity of making changes to the compiler itself was avoided, but it also means that fusion implementations are rather *ad hoc*. Local transformations, if they happen in the right places, in the right order, will fuse functions written in the *foldr/build* style. This makes the fusion transformation rather fragile. Using pragmas, some control can be exerted to prevent optimisations from competing with each other, but there is no way to specify that fusion is the desired outcome. Verifying that fusion has occurred in a given program can require inspecting the compiler output itself, and when fusion fails it can be difficult to figure what is interfering with the transformations.

Shortcut fusion, as previously stated, has grown to encompass multiple techniques and has been implemented in a variety of libraries (discussed further in Chapter 6), and, it has been demonstrated, can be generalised to a single, general program transformation. As will be shown in this chapter, the same goes for some of the pragmatic

[1]The first implementation of *foldr/build* was actually hardwired into the compiler, but then later reimplemented when an infrastructure for algebraic transformations was introduced.

aspects of fusion. The underlying transformations and the order in which they must occur is similar for various forms of shortcut fusion. Such an insight can be used to set up a declarative, fusion-specific interface that allows the programmer to declare what they want to occur without micromanaging the pragmatics. With information about the programmer's intention, it is possible for the compiler to 'do the right thing' without any further intervention.

In this chapter, the pragmatic commonalities among the fusion techniques will be exploited in order to create an alternative infrastructure for fusion. This approach uses a declarative-style frontend that allows the programmer to specify the conversion functions and fusible functions, and a backend that uses this information to perform the necessary transformations in the right order. The infrastructure is implemented in the Utrecht Haskell Compiler. It has no previous support for fusion, and performs very few optimisations at all. Therefore, it serves as a clean slate where interactions with unrelated optimisations do not have to be considered.

The work in this chapter has not yet been published, but was funded by a University of Utrecht Short Stay Fellowship. In contrast with the rest of this thesis, this project was supervised by Atze Dijkstra at the University of Utrecht.

5.1 Background

5.1.1 Fusion in GHC

The fusion infrastructure of GHC serves as the inspiration for the work in this chapter, and this section will give a brief overview of its pragmas and transformations. A detailed example, which shows an implementation for fusion of the *Tree* data type of Chapter 4, can be found in Appendix A.

In GHC, the transformations that concern fusion are performed in a phase of compilation handled by GHC's *simplifier* [Peyton Jones and Santos, 1998]. The simplifier performs a series of local optimisations over GHC's core language, which is a desugared, stripped down version of Haskell. Without any intervention, the simplifier performs inlining, beta-reduction, and other transformations as it spots opportunities during traversals of the abstract syntax tree. Heuristics are used to determine whether a transformation ought to actually be applied or not, since some can be pessimations in certain situations. The simplifier also stops after a certain number of (user-adjustable) passes, because it is not known whether the transformations constitute a terminating system, and some pathological cases could cause the inliner to loop.

Because the simplifier is a general optimisation phase, it does not apply the local transformations with a single, targeted goal in mind. For the purposes of fusion, this can be problematic, because it means optimisations may be preemptively performed that prevent fusion from taking place later on, or it may not inline a function that needs to be inlined in order to spot opportunities for applying algebraic transformations and subsequent fusion transformations.

This is solved by giving the programmer some control over how the simplifier behaves through the use of compiler pragmas. One allows the programmer to specify the necessary algebraic transformations for shortcut fusion as so-called 'rewrite rules' [Peyton Jones et al., 2001]. For example, the rewrite rule for stream fusion can be stated as

```
{-# RULES "stream/unstream fusion"
      forall s. stream (unstream s) = s  --Not sound in general!
#-}
```

This pragma instructs the compiler to replaces occurrences of the left hand side with the right. This rule specifies how a rewrite is performed, but does nothing to alter the compiler's inlining behaviour to either to expose opportunities to apply it, nor prevent these functions from being inlined away before the rewrite rule can be applied.

There are two possible cases in which rewrite rules will not be applied even when they could be. The first case is if the functions that are composed together to produce the rewritable expression are never inlined, therefore never exposing the resulting expression in the first place. This is solved with the `{-# INLINE #-}` pragma, which takes a function name and instructs the simplifier to inline it as much as possible. The second case is when, even if the enclosing functions are inlined properly, the *stream* and *unstream* functions themselves are inlined away before the rewrite rule is applied. This is solved with a variant of the inline pragma, `{-# INLINE [0] #-}`. This takes a function name in does not inline it until pass 0, which is always the final simplifier pass. The assumption made is that this gives the simplifier plenty of opportunities to apply the rewrite rule, and if the function remains by the final pass, it may be inlined to look for non-fusion optimisations.

Even with these pragmas in place, fusion is not guaranteed to occur. In fact, sometimes it is necessary to place inline pragmas on subexpressions of fusible functions in order to convince the simplifier to inline the necessary parts of the function to achieve fusion. Going through this process involves inspecting GHC's core output to ensure that fusion occurs or, if not, attempt to diagnose the problems. This a problem

for the library writer, who must take extra care to check that functions are going to fuse, and for the library user, who may end up with undiagnosable performance problems if the fusion transformations fail in some situations but not others.

5.1.2 The Utrecht Haskell Compiler

The Utrecht Haskell Compiler (UHC) is a Haskell compiler that is designed to be extensible for experimentation with new language features [Dijkstra et al., 2009]. At a high level, UHC compiles Haskell to machine code using a 'compilation by transformation' approach, in which the program is transformed into a series of intermediate languages as part of the compilation process. The languages are summarized in the list below.

- Haskell
- Essential Haskell: A desugared, higher order, polymorphic, lazy functional language
- Core: An untyped, lazy functional language, similar to lambda calculus
- Grin: A virtual machine instruction set with strict semantics
- Silly: a small, C-like imperative language

Of these languages, this chapter is concerned with the Haskell and Core languages. The former is where the frontend declarations are implemented and used by the programmer, and the latter is where all the transformations actually take place. Transformations from one language to another, as well as transformations within a given language, are specified as *attribute grammars*, from which code is generated using the Utrecht University Attribute Grammar Compiler (UUAGC) [Swierstra et al.]. For the rest of this chapter, a basic familiarity with attribute grammars is assumed, but the basic syntax of a UUAGC specification will be reviewed, as they are used to define fusion transformations later on.

5.1.3 The Utrecht University Attribute Grammar Compiler

A UUAGC specification is made up of three types of declarations. The first is a datatype declaration, similar to a Haskell one. For example, a list with integers as elements can declared as

```
DATA List
  | Nil
  | Cons hd : Int tl : List
```

Datatype fields area always named. An attribute is declared with the **ATTR** keyword. For example, the sum of a *List* can be declared as a synthesised attribute as follows:

```
ATTR List [|| sum : Int]
```

After the **ATTR** keyword, the name of the datatype is specified, followed by names of attributes and their types in brackets. Attributes before the first pipe are inherited, those in between the two are chained, and those after the second pipe are synthesised, like *sum*. The calculation of attributes is then specified by a **SEM** declaration:

```
SEM List
  | Nil   lhs.sum = 0
  | Cons  lhs.sum = @hd + @tl.sum
```

The name **lhs**, short for 'left-hand side', signifies that the attribute will be calculated and pushed upward, so that it can be accessed by parent nodes. The right-hand side can contain any Haskell expressions matching the type of the attribute being calculated. Additionally, the @-sign is used to signify the names of datatype fields, and to access synthesized attributes associated with the named field using the dot (.) operator.

As an example of using an inherited attribute, consider taking a *List* and wanting to replace each element with the difference between it and its previous element (i.e. a trivial delta encoding). The predecessor of the current element will be passed down as the inherited attribute *prev*, and the resulting list of numbers will be produced as the synthesised attribute *diffs*:

```
ATTR List [prev : Int || diffs : List]
SEM List
  | Nil   lhs.diffs = Nil
  | Cons  tl.prev   = @hd
  | Cons  lhs.diffs = ( @hd - @lhs.prev) : @lhs.diffs
```

For inherited attributes, the left hand side contains the name of field that the attributed is being pushed down to and the name of the attribute itself. Inherited

attributes are referred to on the right hand side using **@lhs**. A single **ATTR** declaration can declare multiple attributes, and a single **SEM** declaration can contain definitions for multiple attributes, both inherited and synthesised.

UUAGC also provides some conveniences, and one in particular that will be used in subsequent sections. When a data structure is transformed into another of the same type, some fields are often left untouched. It can therefore be laborious to exhaustively define cases for every single constructor, only to have their data copied without any computation. Therefore, UUAGC provides the **SELF** keyword, which can be used as the type of a synthesized attribute. If used, only cases for those constructors whose data is not being directly copied need to be specified.

As an example, the attribute *diffs* can be declared using the keyword **SELF** instead of *List*

ATTR *List* [|| *diffs* : **SELF**]

In this case, the *Nil* case could be elided and UUAGC would generate the necessary code automatically.

The code generated by UUAGC is wrapped in a function that takes any inherited attributes as arguments and returns the synthesised attributes as results. In the case of the UHC compiler, these functions are then hooked into the compiler at the appropriate stage. For this work, attribute grammars will be used to specify transformations over UHC's Core language.

5.1.4 UHC Core

UHC Core is a functional language in which the Haskell syntax of the original program has been desugared and transformed into smaller, more regular constructions. This makes transformations over the code simpler and easier to both write and understand. As an example of a Core program, the $length :: [a] \rightarrow Int$ function in Haskell is compiled into the code shown in Figure 5.1. This code demonstrates the common structure of a program in Core. The abstract syntax that represents this program is defined by UUAGC datatype declarations. At the top level, a program is module, which consists of a name and an expression, given by the declaration

DATA *CModule* = *Mod nm* : *expr* : *CExpr*

Core expressions are similar to those of the lambda calculus. They can be constants, variables, abstractions, and applications:

Figure 5.1 The *length* function compiled down to UHC Core

```
module M =
let rec { M.length = λM.x1 →
  let ! { M._3_42_0 = M.x1 } in
    case M._3_42_0 of
      { UHC.Base.: M.x M.xs →
        let { M._5_8 = M.length M.xs } in
          let { M._5_5 = (UHC.Base.+)
                         UHC.Base.Num
                         (#Int "1" :: Int)
                         (M._5_8 :: Int) } in
            (M._5_5 :: Int)
      ; UHC.Base.[]      → (#Int "0" :: Int)
      ; default          → ⊥ }
} in ...
```

```
DATA CExpr = Int    int    : Int
           | Char   char   : Char
           | String str    : String
           | Var    name   : HsName
           | Lam    arg    : HsName body  : CExpr
           | App    func   : CExpr   arg   : CExpr
           | Case   expr   : CExpr   alts  : [CAlt]   dflt : CExpr
           | Let    categ  : Categ   binds : [CBind]  body : CExpr
```

In an expression, subexpressions are bound to names using **let**-clauses. Let clauses have three categories, plain, recursive (indicated by the **rec** keyword), and strict (as indicated by an exclamation point (!)). Recursive bindings are used for recursive and mutually recursive expressions. Strict bindings indicate that the expression must be evaluated strictly. The most common use of these bindings is for **case** expressions. Unlike GHC core, the **case** keyword does not automatically imply evaluation of the expression being analysed, but it is an invariant of UHC core that all **case** scrutinees must be variables that are bound strictly. All bindings are expressed via **let**-clauses, and all pattern matching uses **case** expressions. Functions with multiple arguments are turned into a series of lambda (*Lam*) expressions, each with a single argument. Likewise, function application (*App*) expressions represent the application of an expression to a single variable.

The above description only delves into a subset of the Core language, as there are also representations for tuples, constructors, etc, but the constructs discussed are the ones that are involved in the fusion transformations. Using the **SELF** keyword, code dealing with rest of the constructors is simply auto-generated, therefore other constructs of Core play no significant role in this development.

5.2 Design

In Section 5.1.1, the fusion procedure for GHC was reviewed, which showed that GHC can perform fusion, but its simplifier needs significant tuning to make this occur with any degree of reliability. The reason for all of this tuning is so that inlining, rewrite rules, and local transformations all fire *in the right order*. It is not the local transformations themselves that are the issue, but rather the manner in which they are applied. Therefore, these local transformations can be used as the building blocks for a more predictable fusion algorithm.

From the procedure outlined in Section 5.1.1, it is possible to extract a general order of transformations for fusion:

1. Inline *only* fusible functions
2. Attempt to remove functions according to specified algebraic transformation(s)
3. Inline remaining subexpressions
4. Apply local transformations

As will be shown, the latter two steps occur repeatedly and are often interleaved, because inlining can expose the opportunity to apply other local transformations, which can in turn reveal more inlinable expressions, and so forth.

From the description above, it is obvious that the order of transformations is relatively fixed, and that the treatment of functions falls into three categories: functions mentioned in the algebraic transformations, functions that are meant to be fused, and subexpressions of these fusible functions. If it is known which functions fall into which category, the rest of the transformation proceeds in a mechanical way. The role of a function can be easily declared by the programmar, and this notion is the main motivation behind the declarative front-end implemented in UHC.

5.2.1 The Frontend

In GHC, the compiler pragmas allow the programmer to control the simplifier to the extent that fusion can occur, but they do not allow the programmer to explicitly state which functions should be fused; the fusion is simply a side effect of an overall optimisation strategy rather than a specific goal. The result is a rather fragile transformation that can be difficult to verify and require several rounds of simplifier tuning to get right.

The overall procedure for tuning, however, is uniform for shortcut fusion regardless of the technique instantiated. Therefore, it is simple to figure out *how* to fuse something if the programmer specifies *what* should be fused. In particular, the compiler needs to know which functions should be considered part of the fusible interface to a datatype and which functions act as conversion functions between the abstract and concrete representations. To this end, the Haskell syntax of UHC is extended with two new keywords. First, given a function f, the declaration

fuse f

indicates that f, when applied to an expression, should be fused. If f is not applied, but appears, for example, as an argument to a higher-order function, the fusion declaration has no effect. This is because any possible fusion opportunities would have to be exposed by inlining the enclosing expression. There is nothing preventing this from happening, but it means that the effect of fusion transformation becomes even wider spread, or the inlining must be done by some other optimisation phase. This detail is somewhat tangential to the focus of the infrastructure, so is not considered here. The second keyword introduced is used to declare those functions that convert between representations:

convert *con*, *abs*

It has the meaning that, in expressions containing fusible functions where one consumes the result of another, $abs\,(con\,x)$ is replaced by x. Typechecking ensures that the original and transformed expressions have the same type, but makes no guarantees about the correctness of the transformation. It is assumed that fusible functions have one of the forms specified in Chapter 4, where *abs* and *con* appear immediately in the definition of f and are not nested in a subexpression.

Together, these two keywords provide the programmer with the necessary declarative power to inform the compiler of their intention. This information is propagated

through the intermediate languages via a map of names to 'roles' (a datatype declaration describing a possible role of a function in fusion). This frontend provides a more declarative way for the programmer to set up shortcut fusion. As will be shown, this information is sufficient to automate the rest of the process.

5.2.2 The Backend

The backend of the fusion extension, where the actual transformations take place, is implemented as a transformation over UHC's Core language. The transformation itself consists of a rewrite phase and a fusion phase. In the rewrite phase, all *App* nodes of the abstract syntax tree are checked for cases where one fusible function consumes the result of another. In such cases, fusible functions are inlined and β-reduced, but all subexpressions are left otherwise intact. The result is then checked for opportunities to remove conversion functions, which are then removed if possible.

This is but one way of using the information declared by the programmer. If two functions marked as fusible are defined as described above and then composed, the algorithm finds the rewrite opportunities. This is a very common use case within libraries that use shortcut fusion, and so has wide applicability despite its simplicity. Rewrite opportunities that would require inspecting more deeply nested subexpressions will not be found, but this is not a fundamental limitation of the declarative approach taken, and the algorithm could be made more sophisticated to accommodate such cases.

The structure of the fusion phase can be broken down into four simple, local transformations that are applied recursively in order to achieve fusion. The description of these transformations is based on description of these optimisations as they are applied to GHC's core language [Peyton Jones and Santos, 1995]. However, they are described here as applied to UHC core language for the purposes of fusion, and some relevant diversions from the original transformations are discussed when necessary. One of the key transformations discussed repeatedly is inlining. Inlining is simply the replacement of a bound variable by its definition:

$$\textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Longrightarrow \textbf{let}\, x = e_1 \,\textbf{in}\, e_2\,[x \setminus e_1]$$

In UHC's Core language, all expressions are bound by **let** statements, where (mutually) recursive definitions must be explicitly indicated using **let rec**. This makes it simple to restrict inlining to nonrecursive functions, which is done to prevent infinite inlining (termination is discussed in greater detail in Section 5.3).

The purpose of inlining is expose opportunities for β-reduction. If the inlined expression is a lambda, that lambda function can be applied to its arguments:

$$(\lambda x \to e_1)\, e_2 \Longrightarrow e_1\,[\,x \setminus e_2\,]$$

In UHC Core, all lambdas have only one argument, and a function with multiple arguments becomes a lambda that takes one argument and returns a new lambda, and so forth, so the above transformation takes care of all situations.

Used together, inlining and β-reduction reveal the definition of a nonrecursive function and partially compute the result of applying it. One characteristic of shortcut fusion is that inlining fusible interface functions results in a pipeline of nonrecursive functions, where a function either calls the previous one in certain way (as is the case in stream fusion and *destroy/unfoldr*, where a stepper function always calls the previous one) or a function always has a placeholder for the next function in the pipeline (which is the case in *foldr/build* fusion). In the first case, inlining a pipeline reveals a series of nested **case** statements, where an expression is matched against an inner case statement, and then this intermediate result is immediately pattern matched again. It is possible to make this operational sequence more explicit using the **case-of-case** transformation[2], which 'pushes' an outer **case** statement into the inner one:

$$\begin{array}{l}
\textbf{let !}\,\{\,x = \textbf{let !}\,\{\,y = e_1\,\}\,\textbf{in} \\
\qquad\qquad \textbf{case}\,y\,\textbf{of}\,\{\,p_1 \to r_1\ \ldots\ p_m \to r_m\,\}\,\}\,\textbf{in} \\
\textbf{case}\,x\,\textbf{of} \\
\quad q_1 \to s_1 \\
\quad \vdots \\
\quad q_n \to s_n
\end{array}$$

$\Longrightarrow$

$$\begin{array}{l}
\textbf{let !}\,\{\,y = e_1\,\}\,\textbf{in} \\
\textbf{case}\,y\,\textbf{of} \\
\quad p_1 \to \textbf{let !}\,\{\,x_1 = r_1\,\}\,\textbf{in case}\,x_1\,\textbf{of}\,\{\,q_1 \to s_1\ \ldots\ q_n \to s_n\,\} \\
\quad \vdots \\
\quad p_m \to \textbf{let !}\,\{\,x_m = r_m\,\}\,\textbf{in case}\,x_m\,\textbf{of}\,\{\,q_1 \to s_1\ \ldots\ q_n \to s_n\,\}
\end{array}$$

[2]The bolded words indicate the keywords involved in the transformation.

This has been modified to cope with the specifics of UHC Core, namely the invariant that **case** scrutinees must be strictly bound expressions. The change in strict bindings does not effect the strictness of the overall expression. In both instances, the expression is strict in both the scrutinee and the result of first (originally inner) **case** statement. It is also an invariant of UHC core that all variable names are unique, so we need not worry about variable capture. One of the caveats of this transformation is that the case alternatives for the outer case are duplicated m times. This can be mitigated by floating out s_1 to s_n as **let** statements. In the case of shortcut fusion, this step is unnecessary, because **case-of-case** is only providing the set up for the final transformation: **case-of**-known constructor.

So far, no constructors have been removed in any of the transformations. Inlining simply copies code, and beta reduction does same. The **case-of-case** transformation changes the syntax of a statement, but does not make it more efficient (and can also cause further code duplication). The purpose of all of them is to expose the inner workings of nonrecursive definitions so that the **case-of**-known constructor transformation can be applied. If the constructor of a case scrutinee is known, then a matching case alternative can be selected, the constructor discarded, and the fields of the constructor substituted in for the bound variables in the alternative:

$$
\begin{array}{l}
\textbf{let}\ !\,\{\,x = C\ e_1\ \ldots\ e_n\,\}\ \textbf{in} \\
\textbf{case}\ x\ \textbf{of} \\
\quad \vdots \\
\quad C\ x_1\ \ldots\ x_n \rightarrow E \\
\quad \vdots
\end{array}
$$

$$\Longrightarrow$$

$$E[x_1 \ldots x_n \backslash e_1 \ldots e_n]$$

Voilà! A constructor has been removed.

The transformations presented in this section achieve fusion by occurring in a specific order: inlining allows for β-reduction, which in term exposes nested **case** statements. The **case-of-case** transformation pushes these cases through, which then creates situations for the **case-of**-known constructor. This simple, fixed order can be applied to create a predictable fusion transformation that can be implemented using attribute grammars, and it turns out that this is well-suited to providing a more predictable fusion algorithm.

5.3 Implementation

The transformations discussed in Section 5.2.2 are local, meaning that they only affect small blocks of code within a program. The fusion algorithm implemented in UHC works by rewriting them. What distinguishes this fusion approach from the GHC simplifier is that this implementation works by composing these transformations in a specific order and targets the removal of intermediate data structures in code that has been written according to the setup specified in Chapter 4. In this section, the implementation of the transformations from the previous section will be discussed as well as how the use of UHC Core and attribute grammars affected this implementation.

5.3.1 Rewriting

As outlined, the first step in fusing a program is to remove conversion functions according to the specification described in Section 5.2. This is not strictly a part of the fusion process itself, but is a prerequisite. As mentioned in Section 5.2.1, the rewrite system is designed to reflect the kinds of laws formulated in Chapter 4. This results in a simpler implementation than the full-blown rewrite rules in GHC, but is sufficient for specifying the kind of algebraic transformations necessary for shortcut fusion.

As with all phases of this fusion algorithm, rewriting only applies in situations where fusible functions are involved; it will not go looking for opportunities for rewrites if the functions involved have not been declared as fusible. If a fusible function consumes the result of another fusible function, however, the result of applying the outer function to the inner function is stored in the *applied* attribute:

```
ATTR CodeAGItf [|| applied : CModule]
ATTR AllCodeNT [|| applied : SELF]

SEM CExpr
  | App lhs.applied =  @applied
  | App loc.applied = if @func.isFusible
    apply @func.applied @arg.applied
    else acoreApp1 @func.applied @arg.applied
  | Var lhs.applied = if @isFusible
    then fromMaybe @applied (Map.lookup (acbrefNm @ref) @lhs.fuseMp)
    else @applied
```

The *applied* attribute inlines any *Var*s that represent fusible functions. The attribute @*isFusible* is *True* at variables that name fusible functions. Because a multiargument function is desugared into multiple, single-argument applications, however, a fusible function may be composed of a chain of *App* nodes. Therefore, the attribute is passed upward through unbroken chains of applications, i.e. an *App* is part of a fusible function if its function is fusible. The *App* case of the *applied* attribute applies the inlined code to its argument, performing β-reduction. The attribute is also stored at each *App* node as a *local* attribute. This means that the result calculated in that attribute is stored at that node and can be referred to in other attributes.

In this case, the *applied* is used locally by the *rewritten* attribute:

```
ATTR CodeAGItf [|| rewritten : CModule]
ATTR AllCodeNT [|| rewritten : SELF]

SEM CExpr
  | App lhs.rewritten =  @loc.rewritten
  | App loc.rewritten = if @func.isFusible ∧ @arg.isFusible
    then rewrite @lhs.convMp @loc.applied
    else acoreApp1 @func.rewritten @arg.rewritten
```

This attribute is responsible for rewriting the inlined code. For each *App* node, the attribute checks that a fusible function is being applied to another and, if so, calls *rewrite*. The *rewrite* function takes two arguments. The second is the locally stored *applied* attribute, which contains the result of β-reducing this function. The first is an inherited attribute that contains a map of function names to conversion roles. This is used in the *rewrite* function to look for rewriteable expressions and remove them. If none are found, the *applied* result is discarded, and the results of rewriting the *App* node's subexpressions are used to construct a new *App* node. As with *applied* this is stored locally, as the result of rewriting an expression is not the final result but instead used as the starting point to attempt fusion.

5.3.2 Fusion

After the conversion functions have been rewritten, the original program has now been prepared for fusion; functions have been inlined to expose possible rewrite opportunities, and these opportunities have been exploited so that the recursive functions that would impede fusion have been removed. The result of this is now passed into a fusion function:

```
ATTR CodeAGItf [|| inlined : CModule]
ATTR AllCodeNT [|| inlined : SELF]

SEM CExpr
  | App lhs.inlined = if @func.isFusible
                      then inline @lhs.funcMp @lhs.convMp @loc.rewritten
                      else acoreApp1 @func.inlined @arg.inlined
```

The *funcMp* attribute passed to *inline* contains a map of named functions to their bodies, which are collected from let bindings and passed to blocks of code where those bindings are in scope. This collection is limited to those functions that are considered safe to inline. As stated, the criterion used to determine safety is whether or not the function is recursive or mutually recursive, as indicated by the **let** binding. This is assumed to prevent inlining that would result in non-termination, although the determined programmer can still create pathological programs that would be inlined infinitely. This vulnerability is not unique to this implementation and also exists in e.g. GHC's inliner. Such inlining is furthermore rather aggressive, and there is a worry that inlining will result in the duplication of work. In this implementation, however, inlining is only applied during and with the goal of fusion. It is therefore assumed that the benefits of aggressive fusion outweigh the possible downsides. This is, however, a fundamental assumption when using shortcut fusion, including in GHC, because the compiler must aggressively inline expressions in order to expose fusion opportunities.

The call to *inline* is also the entry point for the actual fusion algorithm. The transformed expression is defined by the *inlined* attribute. As a synthesized attribute, the algorithm operates in a bottom-up fashion, i.e. subexpressions are fused before the enclosing expression is. In the base case, variables are inlined by looking up the corresponding expression in the *Map* that collects nonrecursive expressions and pushes them down the tree:

```
SEM CExpr
  | Var lhs.inlined = if @isInlinable
                      then fromJust (Map.lookup (acbrefNm @ref)
                                                @lhs.funcMp)
                      else @inlined
```

Inlining unfused expressions would violate what was just said regarding fused subexpressions, however. Therefore, the *convMp* actually stores the fused version of expressions, which are lazily computed if actually inlined. Because these expressions are

nonrecursive, they can be computed before fusing the expression they are mentioned in because they do not depend on or contain it.

Inlining exposes lambdas, which in turn can be applied to their arguments to expose fusion opportunities. Therefore, any time a variable is inlined, and its fused result turns out to be a lambda, it is β-reduced unconditionally:

```
SEM CExpr
  | App lhs.inlined = if @func.isInlinable
                      then inline @lhs.funcMp @lhs.convMp
                                  (apply @func.inlined @arg.inlined)
                      else acoreApp1 @func.inlined @arg.inlined
```

The *inline* function is recursively called on the result of this β-reduction, because this application might reveal new fusion opportunities that should exploited.

Once an expression has been unfolded as much as possible, the real heavy lifting can be done in *Case* expressions:

```
SEM CExpr
  | Case lhs.inlined = maybe (acoreCaseDflt @expr.inlined @alts.inlined
                        (Just @dflt.inlined))
                        (inline @lhs.funcMp @lhs.convMp)
                        (orElse @loc.coc @loc.cok)

SEM CExpr
  | Case loc.cok = do scrutineeExpr ← @loc.scrutineeExpr
                     constExpr ← return scrutineeExpr
                     ctag      ← getConstructor constExpr
                     let cargs = reverse (getArgs constExpr)
                     let alts  = @alts.altPairs
                     cok       ← caseOfKnown ctag cargs alts
                     let cok'  = foldr (λ(cat, bind) e → acoreLetBase cat bind e)
                                   cok
                                   @loc.scrutineeLets
                     return cok'
  | Case loc.coc = do scrutineeExpr ← @loc.scrutineeExpr
                     e                 ← getCaseExpr scrutineeExpr
                     (innerAlts, innerDflt) ← getCaseAlts scrutineeExpr
                     let coc = foldr (λ(categ, binds) expr →
```

```
                                    acoreLetBase categ binds expr)
                          (caseOfCase e @alts.inlined
                             @dflt.inlined innerAlts innerDflt)
                          @loc.scrutineeLets
                        return coc
```

There are two possible transformations that can be applied, **case-of-case** (*coc*) and **case-of**-known constructor (*cok*). At each **case** expression, each of the transformations is attempted, depending on the form the scrutinee. The result of each attempt is stored in a *Maybe* expression, so a successful match and transform is stored, and an unsuccessful one simply stores *Nothing*. According to the invariants of UHC Core language, the scrutinee of a case expression is a variable bound in a strict **let** statement. The *scrutineeExpr* is the result of a search for a non-variable expression, which might follow a chain of strictly bound variables that refer to other variables (such chains are common due to other Core transformations that occur before fusion). If the scrutinee is itself a case expression, the **case-of-case** transformation is performed, pushing the case transformation through. If the scrutinee is a constructor, then the **case-of**-known constructor transformation is performed, removing an allocation. If **case-of-case** needs to be performed, it should be performed before **case-of**-known, according to the order given in Section 5.2.2. This is guaranteed by the use *orElse*, which always tries to apply the first transformation before the second, but these two transformations can never be applied to the same kind of expression, since they expect the case scrutinee to have different forms. When one of them is performed, another recursive call is made to *inline*, since there might be more fusion to be done.

In this implementation, some points might raise questions about termination. An attribute grammar traversal is a fold, which provides the assurance that a single traversal terminates. Each of the local transformations, however, triggers a recursive call. Inlining recursive and mutually recursive functions would then trigger recursive calls *ad infinitum*. As previously stated, this issue is circumvented by excluding recursive and mutually recursive functions from the candidates for inlining. As in GHC, this is considered an acceptable measure to ensure termination. There does exist, however, the possibility of creating a program with no explicitly recursive functions that does not terminate upon inlining. Such issues are much harder to prevent with complete assurance, but are also comparatively much rarer in programs. To address to those cases, however, it is simple to place a limit on how many times the inliner can be allowed to run, and such a solution is used in GHC.

In addition to inlining, the algorithm makes a recursive call after each local transformation is performed. The worry, then, is that the transformations create a cycle of transformations that is repeated forever. However, **case-of-case** is only triggered by nested **case** statements, which then can only trigger **case-of**-known constructor, which, contrary to introducing new **case** statements, actually consumes them. Consuming a **case** statement may bring more case statements together and trigger a new **case-of-case**, but assuming there are a finite number of **case** statements, this can only happen a finite number of times, and does not introduce new **case** statements.

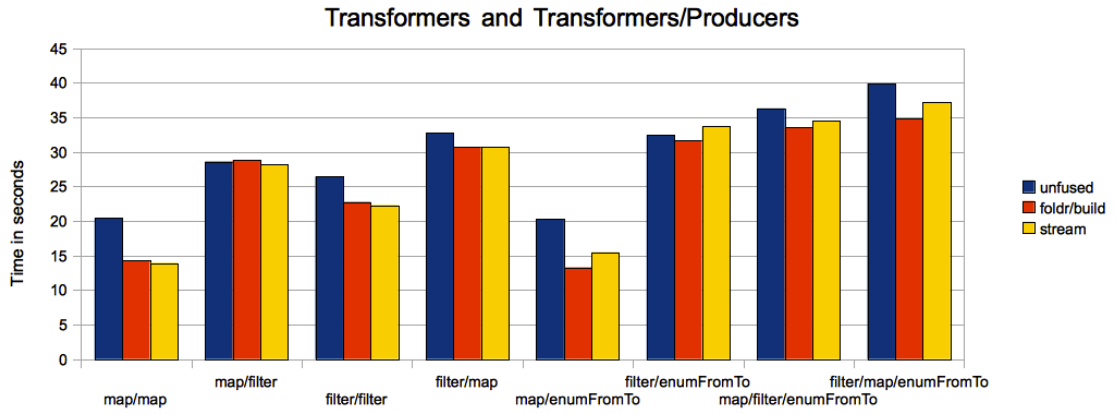
Now, an algorithm has been laid out that can fuse what has been defined as shortcut fusion. Once the recursive combinators have been removed, the remaining parts are nonrecursive functions that can be inlined to expose the subexpressions to a sufficient extent that simple syntactic transformations can finish this job. At this level, the choice of recursion scheme has, to some extent, disappeared. Instead, the only important features are the nonrecursive parts and their syntactic structure. The unifying feature of shortcut fusion here is that these parts either provide a placeholder for the function that appears next in the pipeline or call the one that came before. Via inlining, the algorithm can replace these placeholder variables with the actual functions in the pipeline, which makes it to further syntactic manipulation. This algorithm is therefore suitable for any other technique that expresses functions over a recursive datatype in the same manner, as long as any recursive parts can be encapsulated in combinators that can be unconditionally removed.

5.4 Benchmarks

In previous sections, the necessary steps to perform fusion for shortcut fusion have been identified and implemented in a more predictable, straightforward algorithm. Although the algorithm can fuse the types of programs that have been shown throughout this thesis to remove unnecessary data structures, this does not suffice to show that such transformations actually constitute optimisations. In this section, some benchmarks are presented that test the efficacy of shortcut fusion as an optimisation. The code, as well as the fused Core, can be found in Appendix C.

A set of common list functions are implemented using *foldr/build* and stream fusion, which are the two most common techniques used in the wild. Lists of integers were generated using various implementations of *enumFromTo*. The cost of generating these was benchmarked where a list producer was part of the benchmark. In

Figure 5.2 Benchmarks for fusible transformers and producers



other cases, a pre-generated list was passed into the measured functions. The 'unfused' benchmarks are implementations that were written using explicitly recursively defined functions, whereas the others are fused benchmarks using the techniques indicated in the implementation. The benchmarks were compiled using UHC with the modifications described on a Macbook Pro with a 2.66 GHz Intel Core i7 with 4GB RAM. The UHC compiler itself was compiled using the UUAGC tool along with GHC 7.4.2, as UHC is not capable of compiling itself.

The first set of benchmarks, shown in Figure 5.2, show common compositions of transformers, as well as transformers and producers. The benchmarks show that the impact of using fusion varies in these cases. In general, there are minor improvements, consistent with results on lists for other fusion techniques. There is some variation in the performance between the different fusion techniques. This is not particularly surprising either, given the different practical and pragmatic aspects that have been discussed regarding fusion techniques in previous chapters.

In Figure 5.3, pipelines consumed by *foldr* are shown. Contrasted with the previous benchmarks, it would appear that the difference between fused and unfused code varies more widely, with both the negative and positive impacts much more noticeable. This is similar for the results in Figure 5.4. This suggests that when the output of the program is not a recursive data structure itself, the impact of intermediate data structures is not as strong.

Overall, the result is mixed, with both pessimations and improvements in all situations. This is not necessarily an issue with the fusion algorithm itself but rather something more fundamental with shortcut fusion. By itself, shortcut fusion is not always an optimisation. Rather, the impact of fusion, both in terms of quality and

Figure 5.3 Benchmarks for various *foldr*-consumed pipelines

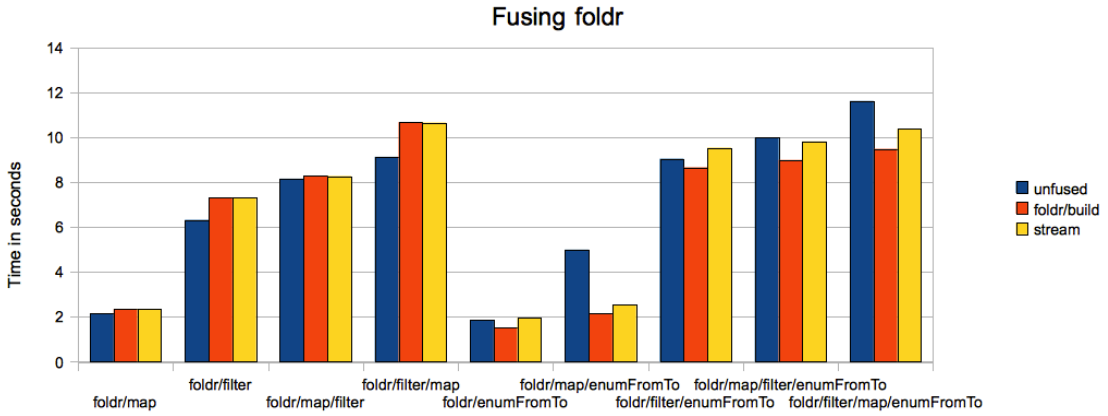
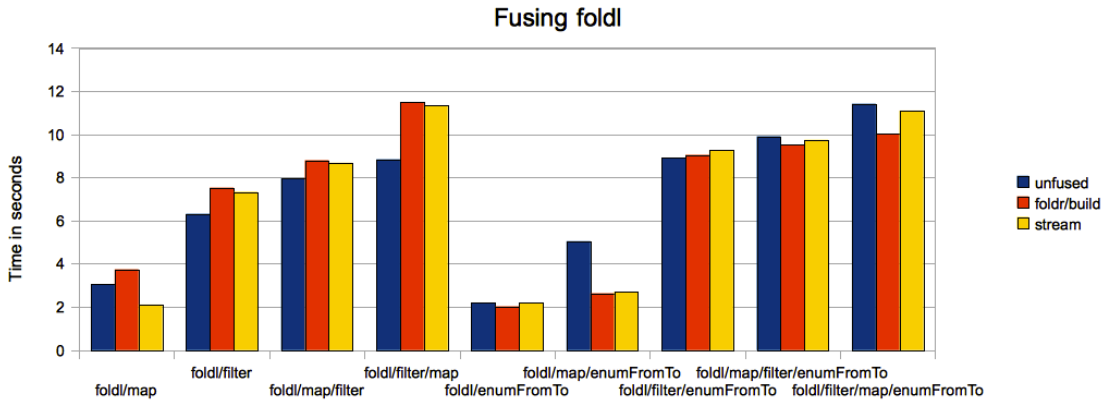


Figure 5.4 Benchmarks for various *foldl*-consumed pipelines



magnitude, is based on several environmental variables. The cost of intermediate data structures can vary with the structure used, with strict arrays, for example, having a much more visible impact than lazy lists. The degree to which fusion limits or allows knock-on optimisations (e.g. unboxing) also influences performance figures. It is, therefore, often considered one tool among many that can increase performance when engineering a data structure and API. Some applications in which fusion has been used with much more obviously positive results are discussed in Chapter 6. UHC, however, has relatively few other optimisations implemented, meaning that these benchmarks may more accurately reflect the performance impact of fusion in isolation as opposed to fusion in conjunction with other optimisations, as is often presented. The benchmarks' mixed results are consistent with this understanding of fusion.

5.5 Conclusions

In Chapter 4, the canonical shortcut fusion techniques were examined, and their commonalities used to show how shortcut fusion can be seen as a general program transformation strategy. In the course of showing how this transformation can be instantiated for various different recursion schemes, the details of optimisation were left to the abstract notion of 'compiler optimisations'. The pragmatics of exactly these optimisations were the topic of this chapter.

Shortcut fusion arose to take advantage of existing compiler optimisations. This was convenient for implementation purposes, because it did not require any changes to the compiler, but also meant that program transformations only happened 'in passing' and required significant tuning to get better guarantees. Furthermore, 'shortcut fusion' itself has not hitherto been seen as a single program transformation for which a single solution could be applied. As a result of the generalisation of Chapter 4, however, such an algorithm could be synthesized by tying together the local transformations in a more targeted fashion. The local transformations that served as the inspiration remained the same, but it turns out that the tuning that the programmer often does for every implementation can be generalised. The result is an algorithm that can transform fusible functions by utilising information the programmer provides about their setup without instructing the compiler *how* it should accomplish fusion, and leaving the rest of the program untouched, which was one of touted benefits of shortcut fusion in the first place.

This notion could be extended to be much more powerful than it has been in this particular work. Once the compiler becomes 'fusion-aware', it can be more aggressive in executing the programmer's intentions beyond the simple syntactic transformations applied here. For example, the fusion pass could provide feedback when fusion fails where it ought to have succeeded e.g. because of a recursive subexpression, or provide metrics for how many constructors were successfully removed. Such power is due to the declarative nature of the frontend, which captures the general setup for shortcut fusion that was established previously. The frontend could also capture the proof obligations for the programmer and attempt to prove them automatically, or provide some assistance in doing so.

The benchmarks show that the fused code performs as expected; fusion, even when applicable, is not always an optimisation when applied naïvely, and even some automation of the setup and transformation cannot escape the performance pitfalls that the implementer is expected navigate in refining a data structure and interface. The reasons for this are varied, and have been covered in the descriptions of the individual fusion techniques themselves and their applications. What they amount to, however, is that being able to write a function in a fusible form does not always lead to a faster program, because factors such as the nature of the data structure itself, other optimisations, and others can interact with or effect fusion, thereby effecting the final out. Despite the lack of a hard and fast guarantee, shortcut fusion (and fusion in general) is nevertheless a useful tool for the library author looking to increase performance. The work in this chapter serves to help solidify the notion that each of these techniques is an instance of a general program transformation strategy for which a targeted infrastructure can be created that facilitates its use.

Chapter 6

Related work

This chapter reviews fusion-related work not already discussed earlier. The most closely related work, the three shortcut fusion techniques analysed, was covered in detail in Chapter 2. This chapter will cover other relevant work related to the development, application, and analysis of shortcut fusion techniques (including those analysed), as well some competing approaches to removing intermediate data structures.

6.1 Shortcut fusion techniques and applications

While Chapter 2 laid out three fusion techniques in terms of lists, stream fusion in particular has been used in other applications. Before being 'ported' to lists, stream fusion was used to fuse operations over arrays, specifically traversals of strings represented as arrays for performance purposes [Coutts et al., 2007b]. This was a particularly successful use of fusion because these arrays were strict, and the allocation of intermediate arrays had a much more visible affect on performance. One of the less emphasized contributions of this application was that, through the use of an explicit *Stream* datatype, the connection to data abstraction was made much more obvious, as was the distinction between the recursion scheme being encoded by the 'concrete' datatype and the underlying data structure. This served as the inspiration for a further extension of this data abstraction idea, which was utilised in abstracting away encoding details in Unicode strings stored in arrays [Harper, 2009]. In this implementation, not only was the traversal of the underlying array encapsulated, but the encoding and decoding of the string, which was stored as an encoded Unicode string, as well.

Functional array fusion [Chakravarty and Keller, 2001] also adapted the idea of fusion to array traversals, actually serving as a pre-cursor to stream fusion. Like

stream fusion, it also uses explicit types to represent conversion between different representations of arrays. In addition to its two combinators, *replicateP* and *loopP*, some rewrite rules fuse array traversals by introducing new combinators. Nevertheless, they reduced the problem of fusion to representing traversals by a small number of fixed combinators that can be used to implement array algorithms. Their explicit use of a different type for implicit (i.e. fusible) array representations strongly resembles the style adopted in this thesis, and hints strongly that this technique would easily be phrased in terms of the data abstraction framework in this thesis and provide a route for proving correctness, something not dealt with rigorously by the authors. Array fusion research has continued since stream fusion was developed and has been refined and extended as part of the Data Parallel Haskell project [Chakravarty and Leshchinskiy, 2007].

This data abstraction notion is also made explicit for stream fusion by Coutts [2011] in his thesis about stream fusion. In it, data abstraction is central to his proof of correctness, leveraging the concept in a similar manner to the work in Chapter 4. In this thesis, the idea has been recast to show that it is a more general characteristic of shortcut fusion rather than unique to the stream fusion. In particular, the notion of identifying an explicit alternate representation, upon which proofs can be framed using data abstraction, applies to other shortcut fusion techniques as well. With proof obligations framed in terms of data abstraction, correctness proofs can also differ. Coutts uses fixpoint induction to satisfy his established proof obligations, while this thesis relates these obligations back to the setting chosen in Chapter 3. The ability to show these as two different methods for satisfying the same obligations is a strength of this characterisation of shortcut fusion. In other aspects, as well, Coutts focusses on the specifics of stream fusion, which often leads to stronger, but less general results. This is the case in discussing when stream fusion is an optimisation and in the definitions of 'good consumers' and 'good producers'. Such conditions are specific to the technique used, so it is difficult to make general statements about these ideas.

In addition to the shortcut fusion techniques analysed, others have been developed along similar lines. As with those analysed, they to take their cues from their predecessors, usually attempting to increase expressibility or fuse a new kind of traversal or data structure. Fernandes et al. [2007] extended *foldr/build* fusion to fuse folds with circular dependencies. They leverage the separation between production and consumption in *foldr/build* fusion to develop a calculational method for writing circular programs using two combinators which can then fused. Martínez and Pardo similarly extended *foldr/build* so that it could fuse functions where an accumulator

is used to produce the result, although this required defining those functions so that the list they consumed would not be fused. As with their predecessor, they depend on parametricity for correctness, with similar caveats.

There has also been particular attention paid to the fusion of programs with monadic computations. The use of folds in monadic computations was explored by Meijer and Jeuring, who developed monadic fold from the standard one, and likewise developed associated laws, including fusion [Meijer and Jeuring, 1995]. This naturally leads to an investigation of extending *foldr/build* fusion to cope with monadic computation. Manzino and Pardo developed an extension to *foldr/build*, similar to the one for circular dependencies, that provides monadic variants of functions *foldr* and *build*. These functions consume a structure to produce a monadic computation and construct a monadic computation that produces a data structure, respectively [Manzino and Pardo, 2008]. These combinators, when used with each other and with their non-monadic counterparts, can be fused to eliminate the intermediate data structures created within a monadic computation. This work arises out of Pardo's work on monadic deforestation using hylomorphisms, discussed in Section 6.3. In contrast to Pardo, who sought to remove the intermediate data structures within a monadic computation, Ghani et al. extended *foldr/build* to remove the monad as well, leaving a pure value as a result [Ghani et al., 2005]. This involves writing programs in terms of *augment*, a generalisation of *build* that Gill presented in his thesis on *foldr/build* [Gill, 1996], and showing that it can be defined in terms of *build* and monadic bind.

Delbianco et al. also explored *foldr/build* fusion for computational effects modelled as applicative functors [Delbianco et al., 2012]. They derive a combinator that encapsulates a fold for lists that performs an effectful computation over each of the elements and a corresponding *build* function. With these combinators, the intermediate data structures formed by composing two functions with applicative effects could be removed. As with the original version of *foldr/build*, their technique generalises to other datatypes. The technique is derived from the ability to phrase applicative traversals in terms of the usual *fold* function, and the correctness relies on the correctness of the *foldr/build* rule rather than a more formal or structured approach.

With all of these approaches, the focus is on adding a new feature to the original combinators. The proofs of correctness usually rely on the original *foldr/build* law being correct, or they appeal directly to parametricity. In terms of the framework of Chapter 4, they differ in which underlying representation they use. The proof obliga-

tions therefore remain the same in this framework, although the details of satisfying them differ.

6.2 Reasoning about shortcut fusion transformations

Originally, *foldr/build* relied on free theorems, and by extension parametricity, for its correctness. This was seen as an advantage, since *foldr/build* fusion simplified not only the pragmatics of deforestation, but came with a simple proof. This was also the route taken by the various extensions to *foldr/build*, as well as *destroy/unfoldr*. The validity of using free theorems as justification has since been questioned, since free theorems do not hold unconditionally for Haskell. Further, in the case of stream fusion, free theorems are no longer sufficient justification. Consequently, the issue of how to prove correctness for various shortcut fusion techniques has been addressed with a variety of approaches.

Johann made progress on the correctness of *foldr/build*, utilising a result that allowed her to prove the correctness of *foldr/build* for languages with higher-order polymorphism and fixpoint recursion [Johann, 2003]. She used a proof technique called 'contextual equivalence' to do this, using the fact that the existence of parametric models of contextual equivalence for such languages had been found. This formalised the folklore of correctness in cases where parametricity holds, as well as the correctness of generalising the transformation to other algebraic datatypes.

Another approach has been to use categorical semantics to model fusion transformations, since there are category-theoretic models of algebraic datatypes, folds, and unfolds. Ghani et al. provide a categorical semantics for *fold/build* and *destroy/unfold*, comparing them with *fold/in* and *out/unfold* as the traditional syntax for data types defined modelled as initial algebras and final coalgebras, respectively [Ghani and Uustalu, 2004]. This allows them to state the properties possessed by *build* and*destroy* that make the two alternative syntaxes correct, and they are able to justify that they are equally expressive.

Johann and Ghani use a category-theoretic setting to develop *foldr/build* fusion for generalised folds, i.e. folds over nested datatypes [Johann and Ghani, 2007]. The result is a fusion rule for generalised folds by providing a corresponding *build* function, and showing how initial algebra semantics are sufficient to model generalised folds. While this extends *foldr/build* fusion further, it is perhaps just as significant that they explicitly recognise the role of Church encodings in *foldr/build* fusion, and the presentation of Church encodings for nested datatypes drives their development.

They continue, however, to rely on parametricity, and acknowledge that relying on parametricity for correctness is a result of the isomorphism of between Church encodings and their underlying datatype. While this is beneficial to their development, it continues to separate stream fusion and its variations from *foldr/build* fusion despite their obvious links. In the use of categorical semantics in this paper, the goal has not been to make new inroads into correctness, but to able to tie together the progress made in modelling these transformations before.

In his work, Voigtländer addresses issues of correctness and pragmatics many times, attempting to address the issue of correctness directly in Haskell, rather than using theoretical settings. He addresses strictness, and its impact on correctness proofs that rely upon parametricity, using *foldr/build* as a specific example and is able formally state the restrictions that must be placed on the rule in order for it to be correct in Haskell, as well as the *destroy/unfoldr* rule. He has also developed a *build/destroy* rule [Voigtländer, 2008], the correctness of which is proven in a similar manner using parametricity with restrictions. Unlike other rules, this does not fully fuse a fold followed by an unfold, something which is not unconditionally possible, but removes the overhead of 'repackaging' the underlying list in different representations. Later, he refines *foldr/build* and *destroy/unfoldr* rules by introducing lambda abstractions that prevent changes in strictness, but full fusion of the code requires more compiler transformations. The same trick changes Pardo et al's circular fusion into a technique that does not rely on laziness by using higher-order functions, a possibility they alluded to as a possible alternative. At the core of Voigtländer's work, however, lies a continued reliance on parametricity. As a consequence, stream fusion has not been subject to the same analysis.

As has been stated, parametricity separates some fusion techniques from others, so depending on it for correctness has been avoided. One of the features the framework established in Chapter 4 is to sidestep some of these issues parametricity encounters, namely that functions can be written that break fusion transformations. Instead, correctness is ensured by establishing properties that the functions to which the program transformation might be applied must respect, which restricts them from engaging in program-breaking behaviour.

In a related vein, Gibbons considers correctness neither through a data refinement framework nor through parametricity, but using an alternate view of equality between *Stream*s and lists called *observational equivalence* [Gibbons, 2008]. This is grounded in the idea that *Stream*s are an abstraction that represent lists as codata, and it is possible to define and prove equivalence 'modulo *Skip*s' between *Stream*s and lists.

Gibbons uses final coalgebra semantics and the universal property of unfolds for this proof. This notion of equivalence between the abstract and concrete datatypes is not directly addressed in the framework established in Chapter 4, because it compares instances of the abstract datatype and emphasises the properties of the conversion and interface functions rather than the underlying structures.

Gill and Hutton have also introduced a program transformation framework based on data abstraction, called the worker/wrapper transformation [Gill and Hutton, 2009]. This framework focusses on the use of data refinement, similarly to Chapter 4, but focusses on changing the data structure used by recursive functions. In shortcut fusion, the approach is generally to eliminate the need for recursive functions through using data refinement. This makes these approaches related by their use of data abstraction but orthogonal in how they apply it. They extended the transformation to deal with folds [Hutton et al., 2010], basing their reasoning on initial algebra semantics, but this involved changing the data structure being folded over, again emphasising the changing of the representation of the data structure being recursed over rather than attempting to remove recursion, as is the case with shortcut fusion.

6.3 Alternative approaches to deforestation

With the problem of fusing recursive functions presenting fundamental challenges, shortcut fusion is not the only approach that has attempted bring automated deforestation into a more tractable realm. An early example is Wadler's Deforestation Algorithm, mentioned in Chapter 1 [Wadler, 1988]. This is a syntax-driven approach, with the restrictions on programs phrased in syntactic terms, as was the justification for the transformation. A similar approach was taken by Voigtländer, but with the goal of fusing programs that required multiple traversals of a data structure into a circular program [Voigtländer, 2004]. Like Wadler, the largely syntactic-driven approach requires significant complexity to implement and requires a restricted language to guarantee correctness.

Such complexities are indeed a driving force behind more structured approaches. Before *foldr/build* fusion, Sheard recognised that writing programs using structured recursion, specifically folds, offered opportunities for automated optimisation [Sheard and Fegaras, 1993]. He created a 'normalisation' algorithm that could fuse functions that were written using folds. Like the Deforestation Algorithm, this places restrictions on the forms of programs that can be written in order for the algorithm to be

correct, but uses structured recursion to characterise the restriction on terms and the expressibility of these restricted programs.

This idea of mixing structured recursion with an optimisation algorithm remained a popular theme in fusion. An approach called 'warm fusion' attempted to automate the programming style required for *foldr/build* fusion [Launchbury and Sheard, 1995]. Instead of requiring programmers to program in a specific style, warm fusion attempts to derive *build* and *fold* functions for datatypes, and then use them to rewrite functions written in an explicitly recursive style. These definitions can then be used to fuse programs using a program transformation that removes *fold* and *build* pairs and fuse the result in a single phase. The entire transformation is implemented as a preprocessor. Because it is entirely based on folds, it suffers from the same drawbacks as *foldr/build* fusion. Nemeth implemented warm fusion in GHC itself, addressing the pragmatics of incorporating the warm fusion algorithm into a compiler [Nemeth, 2000]. Nemeth's approach was less general than the one taken in Chapter 5 in that he focussed only on catamorphisms, but it is worth noting that he ran into similar problems complexities with respect to inlining, optimisation, and less definitive results than traditional compiler optimisations. In perhaps the most structured approach of all, Chitil gave a type-driven approach to fusion [Chitil, 1999]. He presented an algorithm that inspected expressions being consumed by folds and used type inference to determined which list constructors must be abstracted away to create the appropriate *build*-based expression. Furthermore, Chitil claims to be able to identify precisely the functions that need to be inlined based on this type inference. This is similar to the attempt to limit the side effects of fusion presented in Chapter 5 by targeting only potentially fusible functions, though Chitil is able to further automate this process by focussing on a specific fusion technique.

The idea of deriving fusible forms of functions was also investigated by Takano and Meijer, who proposed using hylomorphisms in order to perform fusion in what they called 'calculational form' [Takano and Meijer, 1995]. They presented hylomorphisms in an algebraically compact setting where initial objects and final objects coincide. They use the acid rain theorems that were also presented in Chapter 3 and show how they can be used to fuse programs explicitly written using a hylo combinator. In contrast to our work in Chapter 3, the use of hylomorphisms was as a recursion scheme that could be used to rewrite explicitly recursive functions in a more general way than *foldr/build* and *destroy/unfoldr* fusion rather than for the purpose of modelling shortcut fusion, for which a different setting, recursive coalgebras, was chosen. They also incorrectly asserted that the original hylo-ana rule was sufficient to fuse both

data structures consumed by a *zip*, an error that was later remedied by Hu et al., who derived and presented the parallel hylo-ana rule [Hu et al., 1996]. Hu et al. did not dualise this idea, however, to the cata-hylo rule presented in this thesis. Takano and Meijer's calculational approach was also extended by Pardo to fuse intermediate data structures generated by monadic computations [Pardo, 2001].

Onoue et al. revisited the idea of fusion via using hylomorphisms in their implementation of the HYLO fusion system [Onoue and Hu, 1997]. The approach is similar to warm fusion, but using hylomorphisms and fusion laws as presented by Takano and Meijer. Because hylomorphisms generalise both folds and unfolds, this system is able to capture the recursion scheme of and fuse more programs that warm fusion. The HYLO system was extended by Domínguez and Pardo [2006] to use paramorphisms and a scheme they call *generalised paramorphisms*, a concept which, like hylomorphisms, allows for the preparation of the arguments by an coalgebra. They show that this approach is more expressive than hylomorphisms (as paramorphisms can be used to fuse a larger set of functions than folds). They note, however, that there are cases in which applying fusion to an unfused algorithm actually worsens the *complexity* of the program, for which the authors propose some sort code analysis to detect. They leave this, as well as benchmarks to test performance, as future work. Pardo also suggested that the HYLO system could easily be extended to accommodate his monadic extension mentioned above.

The idea of using fusion to speed up programs written in functional languages has not been limited to Haskell. In striking similarity to stream fusion, With-Loop fusion fuses array traversals in the Single Assignment C programming language Grelck et al. [2006]. The language already requires the programmer to manipulate arrays using predefined operations, and fusion can be performed by transforming occurrences of composed operators. This was accomplished by extending the internal representation of array traversals to be able to model multiple operations in a single traversal. This requires that the internal representation of an array traversal be general enough to capture all of the patterns the programmer works with, in the same way that the expressibility of shortcut fusion is limited by the recursion scheme used. The Clean programming language uses a heavily automated approach to deforestation, very similar to Wadler's, from which they draw inspiration Van Arkel et al. [2002]. They attempt to perform transformations on Clean's core language, using certain safety criteria to determine if an expression should be inlined and possibly fused. These safety criteria attempt to address both issues of termination and duplication of work. The result is that the compiler can safely attempt to fuse arbitrary programs,

but that some fusion opportunities (sometimes a large percentage thereof) can be missed. The idea of automated fusion has also not been limited to the composition of recursive functions. Shivers and Might have taken a similar approach to transducers in Scheme, where they convert composed transducers into continuation-passing style, which is more amenable to manipulation and optimisation [Shivers and Might, 2006]. Similarly to warm fusion and the HYLO system, they attempt to take more idiomatically written programs and preprocess them into a more structured format, and then continue optimisation in that vein.

Supercompilation is related to deforestation in that it removes intermediate data structures while also subsuming a host of other program optimisations [Turchin, 1986]. It works by 'partially executing' a program by inlining functions, including recursive ones, and evaluating and simplifying the result. Interest has recently reawakened in supercompilation for Haskell [Mitchell and Runciman, 2008, Bolingbroke and Peyton Jones, 2010], with promising performance results in experiments. Because it inlines recursive functions, one of the largest challenges faced in supercompilation is deciding when to terminate the transformation, and a significant amount of research in this area focusses on this particular problem. Issues of termination, as well as significant code size bloat and compilation times make this approach impractical for the time being.

Similar program transformation strategies have also been applied to other languages. Paige and Koenig applied the notion of *finite differencing* to programming languages [Paige and Koenig, 1982]. Phrased in very general terms, if an expression $E = f\,(x_1, \ldots, x_n)$ is computed repeatedly in a program region (e.g. a loop), it could be lifted out and computed once, but not if one or more arguments changes each time. However, if the difference between successive versions of E can be computed based on the differences in the changing subterms, then this can be used instead of recomputing the entire expression. This can often constitute an optimisation, reducing memory usage or even reducing algorithmic complexity. As with supercompilation, deforestation is a possible side-effect rather than the stated goal of this transformation. Paige and Koenig present this strategy in very general terms that could be applied at various levels of abstraction in a variety of languages. Like shortcut fusion, the transformation steps can be stated in terms of simpler, chained transformations, each of which is semantics-preserving on its own. The details of these transformations are left to the implementation languages. At this level, the transformations are discussed at the syntactic level, much like those in Chapter 5. In comparison to shortcut fusion, finite differencing does not necessarily have deforestation as a goal, although

this is one possible effect of this optimisation. One of the challenges of this technique is identifying when calculating differences is cheaper than the original expressions and actually constitutes an optimisation, a complexity shared by shortcut fusion.

The differencing approach was revisited and extended Liu et al., where finite differencing (also called *incrementalisation*) paired this with *static caching* [Liu et al., 1998]. This comes from the observation that certain intermediate values in the recomputation E are reused, and can therefore be cached to provide even better performance. This is followed again by a series of transformations, although each is rather complex. First, all subresults are captured, and then they are pruned so that only the useful ones remain. This is accomplished by way of static analysis. Like Paige and Koenig, Liu et al. are able to phrase the overall approach in a general way, but the details quickly become language specific. They consider their approach, as many other techniques, as a candidate to be put in an optimising compiler, meaning that the general case of all possible programs must be considered. This has been applied to the Python programming language [Python Software Foundation], in which a variety of problem domains have been explored, namely set comprehensions [Rothamel and Liu, 2008], abstract data types represented as objects [Liu et al., 2005] and, more concretely, a role-based access control library [Liu et al., 2006]. The results achieve increased performance, although Liu does not distinguish between gains from improvements in algorithmic complexity versus pure memory savings. As with regular incrementalisation, this is not always a clear win in terms of increased performance and in some cases can be a pessimation because of the increased memory used for caching recomputed values. Its efficacy is also highly dependent not only on the semantics of the original program but also the style in which it is written, as with other program transformations. Various applications of incrementalisation and its interaction with program design are discussed in Liu [2013].

In contrast with shortcut fusion, most of these attempt to achieve much more automation or have a much wider scope for optimisation. In many cases, the programmer is free to write explicitly recursive functions and leave the rest to the deforestation algorithm. Although this frees the programmer of some extra work, it makes the deforestation task much more complex. Even so, in many cases, the language under which these systems can work is still restricted, at least in the implementations presented. Furthermore, unless rigorous guarantees can be made about both correctness and success of these algorithms, the programmer is still left with some unpredictability about how the resulting program will perform. The more structured, targeted approach of shortcut fusion circumvents these issues by requiring the programmer

to do some of the work by programming in such a highly stylised way. While this is more onerous for the programmer, the relative simplicity of the pragmatics of shortcut fusion make it immediately applicable and practical.

Chapter 7

Conclusions

The central argument of this thesis arose out of the observation that, despite differences that distinguish various shortcut fusion techniques, the similarities between them was undeniable. When such connections were explored, it became clear that these program transformations were variations on the same central idea. First, each of these program transformations encapsulates a recursion scheme using combinators for production and consumption. Second, these combinators can be removed via algebraic transformations. Finally, the remaining code is nonrecursive, thereby sidestepping the issue of attempting to inline and optimise recursive functions. This provides a clear definition of 'shortcut fusion' in comparison with other deforestation and fusion techniques.

The three techniques that have been analysed were chosen because of the extent to which they have been used in practical settings and addressed in literature. Both *foldr/build* and stream fusion have enjoyed considerable success in practical applications, and both have been the subject of further research in both practical and theoretical domains. While *destroy/unfoldr* has received comparatively little attention, it played an important role in showing the power of considering alternative recursion schemes and served as an important predecessor to stream fusion, which specifically address some of *destroy/unfoldr*'s limitations. The goal of this thesis was to elucidate and formalise precisely what connected these techniques, and investigate what insight could be gained from such an analysis. To this end, these techniques were analysed on three different levels: theory, practice, and pragmatics.

7.1 Theory

The fact that *foldr/build* is based on a structured form of recursion with strongly established theoretical foundations makes it a natural target for comparison in a

theoretical setting. It has indeed been the subject of no small amount of theoretical discussions, in particular centered around correctness. The attractiveness of such an approach is further increased by its relationship with *destroy/unfoldr*, which is easily explained by the categorical concept of duality. Such theoretical analyses, however, generally focus on issues of correctness. In Chapter 3, the chosen fusion techniques were analysed in a theoretical setting, but the focus was not on correctness itself. Rather, the choice of theoretical setting, hylomorphisms and recursive coalgebras, was exploited as venue in the relationships between the techniques could be explored.

From this viewpoint, theoretical considerations such as parametricity, representation change, and expressibility could be brought together 'under one roof'. This had the result of tying together previous results about parametricity and representation change, but also providing new insights. In particular, it was possible to clarify the informal notion of expressibility; the mechanism by which stream fusion breaks free of *destroy/unfoldr*'s limitations is clearly explained as the result of turning some coalgebras into natural transformations. Furthermore, the data abstraction approach that was chosen to prove stream fusion correct fits into this setting. The recognition of the role of data abstraction, tied together with the representation changes identified for *foldr/build* and *destroy/unfoldr*, proved to be the path to developing a general framework for shortcut fusion.

7.2 Practice

The theory contributions serve to tie together the theoretical foundations that underlie some shortcut fusion techniques. By using various forms of structured recursion, certain properties can be assumed about programs that simplify the fusion process. Different recursion schemes yield different benefits, and it is possible to formalise why and how this is the case. All of this fails to address the question, however, of how each of the techniques discussed, with their differing recursion schemes, is an instance of the *same* program transformation. The answer is to specify such a program transformation in a framework that does not depend on *which* recursion scheme is used, but can be instantiated for a variety of them.

Such a framework was developed in Chapter 4 based on the observation that implementations of each of these techniques bore strong similarities to one another. The syntax that was used to implement each of these techniques originally, however, obscured the most important unifying feature, which was the notion of representation change. The use of Church and Cochurch encodings in *foldr/build* and *destroy/unfoldr*

was implicit, while in stream fusion it was explicit. When the former two are recast in the same manner as stream fusion, the similarities begin to fall into place.

Stream fusion was originally set apart not only by its presentation differences, but also by a structural one, where the use of *Skip* seemingly set it apart by changing the correctness conditions. The use of data abstraction to justify correctness, however, applies equally well to other representations, consequently bringing stream fusion back into the fold. Furthermore, the data abstraction approach allows for more flexibility than parametricity, because it captures the notion that the algebraic transformations are used in a certain context, namely one in which the fusible representation is only created and manipulated by a predefined set of interface functions. This places the proof obligations on the interface functions, but loosens the requirement of isomorphism between fusible representations and their underlying datatypes.

From this, it can be concluded that shortcut fusion is a program transformation of which these techniques are instantiation. The realisation that representation is at the heart of shortcut fusion is a potentially powerful development. It suggests that different representations can be used to suit the needs of a given data structure and interface, something that Data Parallel Haskell has also explored in its use of fusion to model array traversals that do not necessarily conform to the structured recursion seen so far.

7.3 Pragmatics

The general framework we have established captures the similarities in how shortcut fusion is instantiated for different recursion schemes. There are also striking similarities in the pragmatics of these techniques, i.e. how the instantiated techniques are transformed by the compiler. Shortcut fusion has previously depended on an existing compiler infrastructure in order to take care of the fusion process, which is one of its benefits in terms of complexity of implementation. The issue with such a dependency, however, is that the behaviour of a general optimisation phase of a compiler is not necessarily well-aligned with the goal of fusion. If optimisations are applied in the wrong order, the entire program transformation can come crashing down. Compilers are not usually aware of larger, ambient goals when applying local optimisations, though. This necessitates some tuning of compiler optimisations to behave in a certain, ordered way.

By examining the pragmatics of each of the techniques analysed, it becomes quickly apparent that this certain, ordered way is actually the same for all short-

cut fusion techniques. Inlining must be controlled to expose rewrite opportunities, and then must be used aggressively to expose fusion opportunities afterward. Using this observation, the details of the tuning can be abstracted away, leaving an infrastructure that allows the programmer to merely fill-in the details specific to a given implementation of shortcut fusion. This consists of the recursive combinators being used and a description of which functions are meant to be targets for fusion.

To this end, a proof-of-concept of such a system has been implemented in the Utrecht Haskell Compiler. The result is a declarative infrastructure for implementing fusion, where the the programmer can specify only the details of their specific implementation of shortcut fusion, and the compiler is correctly tuned to do the rest. The idea of using such a declarative interface presents other possible opportunities for assisting the programmer in writing fusible programs in terms of correctness, testing, and verification of transformations, because now the infrastructure is aware not only of the behaviour that the programmer desires, but also the intention behind it.

The purpose of looking at fusion on these three levels was to bring insights from each of them together. Previously, there was often a large gap between the theoretical and practical worlds, in particular, where the former was more concerned with correctness and the latter with performance. By starting in the theoretical, and threading insights from it through the rest of the thesis, we hope that at least one bridge has been created between these worlds. A well-chosen theoretical setting can be a useful laboratory for experimenting with ideas about how fusion works. These insights can be used to provide practical improvements by providing some structure in how shortcut fusion techniques are developed and implemented. Finally, in recognition that shortcut fusion continues to be a popular and practical method of improving performance, an improved infrastructure has been prototyped based on the generalisation discovered. As a result, we have unified shortcut fusion both by bringing the techniques themselves together and by bridging their theoretical and practical worlds.

7.4 Future Work

On a theoretical level, the use of recursive coalgebras for modelling fusion transformations opens up the question of what recursiveness means for a functional programming language such as Haskell. While the manipulation of recursive coalgebras is not dependent on which ambient category is chosen (provided it has the necessary characteristics), recursiveness has often been proven indirectly by showing that a coalgebra fulfills an equivalent condition, and such conditions are category-specific.

It therefore remains unclear what recursiveness implies about coalgebra in Haskell. The use of recursive coalgebras, as with hylomorphisms before it, also suggests that there might be applications for in other forms of deforestation by directly rewriting functions. The original results about *Skip*, as well as data abstraction, raises the question about whether or not the introduction of *Skip*s, where necessary, might be possible, and whether the data-abstraction proof obligations related to *Skip*s might be automated for languages with the appropriate semantics. It also remains to be seen how other shortcut fusion techniques, for example the variations of *foldr/build* discussed in Chapter 6, can fit into this model of shortcut fusion transformations.

The practical applications of shortcut fusion remain an open area of research. The revelation that shortcut fusion depends on representation changes opens it up to investigations into other representations and applications it might be useful for. Shortcut fusion has largely focussed on linear sequences, and it is hoped that the framework given can be a starting point for investigation into fusion for other structures and applications rather than the usual suspects.

The infrastructure presented in Chapter 5 takes advantage of the shared pragmatic requirements among shortcut fusion techniques. This removes the need for tuning the compiler, but there are further possibilities for development tool support for implementing and using shortcut fusion. There are several possible avenues for improving automation in the implementation, testing, and verification of shortcut fusion. The algorithms for rewriting and inlining presented are somewhat naïve, but they could be extended to provide more functionality. For example, it could make more principled decisions about whether a function needs to be inlined, and the restrictions placed for spotting rewrite opportunities could be further loosened with more intelligent searching. Because the proof obligations are known for interface functions, there is the possibility of providing some proof assistance to check that fusible implementations are correct. Also, because it is known that the programmer expects fusion in certain situations, there is room for further work in detecting when fusion ought to occur and could not, so that the programmer can receive more information about the success or failure of fusion without inspecting transformed program output themselves.

Overall, one of the goals of this thesis was to provide a better, more structured approach to implementing shortcut fusion. We hope that framework and infrastructure provided drives more research in applications of shortcut fusion, and that further compiler support might be sparked by the idea of continuing to improve the infrastructure available to do so by making use of the shared pragmatics laid out in this works. New applications will then continue to drive theoretical questions about its

underlying foundations. The worlds of theory, practice, and pragmatics will continue to influence one another.

References

J. Adámek, D. Lücke, and S. Milius. Recursive coalgebras of finitary functors. *RAIRO - Theoretical Informatics and Applications*, 41(4):447–462, Aug. 2007. ISSN 0988-3754. doi: 10.1051/ita:2007028.

M. Bolingbroke and S. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell - Haskell '10*, volume 45, page 135, New York, New York, USA, Sept. 2010. ACM Press. ISBN 9781450302524. doi: 10.1145/1863523.1863540.

V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1:1–28, 2005. doi: 10.2168/LMCS-1.

V. Capretta, T. Uustalu, and V. Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, Apr. 2006. ISSN 08905401. doi: 10.1016/j.ic.2005.08.005.

M. M. T. Chakravarty and G. Keller. Functional array fusion. *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, 36(10):205–216, 2001. ISSN 03621340. doi: 10.1145/507546.507661.

M. M. T. Chakravarty and R. Leshchinskiy. Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, 2007.

O. Chitil. Type Inference Builds a Short Cut to Deforestation. *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming ICFP 99*, pages 160–249, 1999.

D. Coutts. *Stream Fusion: Practical shortcut fusion for coinductive types*. PhD thesis, University of Oxford, 2011.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion. *ACM SIGPLAN Notices*, 42(9):315, Oct. 2007a. ISSN 03621340. doi: 10.1145/1291220.1291199.

D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting Haskell Strings. In *PADL '07*, volume 4354, pages 50–64. Springer-Verlag, 2007b.

G. A. Delbianco, M. Jaskelioff, and A. Pardo. Applicative shortcut fusion. In R. Peña and R. Page, editors, *Trends in Functional Programming*, volume 7193 of *Lecture Notes in Computer Science*, pages 179–194, Berlin, Heidelberg, May 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32036-1. doi: 10.1007/978-3-642-32037-8.

A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell - Haskell '09*, pages 93–104, New York, New York, USA, 2009. ACM Press. ISBN 9781605585086. doi: 10.1145/1596638.1596650.

F. Domínguez and A. Pardo. Program fusion with paramorphisms. In *MSFP'06 Proceedings of the 2006 international conference on Mathematically Structured Functional Programming*, July 2006.

J. P. Fernandes, A. Pardo, and J. Saraiva. A shortcut fusion rule for circular program calculation. *Proceedings of the ACM SIGPLAN workshop on Haskell workshop - Haskell '07*, page 95, 2007. doi: 10.1145/1291201.1291216.

P. J. Freyd. Remarks on algebraically compact categories. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, pages 95—-106. LMS Lecture Note Series, 1992.

N. Ghani and T. Uustalu. Build, augment and destroy, universally. *Programming Languages and Systems*, pages 327–347, 2004. doi: 10.1.1.73.7890.

N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic Augment and Generalised Short Cut Fusion. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 294—-305, New York, 2005. ACM.

J. Gibbons. Unfolding abstract datatypes. In P. Audebaud and C. Paulin-Mohring, editors, *MPC '08 Proceedings of the 9th international conference on Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, pages 110–133, Berlin, Heidelberg, July 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70593-2. doi: 10.1007/978-3-540-70594-9.

A. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, The University of Glasgow, 1996.

A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(02):227, 2009. ISSN 09567968. doi: 10.1017/S0956796809007175.

A. Gill, J. Launchbury, and S. Peyton Jones. *A short cut to deforestation*. ACM Press, New York, New York, USA, 1993. ISBN 089791595X. doi: 10.1145/165180.165214.

C. Grelck, K. Hinckfuß, and S.-B. Scholz. With-Loop Fusion for Data Locality and Parallelism. In *Implementation and Application of Functional Languages*, 2006.

T. Harper. Stream Fusion on Haskell Unicode Strings. In M. T. Morazán and S.-B. Scholz, editors, *IFL'09 Proceedings of the 21st international conference on Implementation and application of functional languages*, volume 6041 of *Lecture Notes in Computer Science*, pages 125–140, Berlin, Heidelberg, Sept. 2009. Springer Berlin Heidelberg. ISBN 978-3-642-16477-4. doi: 10.1007/978-3-642-16478-1_8.

T. Harper. A library writer's guide to shortcut fusion. In *Proceedings of the 4th ACM symposium on Haskell - Haskell '11*, page 47, New York, New York, USA, 2011. ACM Press. ISBN 9781450308601. doi: 10.1145/2034675.2034682.

W. L. Harrison. The Essence of Multitasking. In *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology, volume 4019 of Lecture Notes in Computer Science*, pages 158–172. Springer, 2006.

R. Hinze, T. Harper, and D. W. James. Theory and Practice of Fusion. Accepted for publication in Post-proceedings of the 22nd Symposium on the Implementation and Application of Functional Languages (IFL '10), 2010.

R. Hinze, T. Harper, and D. W. James. Theory and Practice of Fusion. Technical report, University of Oxford, 2011.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4): 271–281, 1972. ISSN 00015903. doi: 10.1007/BF00289507.

Z. Hu, H. Iwasaki, and M. Takeichi. An Extension Of The Acid Rain Theorem. In *Proceedings of the 2nd Fuji Int Workshop on Functional and Logic Programming*, pages 1–4. In T Ida, A Ohori, and M Takeichi, eds, Proceedings 2nd Fuji Int Workshop on Functional and Logic Programming, Shonan Village, 1996.

G. Hutton, M. Jaskelioff, and A. Gill. Factorising folds for faster functions. *Journal of Functional Programming*, 20(3-4):353–373, 2010. ISSN 09567968. doi: 10.1017/S0956796810000122.

P. Johann. Short cut fusion is correct. *Journal of functional Programming*, 13(4): 797–814, July 2003. ISSN 09567968. doi: 10.1017/S0956796802004409.

P. Johann and N. Ghani. Initial algebra semantics is enough! In *Typed Lambda Calculi and Applications*, volume 4583 of *Lecture Notes in Computer Science*, pages 207–222, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73227-3. doi: 10.1007/978-3-540-73228-0.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '04*, pages 99–110, New York, New York, USA, 2004. ACM Press. ISBN 158113729X. doi: 10.1145/964001.964010.

J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103(2):151–161, 1968. ISSN 0025-5874. doi: 10.1007/BF01110627.

J. Launchbury and T. Sheard. Warm fusion: deriving build-catas from recursive definitions. *Functional Programming Languages and Computer Architecture*, page 314, 1995.

D. Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98, New York, New York, USA, 1983. ACM Press. ISBN 0897910907. doi: 10.1145/567067.567077.

Y. A. Liu. *Systematic Program Design: From Clarity to Efficiency*. Cambridge University Press, 2013.

Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1998. ISSN 01640925. doi: 10.1145/291889.291895.

Y. A. Liu, S. D. Stoller, M. Gorbovitski, T. Rothamel, and Y. E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 40, pages 473–486, 2005. ISBN 1595930310. doi: 10.1145/1103845.1094848.

Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: efficient implementations by transformations. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–120, 2006.

C. Manzino and A. Pardo. Shortcut Fusion of Monadic Programs. *J. UCS*, 14(21): 3431–3446, 2008. doi: 10.3217/jucs-014-21-3431.

E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In *Advanced Functional Programming, LNCS 925*, pages 228–266, May 1995. ISBN 3-540-59451-5.

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, page 124, 1991.

N. Mitchell and C. Runciman. A Supercompiler for Core Haskell. In *Implementation and Application of Functional Languages 2008*, pages 147–164, 2008. doi: 10.1007/978-3-540-85373-2_9.

L. Nemeth. *Catamorphism-Based Program Transformations for Non-Strict Functional Languages*. Phd thesis, University of Glasgow, 2000.

Y. Onoue and Z. Hu. A Calculational Fusion System HYLO. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 76–106, 1997.

R. Paige and S. Koenig. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982. doi: 10.1145/357172.357177.

A. Pardo. Fusion of recursive programs with computational effects. *Theoretical Computer Science*, 260(1-2):165–207, June 2001. ISSN 03043975. doi: 10.1016/S0304-3975(00)00127-4.

S. Peyton Jones and S. Marlow. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume II*, chapter 5. 2012. URL `http://www.aosabook.org/en/ghc.html`.

S. Peyton Jones and Others. Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming*, 13(01):0–255, Jan. 2003. ISSN 0956-7968.

S. Peyton Jones and A. L. M. Santos. Compilation by Transformation in the Glasgow Haskell Compiler. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer London, London, 1995. ISBN 978-3-540-19914-4. doi: 10.1007/978-1-4471-3573-9.

S. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, Sept. 1998. ISSN 01676423. doi: 10.1016/S0167-6423(97)00029-4.

S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233. ACM SIGPLAN, 2001. doi: 10.1.1.22.1330.

Python Software Foundation. Python Language Reference. Technical report. URL `http://www.python.org`.

J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. *Information Processing 83*, 83:513–523, 1983.

T. Rothamel and Y. a. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th international conference on Generative programming and component engineering - GPCE '08*, page 55. ACM Press, 2008. ISBN 9781605582672. doi: 10.1145/1449913.1449923. URL `http://portal.acm.org/citation.cfm?doid=1449913.1449923`.

T. Sheard and L. Fegaras. A fold for all seasons. *Functional Programming Languages and Computer Architecture*, page 233, 1993.

O. Shivers and M. Might. Continuations and transducer composition. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, volume 41, page 295, 2006. ISBN 1595933204.

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming - ICFP '02*, volume 37, pages 124–132, New York, New York, USA, 2002. ACM Press. ISBN 1581134878. doi: 10.1145/581478.581491.

S. D. Swierstra, P. R. Azero Alcocer, and J. Saraiva. Designing and Implementing Combinator Languages. In *Advanced Functional Programmin*, pages 150–206. doi: 10.1007/10704973_4.

A. Takano and E. Meijer. Shortcut deforestation in calculational form. *Functional Programming Languages and Computer Architecture*, page 306, 1995.

P. Taylor. Towards a unified treatment of induction, I: The general recursion theorem. 1996.

V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, June 1986. ISSN 01640925. doi: 10.1145/5956.5957.

D. Van Arkel, J. Van Groningen, and S. Smetsers. Fusion in practice. In *Proceedings of the 14th international conference on Implementation of functional languages*, pages 51–67, 2002.

J. Voigtländer. Using Circular Programs to Deforest in Accumulating Parameters. *Higher-Order and Symbolic Computation (formerly LISP and Symbolic Computation)*, 17(1/2):129–163, Mar. 2004. ISSN 1388-3690. doi: 10.1023/B:LISP.0000029450.36668.cb.

J. Voigtländer. Proving correctness via free theorems: the case of the destroy/build-rule. *ACM/SIGPLAN Workshop Partial Evaluation and Semantics-Based Program Manipulation*, 2008.

P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. ISBN 978-3-540-19027-1. doi: 10.1007/3-540-19027-9_23.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359, 1989. doi: 10.1.1.38.9875.

M. Wang, J. Gibbons, M. Kazutaka, and Z. Hu. Gradual Refinement: Blending Pattern Matching with Data Abstraction. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction*, pages 397–425. Springer Berlin / Heidelberg, 2010.

Appendix A

Implementing shortcut fusion using GHC pragmas

Chapter 5 introduces a new infrastructure for shortcut fusion, and makes references to the existing infrastructure in the Glasgow Haskell Compiler. This appendix contains a more detailed account of these pragmatics, as they were summarised in Harper [2011], for comparison. This replays the development of fusing leaf trees that appears in Chapter 5, but with reference to the pragmatic aspects of implementing this interface in GHC.

We define leaf trees using following datatype declaration:

$$\begin{aligned}\textbf{data}\ \mathit{Tree}\ a = {} & \mathit{Empty} \\ \mid {} & \mathit{Leaf}\ a \\ \mid {} & \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\end{aligned}$$

To compare the two representations we have discussed, we will develop the interface using both Church and Cochurch encodings in parallel.

A.1 Combinators

To begin, we must instantiate a concrete representation for leaf trees, along with appropriate conversion functions for *con* and *abs*. As discussed in Section 4.2, we can use the Church encoding by defining *toCh* and *fromCh*, or the Cochurch encoding by defining *toCoCh* and *fromCoCh*.

For the Church encoding version, we start by declaring the base functor for *Tree*, which is the same as the datatype declaration but with the recursive definition abstracted away:

$$\textbf{data}\ \underline{\mathit{Tree}}\ a\ b = \underline{\mathit{Empty}} \mid \underline{\mathit{Leaf}}\ a \mid \underline{\mathit{Fork}}\ b\ b$$

Because *Tree* is a polymorphic type, the base functor we are concerned with is $\underline{Tree}\ a$. We are now able to define the Church encoding type over *Tree*s:

$$\textbf{data}\ Tree^{\dagger}\ a = Tree^{\dagger}\ (\forall b.(\underline{Tree}\ a\ b \to b) \to b)$$

Next, we provide the definition for our *con* combinator, *toCh*

$$\begin{aligned}
&toCh :: Tree\ a \to Tree^{\dagger}\ a\\
&toCh\ t = Tree^{\dagger}\ (\lambda a \to fold\ a\ t)\\
&fold :: (\underline{Tree}\ a\ b \to b) \to Tree\ a \to b\\
&fold\ a\ Empty \quad\ = a\ \underline{Empty}\\
&fold\ a\ (Leaf\ x) \ = a\ (\underline{Leaf}\ x)\\
&fold\ a\ (Fork\ l\ r) = a\ (\underline{Fork}\ (fold\ a\ l)\\
&\qquad\qquad\qquad\qquad\quad (fold\ a\ r))
\end{aligned}$$

The *fold* function is simply the fold over *Tree*s, which takes an algebra a and applies it accordingly. For Church encodings, this is where the recursion is built-in to the type.

For *abs*, we use *fromCh*, which reconstructs the tree by applying the Church encoding to the initial algebra *in*:

$$\begin{aligned}
&fromCh :: Tree^{\dagger}\ a \to Tree\ a\\
&fromCh\ (Tree^{\dagger}\ fold) = fold\ in\\
&in :: \underline{Tree}\ a\ (Tree\ a) \to Tree\ a\\
&in\ \underline{Empty} \quad\ = Empty\\
&in\ (\underline{Leaf}\ x) \ = Leaf\ x\\
&in\ (\underline{Fork}\ l\ r) = Fork\ l\ r
\end{aligned}$$

in is defined by having one case per constructor. In this case, single step of construction simply swaps the constructors back to those of *Tree*. The built-in recursion of the $Tree^{\dagger}$ takes care of applying *in* recursively.

Dually, we can instantiate *con* and *abs* with conversion functions to and from the Cochurch encoding. We can use the same base functor as for the Church encoding. An unfold, instead of recursively combining results, creates a tree by branching into two subtrees, yielding a value of the sequence, or simply stopping when the subtree is empty.

Again, from Section 4.2, we get the definition of the Cochurch encoding

$$\textbf{data}\ Tree^{\ddagger}\ a = \exists s.\ Tree^{\ddagger}\ (s \to \underline{Tree}\ a\ s)\ s$$

as a pair consisting of a stepper function and an initial seed.

To convert a tree to its Cochurch encoding, we pair the original tree with a function that describes how to perform single step in the unfold:

$$
\begin{aligned}
&\mathit{toCoCh} :: \mathit{Tree}\ a \to \mathit{Tree}^{\ddagger}\ a \\
&\mathit{toCoCh}\ t = \mathit{Tree}^{\ddagger}\ \mathit{out}\ t \\
&\mathit{out}\ \mathit{Empty} \quad\ = \underline{\mathit{Empty}} \\
&\mathit{out}\ (\mathit{Leaf}\ a) \ = \underline{\mathit{Leaf}}\ a \\
&\mathit{out}\ (\mathit{Fork}\ l\ r) = \underline{\mathit{Fork}}\ l\ r
\end{aligned}
$$

Dual to *in*, *out* describes to destruct a recursive data structure into its components. Going back the other way, we recursively apply the stepper function to each successive seed.

$$
\begin{aligned}
&\mathit{fromCoCh} :: \mathit{Tree}^{\ddagger}\ a \to \mathit{Tree}\ a \\
&\mathit{fromCoCh}\ (\mathit{Tree}^{\ddagger}\ h\ s) = [\![h]\!]\ s \\
&[\![h]\!]\ s = \mathbf{case}\ h\ s\ \mathbf{of} \\
&\quad \underline{\mathit{Empty}} \quad\ \to \mathit{Empty} \\
&\quad \underline{\mathit{Leaf}}\ a \quad\ \to \mathit{Leaf}\ a \\
&\quad \underline{\mathit{Fork}}\ sl\ sr \to \mathit{Fork}\ [\![h]\!]\ sl\ [\![h]\!]\ sr
\end{aligned}
$$

We can again see the duality between these two representations. For Church encodings, the recursion pattern is completely baked into the representation as it is created. The Cochurch encoding, on the other hand, provides a blueprint for how to construct a tree, but the recursion itself appears when the encoding is converted back to a regular tree. In both cases, the proof of correctness comes with the encoding for free, because we have proved that Church and Cochurch encodings and the associated fold and unfold semantics satisfy the proof obligations for fusion.

A.2 Rewrite rules

Now that we have defined representations and conversion functions, and discharged the associated proof obligations, it is time to implement the syntactic transformation. Luckily, GHC provides a relatively simple way to do this with the *RULES* pragma [Peyton Jones et al., 2001]. This pragma allows us to specify an equation in which we replace any instance of the left hand side by the right hand side. For example, we can specify how to remove instances of *fromCh* followed by *toCh* by declaring the rule

```
{-# RULES "toCh/fromCh fusion"
   forall x. toCh (fromCh x) = x #-}
```

The `{-#...#-}` brackets signify that the text in between is a compiler pragma. The keyword *RULES* specifies the name of the pragma, and the string in quotes is simply a unique name for the rule, which can be used to identify it in compiler-generated statistics. The rewrite equation itself begins with the keyword *forall*, which allows us to universally quantify one or more variables in the following equation. Finally, the equation contains a left hand side that we wish to rewrite into the right hand side. In our case, we want to remove the unnecessary conversion of a value, so we remove the conversion functions in the expression replace it with the value itself. We can also define the analogous rule for our Cochurch encoding combinators:

```
{-# RULES "toCoCh/fromCoCh fusion"
  forall x. toCoCh (fromCoCh x) = x #-}
```

The programmer should be aware that the *RULES* pragma comes with almost no guarantees. Aside from checking that the types of the two sides match, the *RULES* pragma does nothing to check the correctness of the transformation. Furthermore, specifying this rule does not guarantee that the compiler will rewrite all (or even any) of the situations where this rule can be applied! This is because encountering these situations depends on GHC inlining functions in a pipeline definition in order to expose the conversion functions, but also not inline away the combinators themselves before the rule can be applied. We can address this issue by fine-tuning how GHC inlines our conversion functions, which we will discuss next.

The first case, ensuring that functions containing these combinators are inlined, is a simple matter of using the *INLINE* pragma to encourage the GHC inliner [Peyton Jones and Santos, 1998] to inline it, even if it might not otherwise. However, it it also possible to tell GHC *when* to inline a function, which is important in order to keep the conversion combinators visible long enough for the simplifier to see them and apply the rewrite rule. To accomplish this, we pass an integer that specifies a phase of the simplifier in the pragma. In doing so, we signal to the simplifier *not* to inline the given function until that phase has been reached. Phases are numbered in decreasing order, with the final being phase 0, so if we specify the pragmas

```
{-# INLINE [0] toCh #-}
{-# INLINE [0] fromCh #-}
```

for our Church encoding combinators and

```
{-# INLINE [0] toCoCh #-}
{-# INLINE [0] fromCoCh #-}
```

for our Cochurch encoding combinators, then we give GHC as long as possible to eliminate these functions. If they still remain by the final phase, we assume that they cannot be fused away and allow GHC to inline and optimise them.

The main principle of writing fusible functions is to use the recursion provided by the concrete representation; recall that the purpose of this whole exercise is to allow us provide nonrecursive definitions for our interface functions so that the compiler does the low-level work for us—introducing extra recursion will stump the compiler. Previously, we divided such functions into three categories based upon their use of the conversion combinators. We also use these divisions in establishing guidelines for fusible functions and will deal with an example of each of them here.

A.3 Producers

Producers are functions that produce a data structure without consuming one. As an example of such a function, we use the function *enum*, which takes a pair of integers and generates the enumeration from the first to the second, inclusive:

$$
\begin{aligned}
&\mathit{enum} :: (\mathit{Int}, \mathit{Int}) \to \mathit{Tree}\ \mathit{Int} \\
&\mathit{enum}\ (x, y) \\
&\quad \mid x > y\ = \mathit{Empty} \\
&\quad \mid x \equiv y = \mathit{Leaf}\ x \\
&\quad \mid x < y\ = \mathit{Fork}\ (\mathit{enum}\ (x, \mathit{mid})) \\
&\qquad\qquad\qquad\quad (\mathit{enum}\ (\mathit{mid} + 1, y)) \\
&\quad \textbf{where} \\
&\qquad \mathit{mid} = (x + y)\ `\mathit{div}`\ 2
\end{aligned}
$$

Written as a Church encoding, nothing is going to build in the recursive pattern for us, as this is usually done by *toCh* when converting an already existing data structure. There, the Church-encoding-based producer must encapsulate the recursive pattern to create the $\mathit{Tree}^{\dagger}$ representation of this tree:

$$
\begin{aligned}
&\mathit{between}^{\dagger} :: (\mathit{Int}, \mathit{Int}) \to \mathit{Tree}^{\dagger}\ \mathit{Int} \\
&\mathit{between}^{\dagger}\ (x, y) = \mathit{Tree}^{\dagger}\ (\lambda a \to \mathit{loop}\ a\ (x, y))
\end{aligned}
$$

```
  where
    loop a (x, y)
      | x > y  = a Empty
      | x ≡ y  = a (Leaf x)
      | x < y  = a (Fork (loop a (x, mid))
                         (loop a (mid + 1, y)))
      where
        mid = (x + y) `div` 2
```

As a Church producer, this is an example of a function that is allowed to have recursion. The *loop* function simultaneously describes how to construct the enumeration as well as the placement of the abstracted algebra a, which describes how to reduce the structure. To actually produce the tree, *fromCh* will apply the $Tree^{\dagger}$ function to *in*:

```
enum' :: (Int, Int) → Tree Int
enum' = fromCh ∘ between†
 {-# INLINE between' #-}
```

If *fromCh* is fused away, however, the next algebra will be applied instead, which means the elements of the enumeration will be consumed without writing out the actual tree. Note the use of the unconditional *INLINE* pragma to ensure that we inline *enum'* so that the *fromCh* is exposed and subsequently removed, if possible.

In the Cochurch encoding, *fromCoCh* encapsulates our recursive pattern, and therefore we supply a nonrecursive coalgebra that describes how to construct a single step:

```
between‡ :: (Int, Int) → Tree‡ Int
between‡ (x, y) = Tree‡ h (x, y)
  where
    h (x, y)
      | x > y  = Empty
      | x ≡ y  = Leaf x
      | x < y  = Fork (x, mid) (mid + 1, y)
      where
        mid = (x + y) `div` 2
enum'' :: (Int, Int) → Tree Int
```

$$enum'' = fromCoCh \circ between^{\ddagger}$$
{-# INLINE between" #-}

We depend on *fromCoCh* to recursively apply c to successive seeds until the tree is fully constructed. If, however, the *fromCoCh* call is fused away, the next function has a description of how to build a tree, which it can use to do so, or, alternatively, consume the values as they are yielded to produce a single value.

A.4 Consumers

Whereas Church producers may be recursive and Cochurch producers may not be, the opposite is true for consumers.

Since the *toCh* function encapsulates the recursive pattern, which describes how to reduce the structure to a single value. For example, the function *sum*

$$\begin{aligned}
&sum :: Tree\ Int \to Int \\
&sum\ Empty \quad\ = 0 \\
&sum\ (Leaf\ x) \quad = x \\
&sum\ (Fork\ x\ y) = sum\ x + sum\ y
\end{aligned}$$

combines a *Tree Int* into a single *Int* by adding the elements together.

The function $sum^{\dagger}$ mirrors this form, but removes the recursive call because folds (and therefore Church encodings) build in the recursion for consumption. Instead, we just have to supply an algebra that describes how to deal with a single step:

$$\begin{aligned}
&sum^{\dagger} :: Tree^{\dagger}\ Int \to Int \\
&sum^{\dagger}\ (Tree^{\dagger}\ g) = g\ s \\
&s :: \underline{Tree}\ Int\ Int \to Int \\
&s\ \underline{Empty} \quad\ = 0 \\
&s\ (\underline{Leaf}\ x) \quad = x \\
&s\ (\underline{Fork}\ x\ y) = x + y \\
&sum' :: Tree\ Int \to Int \\
&sum' = sum^{\dagger} \circ toCh
\end{aligned}$$
{-# INLINE sum' #-}

Instead, it is the Cochurch encoding version, $sum^{\ddagger}$, that must do the recursive work on its own:

```
sum‡ :: Tree‡ Int → Int
sum‡ (Tree‡ h s) = loop s
  where
    loop s = case h s of
      Empty → 0
      Leaf x → x
      Fork l r → loop l + loop r
sum'' :: Tree Int → Int
sum'' = sum‡ ∘ toCoCh
 {-# INLINE sum'' #-}
```

To accomplish this task, $sum^{\ddagger}$ is armed with h, which it can apply to the initial seed s. At each step, the *loop* function consumes the result, either by unwrapping and returning an *Int* or recursively obtaining the results of consuming two of subtrees and adding them together.

We now know how to create trees and consume them using this framework. Now, we move on to functions that are both producers and consumers.

A.5 Transformations

A transformation is a function that consumes a tree in order to produce a new one, and is therefore both a producer and a consumer.

Because of this, they should not be recursive in either representation, instead always using the built-in recursive pattern.

As examples of transformations, we develop *Tree* versions of two familiar functions: *reverse* and *filter*. First, take the function *reverse*, which reverses the ordering of the elements of the tree:

```
reverse :: Tree a → Tree a
reverse Empty     = Empty
reverse (Leaf a)  = Leaf a
reverse (Fork l r) = Fork r l
```

The *reverse* function recursive swaps all subtrees and leaves the leaf elements intact.

As discussed in Section 4.2, such functions take the form of a transformation composed with either an algebra or coalgebra. In the case of *reverse*, this transformation defines a single swap of subtrees:

$$
\begin{aligned}
&r :: \underline{Tree}\ a\ c \to \underline{Tree}\ a\ c \\
&r\ \underline{Empty} \quad\ = \underline{Empty} \\
&r\ (\underline{Leaf}\ a) \ = \underline{Leaf}\ a \\
&r\ (\underline{Fork}\ l\ r) = \underline{Fork}\ r\ l
\end{aligned}
$$

For the Church encoding, $reverse^{\dagger}$ creates a new $Tree^{\dagger}$ that applies the Church encoded input value to the abstracted algebra precomposed with r, i.e transforms swaps the subtrees of a *Fork* and then passes that result on to the next algebra:

$$
\begin{aligned}
&reverse^{\dagger} :: Tree^{\dagger}\ a \to Tree^{\dagger}\ a \\
&reverse^{\dagger}\ (Tree^{\dagger}\ g) = Tree^{\dagger}\ (\lambda a \to g\ (a \circ r)) \\
&reverse' :: Tree\ a \to Tree\ a \\
&reverse' = fromCh \circ reverse^{\dagger} \circ toCh \\
&\ \{\text{-\# INLINE reverse' \#-}\}
\end{aligned}
$$

Dually, the Cochurch encoding takes the input stepper function and simply postcomposes r, i.e. as each subtree is yielded, it is then transformed:

$$
\begin{aligned}
&reverse^{\ddagger} :: Tree^{\ddagger}\ a \to Tree^{\ddagger}\ a \\
&reverse^{\ddagger}\ (Tree^{\ddagger}\ h\ s) = Tree^{\ddagger}\ (r \circ h)\ s \\
&reverse'' :: Tree\ a \to Tree\ a \\
&reverse'' = fromCoCh \circ reverse^{\ddagger} \circ toCoCh \\
&\ \{\text{-\# INLINE reverse'' \#-}\}
\end{aligned}
$$

In a similar way, we can define the function *filter* over *Trees*

$$
\begin{aligned}
&filter :: (a \to Bool) \to Tree\ a \to Tree\ a \\
&filter\ p\ Empty \quad\ = Empty \\
&filter\ p\ (Leaf\ a) \ = \textbf{if}\ p\ a\ \textbf{then}\ Leaf\ a\ \textbf{else}\ Empty \\
&filter\ p\ (Fork\ l\ r) = append\ (filter\ p\ l)\ (filter\ p\ r)
\end{aligned}
$$

which takes a predicate p and discards any elements that do not satisfy the predicate. Unlike a *filter* on linear sequences, discarding an element does not mean we have to recursively search for the next element. Instead, we simply return *Empty*.

Like *reverse*, *filter* is defined with a transformation that is composed with a (co)algebra

$$
\begin{aligned}
&f :: (a \to Bool) \to \underline{Tree}\ a\ c \to \underline{Tree}\ a\ c \\
&f\ p\ \underline{Empty} \quad\ = \underline{Empty}
\end{aligned}
$$

$$
\begin{aligned}
f\ p\ (\underline{Leaf}\ x) &= \mathbf{if}\ p\ x\ \mathbf{then}\ (\underline{Leaf}\ x)\ \mathbf{else}\ \underline{Empty} \\
f\ p\ (\underline{Fork}\ l\ r) &= \underline{Fork}\ l\ r
\end{aligned}
$$

which tests each of the elements using the predicate p, and returns *Empty* if they fail.

The definition for the Church encoding version of *filter* is then similar to that of *reverse*:

$$
\begin{aligned}
&\mathit{filter}^{\dagger} :: (a \to \mathit{Bool}) \to \mathit{Tree}^{\dagger}\ a \to \mathit{Tree}^{\dagger}\ a \\
&\mathit{filter}^{\dagger}\ p\ (\mathit{Tree}^{\dagger}\ g) = \mathit{Tree}^{\dagger}\ (\lambda a \to g\ (a \circ (f\ p))) \\
&\mathit{filter}' :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Tree}\ a \\
&\mathit{filter}'\ p = \mathit{fromCh} \circ \mathit{filter}^{\dagger}\ p \circ \mathit{toCh} \\
&\ \{\text{-\# INLINE filter' \#-}\}
\end{aligned}
$$

and likewise for the Cochurch encoding:

$$
\begin{aligned}
&\mathit{filter}^{\ddagger} :: (a \to \mathit{Bool}) \to \mathit{Tree}^{\ddagger}\ a \to \mathit{Tree}^{\ddagger}\ a \\
&\mathit{filter}^{\ddagger}\ p\ (\mathit{Tree}^{\ddagger}\ h\ s) = \mathit{Tree}^{\ddagger}\ (f\ p \circ h)\ s \\
&\mathit{filter}'' :: (a \to \mathit{Bool}) \to \mathit{Tree}\ a \to \mathit{Tree}\ a \\
&\mathit{filter}''\ p = \mathit{fromCoCh} \circ \mathit{filter}^{\ddagger}\ p \circ \mathit{toCoCh} \\
&\ \{\text{-\# INLINE filter'' \#-}\}
\end{aligned}
$$

At this point, we are able to write fusible producers, consumers, and transformers by defining functions over our concrete representation and converting to and from it when necessary. The goal of this effort is to provide an implementation of an interface that is more efficient and therefore obtains better performance than the analogous functions over the abstract datatype. There may be cases, however, where the analogous function over the abstract datatype is more efficient in certain contexts.

A.6 Rewrite rules, revisited

One of the features of leaf trees is that appending two trees is a constant time, nonrecursive operation:

$$
\begin{aligned}
&\mathit{append} :: \mathit{Tree}\ a \to \mathit{Tree}\ a \to \mathit{Tree}\ a \\
&\mathit{append}\ \mathit{t1}\ \mathit{Empty} = \mathit{t1} \\
&\mathit{append}\ \mathit{Empty}\ \mathit{t2} = \mathit{t2} \\
&\mathit{append}\ \mathit{t1}\ \mathit{t2} \qquad = \mathit{Fork}\ \mathit{t1}\ \mathit{t2}
\end{aligned}
$$

This is extremely efficient as it requires no recursion and no copying. The downside, however, is that, if this is in the middle of a recursive pipeline such as

$$\begin{aligned}&\mathit{sumApp}\,(x, y) = \mathit{sum}\,(\mathit{append}\,(\mathit{enum}\,(x, y))\\&\quad(\mathit{enum}\,(x, y)))\end{aligned}$$

the fusion of the pipeline breaks down because *enum* has to write out an intermediate data structure for use by *append*, and then *sum* consumes the newly appended tree. We can, of course, define fusible versions of *append* using both Church encodings and Cochurch encodings. The Church encoding version applies the abstracted algebra h' to the two subtrees joined by a *Fork*, each of which is also applied to h'.

$$\begin{aligned}&\mathit{append}^{\dagger} :: \mathit{Tree}^{\dagger}\ a \to \mathit{Tree}^{\dagger}\ a \to \mathit{Tree}^{\dagger}\ a\\&\mathit{append}^{\dagger}\,(\mathit{Tree}^{\dagger}\ \mathit{g1})\,(\mathit{Tree}^{\dagger}\ \mathit{g2}) =\\&\quad\mathit{Tree}^{\dagger}\,(\lambda a \to a\,(\mathit{Fork}\,(\mathit{g1}\ a)\,(\mathit{g2}\ a)))\\&\mathit{append}' :: \mathit{Tree}\ a \to \mathit{Tree}\ a \to \mathit{Tree}\ a\\&\mathit{append}'\ \mathit{t1}\ \mathit{t2} = \mathit{fromCh}\,(\mathit{append}^{\dagger}\,(\mathit{toCh}\ \mathit{t1})\,(\mathit{toCh}\ \mathit{t2}))\\&\ \{\text{-\# INLINE append' \#-}\}\end{aligned}$$

The Cochurch encoding version uses a feature we have hitherto been silent about: state. We can think of the seed of a Cochurch encoding as a state, and the stepper function as an iterator. The iterator takes this state as an argument and uses it to perform a computation that produces values each time it is applied and an updated state. In our case, the possibilities are to halt the computation with no value using *Empty*, halt but yield a value using *Leaf*, or to branch into two new states using *Fork*. Our Cochurch encoding does not specify a specific state type, only requiring that it match the argument type of the stepper function. Therefore, we can transform a Cochurch encoded tree by creating a stepper function that applies prior one, then wraps successive states in a new, more expressive state type. This new state type can then be used to determine what do to next. For the $\mathit{append}^{\ddagger}$ function,

$$\begin{aligned}&\mathit{append}^{\ddagger} :: \mathit{Tree}^{\ddagger}\ a \to \mathit{Tree}^{\ddagger}\ a \to \mathit{Tree}^{\ddagger}\ a\\&\mathit{append}^{\ddagger}\,(\mathit{Tree}^{\ddagger}\ h_1\ s_1)\,(\mathit{Tree}^{\ddagger}\ h_2\ s_2) = \mathit{Tree}^{\ddagger}\ h'\ \mathit{Nothing}\\&\quad\textbf{where}\\&\qquad h'\ \mathit{Nothing} = \mathit{Fork}\,(\mathit{Just}\,(\mathit{Tree}^{\ddagger}\ h_1\ s_1))\\&\qquad\qquad\qquad\qquad(\mathit{Just}\,(\mathit{Tree}^{\ddagger}\ h_2\ s_2))\\&\qquad h'\,(\mathit{Just}\,(\mathit{Tree}^{\ddagger}\ h\ s)) = \textbf{case}\ h\ s\ \textbf{of}\\&\qquad\quad\mathit{Empty}\ \to \mathit{Empty}\end{aligned}$$

$$
\begin{aligned}
&\underline{Leaf}\ a \ \ \rightarrow \underline{Leaf}\ a \\
&\underline{Fork}\ l\ r \rightarrow \underline{Fork}\ (Just\ (Tree^{\ddagger}\ h\ l)) \\
&\qquad\qquad\qquad\quad (Just\ (Tree^{\ddagger}\ h\ r))
\end{aligned}
$$

$$
\begin{aligned}
&append'' :: Tree\ a \rightarrow Tree\ a \rightarrow Tree\ a \\
&append''\ t1\ t2 = fromCoCh \\
&\quad (append^{\ddagger}\ (toCoCh\ t1)\ (toCoCh\ t2)) \\
&\ \{\text{-\# INLINE append'' \#-}\}
\end{aligned}
$$

we wrap the original state in a *Maybe* type that signals whether the stepper function is is being called the first time (the *Nothing* case), in which case it should yield its two arguments joined by a *Fork*. The *Nothing* case then wraps each resulting state so that it contains the seed and stepper function of one of the arguments. When the stepper function is applied to each of these states, it can then apply the correct stepper function for branch it is being called on.

For both Church and Cochurch encodings, these functions allow us to fuse *append*s, solving the problem within a pipeline. When unfused, however, these functions are extremely inefficient. Even if we make no changes to a tree, converting it to another representation and back again involves a full traversal of the structure and forces the entire tree to be copied. The original version, on the other hand, did not need to inspect either subtree and could join them without copying. When *append* is part of a pipeline that already does this, it actually improves performance by fusing and avoiding intermediate data structures. Otherwise, it actually creates even more intermediate data structures than the original.

We now have a situation where we would like to use different versions of *append* based on whether or not it appears in a pipeline. We could, of course, provide an interface that allows the programmer to choose between different implementations of *append*. It would be unacceptably arduous, however, to require that the programmer always choose the right one for a given situation, and for large programs this might not even be feasible. Fortunately, the rewrite pragmas that we discussed in Section A.2 allow us define two versions of a function and then leave it to the compiler to choose the appropriate one for us at compile time:

```
{-# RULES
  "append -> fused" [~1] forall t1 t2.
     append t1 t2 =
       fromCh (append^ (toCh t1) (toCh t2))
  "append -> unfused" [1] forall t1 t2.
```

```
fromCh (append† (toCh t1) (toCh t2)) =
   append t1 t2 #-}
```

These rules use the simplifier phase notation that we previously used with the *INLINE* pragma. The symbol can be read as 'before'. The first rule swaps the out the bog-standard definition of *append* for the Church encoded version whenever it is encountered before simplifier phase 1. Once the simplifier reaches this, its second to last phase, it checks to see if there are any instances of $append^{\dagger}$ left that have not either of their conversion combinators fused away. If not, it puts the original *append* back, since it would be more efficient in this situation. We can, of course, implement the same sort of rules for the Cochurch encoding version by swapping the combinators and using $append^{\ddagger}$ instead.

Appendix B

Attribute Grammar Transformations

B.1 ElimDeadCode.ag

```
%%[93 hs module {%{EH}Core.Trf.ElimDeadCode} import ({%{EH}Base.Common},{%{EH}Core},{%{EH}Ty}, qualified Data.Set as Set!
!, Data.Set(Set(..)))
%%]


%%[93 ag import ({Core/AbsSyn})

{
cmodTrfElimDeadCode :: CModule -> CModule
cmodTrfElimDeadCode cmod =
        let t = wrap_CodeAGItf (sem_CodeAGItf (CodeAGItf_AGItf cmod))
                               (Inh_CodeAGItf {})
        in trf_Syn_CodeAGItf t
}

WRAPPER CodeAGItf

ATTR AllCodeNT [ | | varS USE {`Set.union`} {Set.empty} : {Set HsName} ]

SEM CExpr
        | Var lhs.varS = Set.singleton $ acbrefNm @ref
        | Let lhs.varS = if @loc.removable
                         then @body.varS
                         else (@binds.varS `Set.union` @body.varS) `Set.difference` Set.fromList @binds.nms

ATTR CodeAGItf [ | | trf : CModule]
ATTR AllCodeNT [ | | trf : SELF]

SEM CExpr
        | Let loc.removable = not $ Set.fold (\ a b -> a `elem` @binds.nms || b) False @body.varS

SEM CExpr
        | Let lhs.trf = if @loc.removable
                        then @body.trf
                        else @trf

ATTR CBind [ | | nm : {HsName} ]

SEM CBind
        | Bind lhs.nm = @nm

ATTR CBindL [ | | nms : {[HsName]} ]

SEM CBindL
        | Cons lhs.nms = @hd.nm : @tl.nms
        | Nil  lhs.nms = []
%%]
```

B.2 Fusion.cag

```
%%[93 hs module {%{EH}Core.Trf.Fusion} import ({%{EH}Core.Trf.Rewrite} (rewrite, ConvMp(..), Role(..)), {%{EH}Core.Trf.I!
!nline} (inline, FuncMp(..)))
%%]

%%[93 hs import (qualified Data.Set as Set,Data.Maybe (fromMaybe, isJust, isNothing),Debug.Trace,{%{EH}Ty},{%{EH}Base.Co!
!mmon},{%{EH}LamInfo},{%{EH}Core},{%{EH}AbstractCore}, {%{EH}Core.Trf.Subst} (apply),  qualified Data.Map as Map)
%%]
```

```
%%[93 ag import ({Core/AbsSyn})

{
cmodTrfFusion :: LamMp -> CModule -> CModule
cmodTrfFusion lamMp cmod =
     let t = wrap_CodeAGItf (sem_CodeAGItf (CodeAGItf_AGItf cmod))
                            (Inh_CodeAGItf {lamMp_Inh_CodeAGItf = lamMp
                                           } )
     in inlined_Syn_CodeAGItf t
}

-- PRAGMA nocycle

-- Main idea: inlined should inline all fusible functions (and their nonrecursive subterms)
--            rewritten should take the result and do forall x. cons (abs x) |-> x
--            fused should finish the job

-- applied should find fusible functions in a pipeline, inline them, and try to find the conversion functions

ATTR CodeAGItf [ || applied : CModule]
ATTR AllCodeNT [ || applied : SELF]

SEM CExpr
        | App lhs.applied = @applied
        | App loc.applied = if @func.isFusible
                            then trace "Applying..." $ apply @func.applied @arg.applied
                            else acoreApp1 @func.applied @arg.applied
        | Var lhs.applied = if @isFusible
                            then fromMaybe @applied (Map.lookup (acbrefNm @ref) @lhs.fuseMp)
                            else @applied


ATTR CodeAGItf [ | | rewritten : CModule]
ATTR AllCodeNT [ | | rewritten : SELF]

SEM CExpr
       | App lhs.rewritten = @loc.rewritten
       | App loc.rewritten = if @func.isFusible && @arg.isFusible
                             then trace "Rewriting..." $ rewrite @lhs.convMp @loc.applied
                             else acoreApp1 @func.rewritten @arg.rewritten

 --Fuse all the things.  Any fusible function should be fair game, regardless of whether it is applied to a fusible one o!
!r not.  We are no longer looking for rewrite rules, so conversion functions inlined, too, if they remain

ATTR CodeAGItf [ | | inlined : CModule]
ATTR AllCodeNT [ | | inlined : SELF   ]

SEM CExpr
       | App lhs.inlined = if @func.isFusible
                           then trace ("Calling inline with funcMp " ++ (show $ Map.keys @lhs.funcMp)) $ inline @lhs.fun!
!cMp @lhs.convMp @loc.rewritten
                           else acoreApp1 @func.inlined @arg.inlined

 ---------------------------------------------------------------------------
 -- Tests to see if a function is fusible.  A @func of an App is fusible if it
 -- is a named function found in the fuseMp, if it is a function application
 -- whose @func is fusible.
 ---------------------------------------------------------------------------

 ATTR CExpr [ | | isFusible : Bool ]

 SEM CExpr
        | Var loc.isFusible           = isJust (Map.lookup (acbrefNm @ref) @lhs.fuseMp)
        | Var lhs.isFusible           = @isFusible
        | App lhs.isFusible           = @func.isFusible
        | Ann lhs.isFusible           = @expr.isFusible
        | Let lhs.isFusible           = @body.isFusible
        | * - App Var Ann Let lhs.isFusible   = False

 -- Transformations to actually perform a beta reduction. An App node is beta reduced by calling
 -- subst nm body expr, where nm is the variable to replace, body is the body of the function, and
 -- expr is the expr to substitute in for nm that passes in the substExpr attribute.

 ---------------------------------------------------------------------------
 -- Gather the names of conversions and their role into a Map
 ---------------------------------------------------------------------------

 ATTR AllCodeNT [ convMp : ConvMp | | ]

 SEM CodeAGItf
        | AGItf module.convMp = Map.empty

 SEM CExpr
        | Let body.convMp = @binds.gathConvMp `Map.union` @lhs.convMp

 ATTR CBindL [ | | gathConvMp USE {`Map.union`} {Map.empty} : ConvMp ]

 ATTR CBind  [ | | gathConvMp : ConvMp ]

 SEM CBind
        | Bind loc.gathConvMp = fromMaybe Map.empty (do
```

```
                                { lamInfo <- Map.lookup @nm @lhs.lamMp
                                ; role    <- Map.lookup acbaspkeyFusionRole (laminfoBindAspMp lamInfo)
                                ; return (case role of
                                            LamInfoBindAsp_FusionRole FusionRole_BuildLeft  -> Map.singleton @nm Ab!
!s
                                            LamInfoBindAsp_FusionRole FusionRole_BuildRight -> Map.singleton @nm Co!
!n
                                            _                                               -> Map.empty)
                                })

 ------------------------------------------------------------------------
 -- Pass fusible function names and bodies down the expressions
 ------------------------------------------------------------------------

 WRAPPER CodeAGItf

 {
 type FuncS  = Set.Set HsName
 }

 ATTR CodeAGItf AllCodeNT [lamMp : LamMp  | | ] -- passed in, contains which functions have which fusion roles
 ATTR AllCodeNT [fuseMp : FuncMp | | ] -- starts blank, this will contain the transformed code (circular dependecy) for t!
!he functions that need to be inlined for fusion

 SEM CodeAGItf
         | AGItf module.fuseMp = Map.empty


 SEM CExpr
         | Let body.fuseMp  = @binds.gathFuseMp `Map.union` @lhs.fuseMp -- gather the functions from let bindings that ne!
!ed to be fused, and add them to the parent's fusion map, and push this down the tree


 ATTR CBindL [ || gathFuseMp USE {`Map.union`} {Map.empty} : FuncMp ] -- combine all the maps for all the elements of a C!
!Bind list

 ATTR CBind [ || gathFuseMp : FuncMp ]

 SEM CBind
   | Bind loc.gathFuseMp = fromMaybe Map.empty (do
                             { lamInfo <- Map.lookup @nm @lhs.lamMp -- look up a bindings lamInfo
                             ; role    <- Map.lookup acbaspkeyFusionRole (laminfoBindAspMp lamInfo)
                             ; return (case role of
                                         LamInfoBindAsp_FusionRole FusionRole_Fuse -> Map.singleton @nm $ head @bindAspec!
!ts.gathExpr -- add the name + expression
                                         _                                         -> Map.empty)
                             })

 ATTR CBindAspectL [ || gathExpr USE {++} {[]} : {[CExpr]} ] -- this should just be always be a singleton list, but proba!
!bly not the safest way...

 ATTR CBindAspect [ || gathExpr USE {++} {[]} : {[CExpr]}]

 SEM CBindAspect
         | Bind lhs.gathExpr = [@expr.inlined]


 ------------------------------------------------------------------------
 -- Collect the names of fusible transformers (that is, functions that are used
 -- in fusible functions that aren't conversion functions)
 ------------------------------------------------------------------------

 ATTR AllCodeNT [ inFusibleFunc : Bool | | ]

 SEM CodeAGItf
         | AGItf module.inFusibleFunc = False


 SEM CBind
         | Bind bindAspects.inFusibleFunc = @lhs.inFusibleFunc || isJust (Map.lookup @nm @gathFuseMp)


 ATTR AllCodeNT [ funcMp : FuncMp | | ]

 SEM CodeAGItf
         | AGItf module.funcMp = Map.empty
 {-
 ATTR CodeAGItf AllCodeNT [ | | funcS USE {`Set.union`} {Set.empty} : FuncS ]


 SEM CExpr
         | Var lhs.funcS = if @lhs.inFusibleFunc
                           then trace ("Adding function " ++ show (acbrefNm @ref) ++ " to the funcS") $ Set.singleton (ac!
!brefNm @ref)
                           else Set.empty
 -}
 SEM CExpr
         | Let body.funcMp = if @categ == acoreBindcategRec
                             then @lhs.funcMp `Map.difference` @binds.gathFuncMp
                             else @binds.gathFuncMp `Map.union` @lhs.funcMp
         | Let binds.funcMp = @lhs.funcMp
```

```
ATTR CBindL [ | | gathFuncMp USE {`Map.union`} {Map.empty} : FuncMp ]

ATTR CBind [ | | gathFuncMp : FuncMp ]

SEM CBind
        | Bind lhs.gathFuncMp = if not (isNothing (Map.lookup @nm @lhs.convMp))
                                then trace ("Not adding function " ++ show @nm ++ "because it was a conversion function"!
!) $ Map.empty
                                else trace ("Adding function " ++ show @nm ++ "to the funcMp") $ Map.singleton @nm $ hea!
!d @bindAspects.gathExpr

ATTR CExpr [ | | varName : {Maybe HsName} ]

SEM CExpr
        | Var lhs.varName     = Just $ acbrefNm @ref
        | * - Var lhs.varName = Nothing
%%]
```

B.3 Inline.cag

```
%%[93 hs module {%{EH}Core.Trf.Inline} import ({%{EH}Ty},{%{EH}Base.Common},{%{EH}LamInfo},{%{EH}Core},{%{EH}AbstractCor!
!e}, {%{EH}Core.Trf.Subst} (apply), {%{EH}Core.Trf.Rewrite}(ConvMp(..)), qualified Data.Map as Map, Debug.Trace(trace), D!
!ata.Maybe(fromJust,isJust,isNothing), Control.Monad(when), {%{EH}Core.Trf.Subst}(cexprSubst))
%%]

%%[93 ag import ({Core/AbsSyn})

-- CURRENTLY BROKEN FOR TYPECLASSES

{
inline :: FuncMp -> ConvMp -> CExpr -> CExpr
inline funcMp convMp expr =
        let t = wrap_CExpr (sem_CExpr expr)
                           (Inh_CExpr { funcMp_Inh_CExpr = funcMp
                                      , convMp_Inh_CExpr = convMp
                                      }
                           )
        in inlined_Syn_CExpr t

caseOfCase :: CExpr -> CAltL -> CExpr -> CAltL -> CExpr -> CExpr
caseOfCase e outerAlts outerDflt innerAlts innerDflt = acoreCaseDflt e (pushCases outerAlts outerDflt innerAlts) (Just $!
! acoreCaseDflt innerDflt outerAlts (Just outerDflt))
   where
     pushCases :: CAltL -> CExpr -> CAltL -> CAltL
     pushCases outer dflt inner = map (\ (CAlt_Alt p r) -> CAlt_Alt p (acoreCaseDflt r outer (Just dflt))) inner
}

{
caseOfKnown :: CTag -> [CExpr] -> [(CPat,CExpr)] -> Maybe CExpr
caseOfKnown ctag cargs calts = do (binds,expr) <- (case match ctag calts of {Nothing -> trace "CoK: no match found" Noth!
!ing; x -> x})
                                  trace ("CoK: Found match with binds " ++ show binds ++ " and expr " ++ show expr) $ re!
!turn ()
                                  let substs = zip cargs binds
                                  trace ("CoK: Making substs " ++ show substs) $ return ()
                                  let expr' = foldr (\ (carg,bind) expr -> cexprSubst bind expr carg) expr substs
                                  trace ("CoK: result is " ++ show expr') $ return expr'

-- Only matches identical CTags, not vars that would work or the default, which would be preferred.

match :: CTag -> [(CPat,CExpr)] -> Maybe ([HsName],CExpr)
match ctag x@((CPat_Con ptag _ binds, expr):xs)
        | ctag == ptag = Just (map getfldNm binds,expr) -- vars to be subst and the expr
        | otherwise    = match ctag xs
match ctag (_:xs) = match ctag xs
match ctag []     = Nothing

getfldNm :: CPatFld -> HsName
getfldNm (CPatFld_Fld _ _ fldNm _) = fldNm
}

{

type FuncMp = Map.Map HsName CExpr

}

WRAPPER CExpr

ATTR AllCodeNT [ funcMp : FuncMp convMp : ConvMp | | inlined : SELF ]

SEM CExpr
        | Let body.funcMp = if @categ == acoreBindcategRec
                            then @lhs.funcMp `Map.difference` @binds.gathFuncMp
                            else @binds.gathFuncMp `Map.union` @lhs.funcMp
        | Let binds.funcMp = @lhs.funcMp
```

```
ATTR CBindL [ | | gathFuncMp USE {`Map.union`} {Map.empty} : FuncMp ]

ATTR CBind [ | | gathFuncMp : FuncMp ]

ATTR CBindAspectL [ || gathExpr USE {++} {[]} : {[CExpr]} ] -- this should just be always be a singleton list, but proba!
!bly not the safest way...

ATTR CBindAspect [ || gathExpr USE {++} {[]} : {[CExpr]}]

SEM CBindAspect
        | Bind lhs.gathExpr = [@expr.inlined]

SEM CBind
        | Bind lhs.gathFuncMp = if not (isNothing (Map.lookup @nm @lhs.convMp))
                                then {-trace ("Not adding function " ++ show @nm ++ "because it was a conversion functio!
!n") $-} Map.empty
                                else {-trace ("Adding function " ++ show @nm ++ "to the funcMp") $-} Map.singleton @nm $!
! head @bindAspects.gathExpr


SEM CExpr
        | App lhs.inlined = if @func.isInlinable
                            then {- trace ("Inlining " ++ show @func.inlined) $ -} inline @lhs.funcMp @lhs.convMp $ (app!
!ly @func.inlined @arg.inlined) -- application may have triggered new opportunies, hence the recursive call.
                            else acoreApp1 @func.inlined @arg.inlined
        | Var lhs.inlined = if @isInlinable
                            then fromJust (Map.lookup (acbrefNm @ref) @lhs.funcMp)
                            else @inlined
        | Case lhs.inlined = maybe (acoreCaseDflt @expr.inlined @alts.inlined (Just @dflt.inlined)) (inline @lhs.funcMp !
!@lhs.convMp) (orElse @loc.coc @loc.cok)

SEM CExpr
        | Case loc.cok = trace "Looking to apply CoK" $
                         (do { scrutineeExpr <- @loc.scrutineeExpr
                             ; trace ("CoK: Found scrutinee: " ++ show scrutineeExpr) $ return ()
                             ; constExpr   <- return scrutineeExpr
                             ; trace ("CoK: Found constructor expression: " ++ show constExpr) $ return ()
                             ; ctag        <- getConstructor constExpr
                             ; trace ("CoK: Found constructor: " ++ show constExpr) $ return ()
                             ; let cargs   = reverse (getArgs constExpr)
                             ; let alts    = @alts.altPairs
                             ; cok <- trace "CoK: Calling CoK" $ caseOfKnown ctag cargs alts
                             ; let cok' = foldr (\ (cat,bind) e -> acoreLetBase cat bind e) cok @loc.scrutineeLets
                             ; return cok'
                             })
        | Case loc.coc = trace "Looking to apply CoC" $
                         do { scrutineeExpr         <- @loc.scrutineeExpr -- the thing being scrutinised
                            ; e                     <- getCaseExpr scrutineeExpr  -- is it another case? If so, return!
! the scrutinee
                            ; (innerAlts,innerDflt) <- getCaseAlts scrutineeExpr -- then get the alts and the default
                            ; let coc = foldr (\ (categ,binds) expr -> acoreLetBase categ binds expr) (trace "Case sta!
!tement found in scrutinee, calling CoC" $ caseOfCase e @alts.inlined @dflt.inlined innerAlts innerDflt) @loc.scrutineeLe!
!ts
                            ; trace ("CoC result: " ++ show coc) $return coc
                            }
SEM CExpr
       | Ann lhs.inlined = trace "Removing an annotation" $ @expr.inlined

SEM CExpr
       | Case loc.scrutinee = orElse (do { scrutineeNm <- @expr.varName
                                         ; case Map.lookup scrutineeNm @lhs.funcMp of { Nothing -> trace "No case scruti!
!nee found in funcMp" Nothing; Just x -> trace ("Found scrutinee " ++ show x) (Just x) } } ) (Just @expr.inlined)
       | Case loc.scrutineeLets = maybe [] getLets @loc.scrutinee
       | Case loc.scrutineeExpr = @loc.scrutinee >>= (return . getLetExpr)

{
orElse :: Maybe a -> Maybe a -> Maybe a
orElse x y = case x of
               Just _  -> x
               Nothing -> y

getLets :: CExpr -> [(CBindCateg,CBindL)]
getLets (CExpr_Let categ binds expr) = (categ, binds) : getLets expr
getLets _                            = []

getLetExpr :: CExpr -> CExpr
getLetExpr (CExpr_Let _ _ expr) = getLetExpr expr
getLetExpr expr                 = expr

isCase :: CExpr -> Bool
isCase (CExpr_Case _ _ _)  = True
isCase (CExpr_Let _ _ expr) = isCase expr

getCaseExpr :: CExpr -> Maybe CExpr
getCaseExpr (CExpr_Case expr _ _) = Just expr
getCaseExpr _                     = Nothing

getLetCaseExpr (CExpr_Let _ _ expr)  = getLetCaseExpr expr
getLetCaseExpr (CExpr_Case expr _ _) = expr

getCaseAlts :: CExpr -> Maybe (CAltL, CExpr)
```

```
getCaseAlts (CExpr_Case _ alts dflt) = Just (alts,dflt)
getCaseAlts _                        = Nothing

-- getConstructorExpr :: CExpr -> Maybe CExpr
-- getConstructorExpr (CExpr_Ann _ expr) = getConstructorExpr expr
-- getConstructorExpr expr                  = Just expr


getConstructor :: CExpr -> Maybe CTag
getConstructor (CExpr_App func arg)       = getConstructor func
getConstructor (CExpr_Tup tag)            = Just tag
getConstructor _                          = Nothing

getArgs :: CExpr -> [CExpr]
getArgs (CExpr_App func arg) = arg : getArgs func
getArgs _                    = []
}


ATTR AllCodeNT [ | | self : SELF ]

ATTR CAlt [ | | altPair : {(CPat,CExpr)} ]

SEM CAlt
        | Alt lhs.altPair = (@pat.self, @expr.inlined)


ATTR CAltL [ | | altPairs : {[(CPat, CExpr)]} ]

SEM CAltL
        | Cons lhs.altPairs = @hd.altPair : @tl.altPairs
        | Nil  lhs.altPairs = []

ATTR CExpr [ | | isInlinable : Bool ]

SEM CExpr
        | Var loc.isInlinable = isJust (Map.lookup (acbrefNm @ref) @lhs.funcMp)
        | Var lhs.isInlinable = @isInlinable
        | App lhs.isInlinable = @func.isInlinable
        | Lam lhs.isInlinable = True
        | * - App Var Lam lhs.isInlinable = False

ATTR CExpr [ | | varName : {Maybe HsName} ]

SEM CExpr
        | Var loc.varName     = Just $ acbrefNm @ref
        | Var lhs.varName     = @varName
        | * - Var lhs.varName = Nothing

%%]
```

B.4 LetCaseScrutinee.ag

```
%%[93 hs module {%{EH}Core.Trf.LetCaseScrutinee} import ({%{EH}Base.Common}, {%{EH}Ty}) export (cmodTrfLetCaseScrutinee)
%%]

%%[93 hs import (Debug.Trace, Data.Set(Set(..)), qualified Data.Set as Set, Data.Maybe(fromMaybe))
%%]

%%[93 hs import({%{EH}AbstractCore},{%{EH}Core})
%%]

%%[93 ag import ({Core/AbsSyn})
{
cmodTrfLetCaseScrutinee :: HsName -> UID -> CModule -> CModule
cmodTrfLetCaseScrutinee modNm uniq cmod
  = let t = wrap_CodeAGItf (sem_CodeAGItf (CodeAGItf_AGItf cmod))
                           (Inh_CodeAGItf { gUniq_Inh_CodeAGItf = uniq
                                          , modNm_Inh_CodeAGItf = modNm
                                          })
    in cTrf_Syn_CodeAGItf t
}


WRAPPER CodeAGItf

ATTR AllCodeNT CodeAGItf [ modNm: HsName | | ]

ATTR AllCodeNT [ | gUniq: UID | ]
ATTR CodeAGItf [ gUniq: UID | | ]

SEM CExpr
  | App         (func.gUniq,loc.lUniq)      =   mkNewUID @lhs.gUniq
  | Lam         (body.gUniq,loc.lUniq,loc.lUniq2)
                                            =   mkNewLevUID2 @lhs.gUniq
  | Let         (binds.gUniq,loc.lUniq)     =   mkNewUID @lhs.gUniq
  | Case TupDel TupIns TupUpd
```

```
              (expr.gUniq,loc.lUniq)      =   mkNewUID @lhs.gUniq

ATTR AllCodeNT [ sBindS : {Set HsName} | | ]

SEM CodeAGItf
  | AGItf module.sBindS = Set.empty

SEM CExpr
  | Let body.sBindS = if @categ == acoreBindcategStrict
                      then foldr Set.insert @lhs.sBindS @binds.nmL
                      else @lhs.sBindS

ATTR CBindL [ | | nmL : {[HsName]} ]

SEM CBindL
  | Cons lhs.nmL = @hd.nm : @tl.nmL
  | Nil lhs.nmL  = []

ATTR CBind [ | | nm : HsName ]

SEM CBind
  | Bind lhs.nm = @nm

ATTR AllCodeNT [ | | cTrf : SELF ]
ATTR CodeAGItf [ | | cTrf : CModule]

SEM CExpr
  | Case lhs.cTrf = fromMaybe (trace ("Found non-strictly bound scrutinee" ++ show @expr.cTrf) $ acoreLet acoreBindcateg!
!Strict [(acoreBind1 @trNm @expr.cTrf)] (acoreCaseDflt (acoreVar @trNm) @alts.cTrf (Just @dflt.cTrf)))
                      (do { nm <- @expr.varName
                          ; if Set.member nm @lhs.sBindS
                            then return $ acoreCaseDflt @expr.cTrf @alts.cTrf (Just @dflt.cTrf)
                            else if (show @expr.cTrf) == "UHC.Base.undefined" then return @expr.cTrf else Nothing })

SEM CExpr
  | Case loc.trNm = hsnQualUniqify @lhs.modNm $ uidHNm @lUniq

ATTR CExpr [ | | varName : {Maybe HsName} ]

SEM CExpr
  | Var lhs.varName     = Just $ acbrefNm @ref
  | * - Var lhs.varName = Nothing
%%]
```

B.5 Rewrite.cag

```
%%[93 hs module {%{EH}Core.Trf.Rewrite} import (Data.Maybe(fromMaybe), Debug.Trace(trace),qualified Data.Map as Map, {%{!
!EH}AbstractCore},{%{EH}Core},{%{EH}Ty},{%{EH}Base.Common})
%%]

%%[93 ag import ({Core/AbsSyn})

{
rewrite :: ConvMp -> CExpr -> CExpr
rewrite convMp expr =
        let t = wrap_CExpr (sem_CExpr expr)
                           (Inh_CExpr { convMp_Inh_CExpr = convMp
                                      , leftApp_Inh_CExpr = Nothing
                                      } )
        in rewritten_Syn_CExpr t
}

{

data Role   = Abs | Con

isCon Con = True
isCon _   = False

isAbs Abs = True
isAbs _   = False

type ConvMp = Map.Map HsName Role

}

WRAPPER CExpr

ATTR CodeAGItf [ convMp : ConvMp | | ]
ATTR AllCodeNT [ convMp : ConvMp | | rewritten : SELF]

SEM CExpr
        | App lhs.rewritten = if @loc.isRewritable || @arg.isRewritten
                              then @arg.rewritten
                              else acoreApp1 @func.rewritten @arg.rewritten

ATTR CExpr [ | | isRewritten : Bool isRewritable : Bool]
```

```
SEM CExpr
        | * lhs.isRewritten    = @isRewritable
        | App loc.isRewritable = fromMaybe False (do leftNm    <- @lhs.leftApp
                                                     leftRole  <- Map.lookup leftNm @lhs.convMp
                                                     rightNm   <- @func.varName
                                                     rightRole <- Map.lookup rightNm @lhs.convMp
                                                     return (isCon leftRole && isAbs rightRole))
        | Let loc.isRewritable = @body.isRewritable
        | Ann loc.isRewritable = @expr.isRewritable
        | * - App Let Ann loc.isRewritable = False

ATTR CExpr [ | | varName : {Maybe HsName} ]

SEM CExpr
        | Var lhs.varName     = Just $ acbrefNm @ref
        | * - Var lhs.varName = Nothing



ATTR CExpr [ leftApp : {Maybe HsName} | | ]

SEM CExpr
        | App func.leftApp = Nothing
        | App arg.leftApp  = @func.varName

SEM CAlt
        | Alt expr.leftApp = Nothing

SEM CBindAspect
        | Bind expr.leftApp = Nothing
        | FFE  expr.leftApp = Nothing
        | Val  expr.leftApp = Nothing

SEM CModule
        | Mod expr.leftApp = Nothing

SEM CPatFld
        | Fld offset.leftApp = Nothing

SEM MbCExpr
        | Just just.leftApp = Nothing
%%]
```

B.6 Subst.ag

```
%%[93 hs module {%{EH}Core.Trf.Subst} import ({%{EH}AbstractCore},{%{EH}Core},{%{EH}Ty},{%{EH}Base.Common}, Debug.Trace(!
!trace))
%%]

%%[93 ag import ({Core/AbsSyn})

{

apply :: CExpr -> CExpr -> CExpr
apply (CExpr_Lam nm body) arg = cexprSubst nm body arg -- perform a beta reduction
apply func                arg = trace ("tried to inline a non-Lambda expression " ++ show func) $ acoreApp1 func arg -- !
!cannot perform a beta reduction

cexprSubst :: HsName -> CExpr -> CExpr -> CExpr
cexprSubst nm body arg =
        let t = wrap_CExpr (sem_CExpr body)
                           (Inh_CExpr { substExpr_Inh_CExpr = arg
                                      , substName_Inh_CExpr = nm
                                      } )
        in substd_Syn_CExpr t

}

WRAPPER CExpr

ATTR AllCodeNT [ substExpr : CExpr substName : {HsName} | | substd : SELF  ]

SEM CExpr
        | Var lhs.substd = if (acbrefNm @ref) == @lhs.substName
                           then @lhs.substExpr
                           else @substd
%%]
```

Appendix C

UHC Benchmark Source Code

C.1 Unfused benchmarks

```
{-# LANGUAGE ExistentialQuantification, Rank2Types #-}

module BenchList where

import System.IO (hFlush, stdout)
import System.CPUTime (getCPUTime)
import Data.List (foldl')
import qualified Data.List as L

data Result = T | B

data F a = forall b. F (a -> b) | forall b. Flist (a -> [b])

class Forceable a where
    force :: a -> IO Result
    force v = v `seq` return T

seqList = L.foldl' (flip seq) (return ())

instance Forceable [a] where
    force = L.foldl' (flip seq) (return T)

instance Forceable Char
instance Forceable Bool
instance Forceable Int


run :: Int -> a -> [(String,[F a])] -> IO ()
run c x tests = sequence_ $ zipWith (runTest c x) [1..] tests

runTest :: Int -> a -> Int -> (String,[F a]) -> IO ()
runTest count x n (name,tests) = do
  let s = show n
  putStr ((if length s <= 1 then '0' : s else s) ++ "\t")
  fn tests
  putStr ("\t#" ++ (show name) ++ "\n")
  hFlush stdout
        where fn xs = case xs of
                        [f,g,h] -> runN count f x >> putStr "\t"
                                >> runN count g x >> putStr "\t"
                                >> runN count h x >> putStr "\t"
                        [f,g]   -> runN count f x >> putStr "\t"
                                >> runN count g x >> putStr "\t"
                        [f]     -> runN count f x -- >> putStr "\t"
                        _       -> return ()
              run f x = time f x
              runN 0 f x = return ()
              runN c f x = run f x >> runN (c-1) f x

time (Flist f) a = do
  start <- getCPUTime
  v     <- seqList (f a)
  end   <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12 :: Double
  putStr (takeWhile (/= '.') (show diff) ++ (take 4 (dropWhile (/= '.') (show diff))) ++ "\t")
  hFlush stdout

time (F f) a = do
  start <- getCPUTime
  v <- let x = (f a) in seq x (return x)
  end <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12 :: Double
```

```
  putStr (takeWhile (/= '.') (show diff) ++ take 3 (dropWhile (/= '.') (show diff)))
  hFlush stdout

mapMap :: [Int] -> [Int]
mapMap xs = map (subtract 1) (map (+1) xs)

mapFilter :: [Int] -> [Int]
mapFilter xs = map (+1) (filter odd xs)

filterMap :: [Int] -> [Int]
filterMap xs = filter odd (map (+1) xs)

filterFilter :: [Int] -> [Int]
filterFilter xs = filter odd (filter (\ x -> x `mod` 3 /= 0) xs)

main :: IO ()
main = do let testL = enumFromTo 1 5000000 :: [Int]
          seqList testL
          let smallTestL = L.take 1500000 testL :: [Int]
          seqList smallTestL
          run 5 testL trans_tests
          putStr "\n"
          run 5 (1 :: Int,5000000 :: Int) prod_tests
          putStr "\n"
          run 5 smallTestL con_tests
          putStr "\n"
          run 5 (1 :: Int,1500000 :: Int) con_prod_tests

trans_tests :: [(String, [F [Int]])]
trans_tests = [ ("map . map"                      , [Flist mapMap] )
              , ("map . filter"                   , [Flist mapFilter])
              , ("filter . filter"                , [Flist filterFilter] )
              , ("filter . map"                   , [Flist filterMap])
              ]

mapEnumFromTo :: (Int,Int) -> [Int]
mapEnumFromTo (x,y) = map (+1) (enumFromTo x y)

filterEnumFromTo :: (Int, Int) -> [Int]
filterEnumFromTo (x,y) = filter odd (enumFromTo x y)

mapFilterEnumFromTo :: (Int, Int) ->  [Int]
mapFilterEnumFromTo (x,y) = map (+1) (filter odd (enumFromTo x y))

filterMapEnumFromTo :: (Int, Int) ->  [Int]
filterMapEnumFromTo (x,y) = filter odd (map (+1) (enumFromTo x y))

prod_tests = [ ("map . enumFromTo"                , [Flist mapEnumFromTo])
             , ("filter . enumFromTo"             , [Flist filterEnumFromTo])
             , ("map . filter . enumFromTo"       , [Flist mapFilterEnumFromTo])
             , ("filter . map . enumFromTo"       , [Flist filterMapEnumFromTo])
             ]

foldrMap :: [Int] -> Int
foldrMap xs = foldr (+) 0 (map (+1) xs)

foldrFilter :: [Int] -> Int
foldrFilter xs = foldr (+) 0 (filter odd xs)

foldrMapFilter :: [Int] -> Int
foldrMapFilter xs = foldr (+) 0 (map (+1) (filter odd xs))

foldrFilterMap :: [Int] -> Int
foldrFilterMap xs = foldr (+) 0 (filter odd (map (+1) xs))

foldlMap :: [Int] -> Int
foldlMap xs = foldl' (+) 0 (map (+1) xs)

foldlFilter :: [Int] -> Int
foldlFilter xs = foldl' (+) 0 (filter odd xs)

foldlMapFilter :: [Int] -> Int
foldlMapFilter xs = foldl' (+) 0 (map (+1) (filter odd xs))

foldlFilterMap :: [Int] -> Int
foldlFilterMap xs = foldl' (+) 0 (filter odd (map (+1) xs))


con_tests = [ ("foldr . map"                    , [F foldrMap] )
            , ("foldr . filter"                 , [F foldrFilter] )
            , ("foldr . map . filter"           , [F foldrMapFilter] )
            , ("foldr . filter . map"           , [F foldrFilterMap] )
            , ("foldl . map"                    , [F foldlMap] )
            , ("foldl . filter"                 , [F foldlFilter] )
            , ("foldl . map . filter"           , [F foldlMapFilter] )
            , ("foldl . filter . map"           , [F foldlFilterMap] )
            ]

foldrEnumFromTo :: (Int, Int) ->  Int
foldrEnumFromTo (x,y) = foldr (+) 0 (enumFromTo x y)

foldrMapEnumFromTo :: (Int, Int) ->  Int
```

```
foldrMapEnumFromTo (x,y) = foldr (+) 0 (map (+1) (enumFromTo x y))

foldrFilterEnumFromTo :: (Int, Int) ->  Int
foldrFilterEnumFromTo (x,y) = foldr (+) 0 (filter odd (enumFromTo x y))

foldrMapFilterEnumFromTo :: (Int, Int) ->  Int
foldrMapFilterEnumFromTo (x,y) = foldr (+) 0 (map (+1) (filter odd (enumFromTo x y)))

foldrFilterMapEnumFromTo :: (Int, Int) ->  Int
foldrFilterMapEnumFromTo (x,y) = foldr (+) 0 (filter odd (map (+1) (enumFromTo x y)))

foldlEnumFromTo :: (Int, Int) ->  Int
foldlEnumFromTo (x,y) = foldl' (+) 0 (enumFromTo x y)

foldlMapEnumFromTo :: (Int, Int) ->  Int
foldlMapEnumFromTo (x,y) = foldl' (+) 0 (map (+1) (enumFromTo x y))

foldlFilterEnumFromTo :: (Int, Int) ->  Int
foldlFilterEnumFromTo (x,y) = foldl' (+) 0 (filter odd (enumFromTo x y))

foldlMapfilterEnumFromTo :: (Int, Int) ->  Int
foldlMapFilterEnumFromTo (x,y) = foldl' (+) 0 (map (+1) (filter odd (enumFromTo x y)))

foldlFilterMapEnumFromTo :: (Int, Int) ->  Int
foldlFilterMapEnumFromTo (x,y) = foldl' (+) 0 (filter odd (map (+1) (enumFromTo x y)))

con_prod_tests = [ ("foldr . enumFromTo"               , [F foldrEnumFromTo] )
                 , ("foldr . map . enumFromTo"         , [F foldrMapEnumFromTo])
                 , ("foldr . filter . enumFromTo"      , [F foldrFilterEnumFromTo])
                 , ("foldr . map . filter . enumFromTo", [F foldrMapFilterEnumFromTo])
                 , ("foldr . filter . map . enumFromTo", [F foldrFilterMapEnumFromTo] )
                 , ("foldl . enumFromTo"               , [F foldlEnumFromTo])
                 , ("foldl . map . enumFromTo"         , [F foldlMapEnumFromTo])
                 , ("foldl . filter . enumFromTo"      , [F foldlFilterEnumFromTo])
                 , ("foldl . map . filter . enumFromTo", [F foldlMapFilterEnumFromTo])
                 , ("foldl . filter . map . enumFromTo", [F foldlFilterMapEnumFromTo])
                ]
```

C.2 foldr/build benchmarks

```
{-# LANGUAGE ExistentialQuantification, Rank2Types #-}

module Bench where

import Prelude hiding (map,filter,foldl,foldr,enumFromTo)
import qualified Prelude as P
import System.IO (hFlush, stdout)
import System.CPUTime (getCPUTime)
import qualified Data.List as L


data Result = T | B

data F a = forall b. F (a -> b) | forall b. Flist (a -> [b])

class Forceable a where
    force :: a -> IO Result
    force v = v `seq` return T

seqList = L.foldl' (flip seq) (return ())

instance Forceable [a] where
    force = L.foldl' (flip seq) (return T)

instance Forceable Char
instance Forceable Bool
instance Forceable Int
instance Forceable (Fold a)

run :: Int -> a -> [(String,[F a])] -> IO ()
run c x tests = sequence_ $ zipWith (runTest c x) [1..] tests

runTest :: Int -> a -> Int -> (String,[F a]) -> IO ()
runTest count x n (name,tests) = do
  let s = show n
  putStr ((if length s <= 1 then '0' : s else s) ++ "\t")
  fn tests
  putStr ("\t#" ++ (show name) ++ "\n")
  hFlush stdout
         where fn xs = case xs of
                         [f,g,h] -> runN count f x >> putStr "\t"
                                  >> runN count g x >> putStr "\t"
                                  >> runN count h x >> putStr "\t"
                         [f,g]   -> runN count f x >> putStr "\t"
                                 >> runN count g x >> putStr "\t"
                         [f]     -> runN count f x -- >> putStr "\t"
                         _       -> return ()
               run f x = time f x
               runN 0 f x = return ()
               runN c f x = run f x >> runN (c-1) f x


time (Flist f) a = do
  start <- getCPUTime
  v     <- seqList (f a)
  end   <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12 :: Double
  putStr (takeWhile (/= '.') (show diff) ++ (take 4 (dropWhile (/= '.') (show diff))) ++ "\t")
  hFlush stdout

time (F f) a = do
  start <- getCPUTime
  v <- let x = (f a) in seq x (return x)
  end <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12 :: Double
  putStr (takeWhile (/= '.') (show diff) ++ take 4 (dropWhile (/= '.') (show diff)) ++ "\t")
  hFlush stdout

data Stream a = forall s. Stream (s -> Step a s) s

data Step a s = Done
              | Yield a s
              | Skip s

unstream :: Stream a -> [a]
unstream (Stream next s) = unfold s
  where
    unfold s = case next s of
                 Done       -> []
                 Skip s'    -> unfold s'
                 Yield x s' -> x : unfold s'

stream :: [a] -> Stream a
stream lst = Stream next lst
  where
    next []     = Done
    next (x:xs) = Yield x xs

mapS :: (a -> b) -> Stream a -> Stream b
```

```
mapS f (Stream next s) = Stream next' s
  where
    next' s = case next s of
                Done       -> Done
                Skip s'    -> Skip s'
                Yield x s' -> Yield (f x) s'

filterS :: (a -> Bool) -> Stream a -> Stream a
filterS p (Stream next s) = Stream next' s
  where
    next' s = case next s of
                Done       -> Done
                Skip s'    -> Skip s'
                Yield x s' -> if p x then Yield x s' else Skip s'

foldlS :: (b -> a -> b) -> b -> Stream a -> b
foldlS f z (Stream next s) = foldlS_loop z s
  where
    foldlS_loop z s = case next s of
                      Done       -> z
                      Skip s'    -> foldlS_loop z s'
                      Yield x s' -> foldlS_loop (f z x) s'

foldrS :: (a -> b -> b) -> b -> Stream a -> b
foldrS f z (Stream next s) = foldrS_loop s
  where
    foldrS_loop s = case next s of
                      Done       -> z
                      Skip s'    -> foldrS_loop s'
                      Yield x s' -> f x (foldrS_loop s')

enumFromToS :: Int -> Int -> Stream Int
enumFromToS m n = Stream next m
  where
    next x = if x > n then Done else Yield x (x+1)

data Fold a = Fold (forall b. (List a b -> b) -> b)

data List a b = Cons a b | Nil

fold :: [a] -> Fold a
fold lst = Fold (\ a -> let fold_loop [] = a Nil; fold_loop (x:xs) = a (Cons x (fold_loop xs))
                        in fold_loop lst)

build :: Fold a -> [a]
build (Fold h) = h inn
  where
    inn (Cons a b) = a : b
    inn Nil        = []

mapF :: (a -> b) -> Fold a -> Fold b
mapF f (Fold h) = Fold (\ a -> h (hmap a))
  where
    hmap a' (Cons a b) = a' (Cons (f a) b)
    hmap a' Nil        = a' Nil

map' :: (a -> b) -> [a] -> [b]
map' f xs = build (mapF f (fold xs))

filterF :: (a -> Bool) -> Fold a -> Fold a
filterF p (Fold h) = Fold (\ a -> h (hfilter a))
  where
    hfilter a' (Cons a b) = if p a then a' (Cons a b) else b
    hfilter a' Nil        = a' Nil

filter' :: (a -> Bool) -> [a] -> [a]
filter' p xs = build (filterF p (fold xs))

foldlF :: (b -> a -> b) -> b -> Fold a -> b
foldlF f z (Fold h) = h hfoldl z
  where
    hfoldl (Cons b g) = (\ a -> g (f a b))
    hfoldl Nil        = id

foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f z xs = (foldlF f z (fold xs))

foldrF :: (a -> b -> b) -> b -> Fold a -> b
foldrF f z (Fold h) = h hfoldr
  where
    hfoldr (Cons a b) = f a b
    hfoldr Nil        = z

foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' f z xs = (foldrF f z (fold xs))

enumFromToF :: Int -> Int -> Fold Int
enumFromToF m n = Fold (\ a -> let enumFromToLoop x = if x > n then a Nil else a (Cons x (enumFromToLoop (x+1)))
                               in enumFromToLoop m)

enumFromTo' :: Int -> Int -> [Int]
enumFromTo' m n = build (enumFromToF m n)
```

```
mapMapF :: [Int] -> [Int]
mapMapF xs = map' (subtract 1) (map' (+1) xs)

mapFilterF :: [Int] -> [Int]
mapFilterF xs = map' (+1) (filter' odd xs)

filterMapF :: [Int] -> [Int]
filterMapF xs = filter' odd (map' (+1) xs)

filterFilterF :: [Int] -> [Int]
filterFilterF xs = filter' odd (filter' (\ x -> x `mod` 3 /= 0) xs)

main :: IO ()
main = do let testL = P.enumFromTo 1 5000000 :: [Int]
          seqList testL
          let smallTestL = L.take 1500000 testL
          seqList smallTestL
          run 5 testL trans_tests
          putStr "\n"
          run 5 (1 :: Int,5000000 :: Int) prod_tests
          putStr "\n"
          run 5 smallTestL con_tests
          putStr "\n"
          run 5 (1 :: Int,1500000 :: Int) con_prod_tests

trans_tests :: [(String, [F [Int]])]
trans_tests = [ ("map . map"                        , [Flist mapMapF] )
              , ("map . filter"                     , [Flist mapFilterF])
              , ("filter . filter"                  , [Flist filterFilterF] )
              , ("filter . map"                     , [Flist filterMapF])
              ]

mapEnumFromToF (x,y) = map' (+1) (enumFromTo' x y)

filterEnumFromToF (x,y) = filter' odd (enumFromTo' x y)

mapFilterEnumFromToF (x,y) = map' (+1) (filter' odd (enumFromTo' x y))

filterMapEnumFromToF (x,y) = filter' odd (map' (+1) (enumFromTo' x y))

prod_tests = [ ("map . enumFromTo"                , [Flist mapEnumFromToF])
             , ("filter . enumFromTo"             , [Flist filterEnumFromToF])
             , ("map . filter . enumFromTo"       , [Flist mapFilterEnumFromToF])
             , ("filter . map . enumFromTo"       , [Flist filterMapEnumFromToF])
             ]

foldrMapF xs = foldr' (+) 0 (map' (+1) xs)

foldrFilterF xs = foldr' (+) 0 (filter' odd xs)

foldrMapFilterF xs = foldr' (+) 0 (map' (+1) (filter' odd xs))

foldrFilterMapF xs = foldr' (+) 0 (filter' odd (map' (+1) xs))

foldlMapF xs = foldl' (+) 0 (map' (+1) xs)

foldlFilterF xs = foldl' (+) 0 (filter' odd xs)

foldlMapFilterF xs = foldl' (+) 0 (map' (+1) (filter' odd xs))

foldlFilterMapF xs = foldl' (+) 0 (filter' odd (map' (+1) xs))

con_tests = [ ("foldr . map"                      , [F foldrMapF] )
            , ("foldr . filter"                   , [F foldrFilterF] )
            , ("foldr . map . filter"             , [F foldrMapFilterF] )
            , ("foldr . filter . map"             , [F foldrFilterMapF] )
            , ("foldl . map"                      , [F foldlMapF] )
            , ("foldl . filter"                   , [F foldlFilterF] )
            , ("foldl . map . filter"             , [F foldlMapFilterF] )
            , ("foldl . filter . map"             , [F foldlFilterMapF] )
            ]

foldrEnumFromToF (x,y) = foldr' (+) 0 (enumFromTo' x y)
foldrMapEnumFromToF (x,y) = foldr' (+) 0 (map' (+1) (enumFromTo' x y))
foldrFilterEnumFromToF (x,y) = foldr' (+) 0 (filter' odd (enumFromTo' x y))
foldrMapFilterEnumFromToF (x,y) = foldr' (+) 0 (map' (+1) (filter' odd (enumFromTo' x y)))
foldrFilterMapEnumFromToF (x,y) = foldr' (+) 0 (filter' odd (map' (+1) (enumFromTo' x y)))

foldlEnumFromToF (x,y) = foldl' (+) 0 (enumFromTo' x y)
foldlMapEnumFromToF (x,y) = foldl' (+) 0 (map' (+1) (enumFromTo' x y))
foldlFilterEnumFromToF (x,y) = foldl' (+) 0 (filter' odd (enumFromTo' x y))
foldlMapFilterEnumFromToF (x,y) = foldl' (+) 0 (map' (+1) (filter' odd (enumFromTo' x y)))
foldlFilterMapEnumFromToF (x,y) = foldl' (+) 0 (filter' odd (map' (+1) (enumFromTo' x y)))

con_prod_tests = [ ("foldr . enumFromTo"               , [F foldrEnumFromToF] )
                 , ("foldr . map . enumFromTo"         , [F foldrMapEnumFromToF])
                 , ("foldr . filter . enumFromTo"      , [F foldrFilterEnumFromToF])
                 , ("foldr . map . filter . enumFromTo", [F foldrMapFilterEnumFromToF])
                 , ("foldr . filter . map . enumFromTo", [F foldrFilterMapEnumFromToF] )
                 , ("foldl . enumFromTo"               , [F foldlEnumFromToF])
                 , ("foldl . map . enumFromTo"         , [F foldlMapEnumFromToF])
```

```
, ("foldl . filter . enumFromTo"      , [F foldlFilterEnumFromToF])
, ("foldl . map . filter . enumFromTo", [F foldlMapFilterEnumFromToF])
, ("foldl . filter . map . enumFromTo", [F foldlFilterMapEnumFromToF])
]
```

C.3 stream fusion benchmarks

```
{-# LANGUAGE ExistentialQuantification, Rank2Types #-}

module Bench where

import Prelude hiding (map,filter,foldl,foldr,enumFromTo)
import qualified Prelude as P
import System.IO (hFlush, stdout)
import System.CPUTime (getCPUTime)
import qualified Data.List as L

data Result = T | B

data F a = forall b. F (a -> b) | forall b. Flist (a -> [b])

class Forceable a where
    force :: a -> IO Result
    force v = v `seq` return T

seqList = L.foldl' (flip seq) (return ())

instance Forceable [a] where
    force = L.foldl' (flip seq) (return T)

instance Forceable Char
instance Forceable Bool
instance Forceable Int

run :: Int -> a -> [(String,[F a])] -> IO ()
run c x tests = sequence_ $ zipWith (runTest c x) [1..] tests

runTest :: Int -> a -> Int -> (String,[F a]) -> IO ()
runTest count x n (name,tests) = do
  let s = show n
  putStr ((if length s <= 1 then '0' : s else s) ++ "\t")
  fn tests
  putStr ("\t#" ++ (show name) ++ "\n")
  hFlush stdout
        where fn xs = case xs of
                        [f,g,h] -> runN count f x >> putStr "\t"
                                 >> runN count g x >> putStr "\t"
                                 >> runN count h x >> putStr "\t"
                        [f,g]   -> runN count f x >> putStr "\t"
                                >> runN count g x >> putStr "\t"
                        [f]     -> runN count f x -- >> putStr "\t"
                        _       -> return ()
              run f x = time f x
              runN 0 f x = return ()
              runN c f x = run f x >> runN (c-1) f x


time (Flist f) a = do
  start <- getCPUTime
  v     <- seqList (f a)
  end   <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12 :: Double
  putStr (takeWhile (/= '.') (show diff) ++ (take 4 (dropWhile (/= '.') (show diff))) ++ "\t")
  hFlush stdout

time (F f) a = do
  start <- getCPUTime
  v <- let x = (f a) in seq x (return x)
  end <- getCPUTime
  let diff = (fromIntegral (end - start)) / 10^12 :: Double
  putStr (takeWhile (/= '.') (show diff) ++ take 4 (dropWhile (/= '.') (show diff)) ++ "\t")
  hFlush stdout

data Stream a = forall s. Stream (s -> Step a s) s

data Step a s = Done
              | Yield a s
              | Skip s

unstream :: Stream a -> [a]
unstream (Stream next s) = unfold s
  where
    unfold s = case next s of
                 Done       -> []
                 Skip s'    -> unfold s'
                 Yield x s' -> x : unfold s'

stream :: [a] -> Stream a
stream lst = Stream next lst
  where
    next []     = Done
    next (x:xs) = Yield x xs

mapS :: (a -> b) -> Stream a -> Stream b
mapS f (Stream next s) = Stream next' s
  where
```

```
    next' s = case next s of
                Done       -> Done
                Skip s'    -> Skip s'
                Yield x s' -> Yield (f x) s'

filterS :: (a -> Bool) -> Stream a -> Stream a
filterS p (Stream next s) = Stream next' s
  where
    next' s = case next s of
                Done       -> Done
                Skip s'    -> Skip s'
                Yield x s' -> if p x then Yield x s' else Skip s'

foldlS :: (b -> a -> b) -> b -> Stream a -> b
foldlS f z (Stream next s) = foldlS_loop z s
  where
    foldlS_loop z s = case next s of
                      Done       -> z
                      Skip s'    -> foldlS_loop z s'
                      Yield x s' -> foldlS_loop (f z x) s'

foldrS :: (a -> b -> b) -> b -> Stream a -> b
foldrS f z (Stream next s) = foldrS_loop s
  where
    foldrS_loop s = case next s of
                      Done       -> z
                      Skip s'    -> foldrS_loop s'
                      Yield x s' -> f x (foldrS_loop s')

enumFromToS :: Int -> Int -> Stream Int
enumFromToS m n = Stream next m
  where
    next x = if x > n then Done else Yield x (x+1)


map' :: (a -> b) -> [a] -> [b]
map' f xs = unstream (mapS f (stream xs))

filter' :: (a -> Bool) -> [a] -> [a]
filter' p xs = unstream (filterS p (stream xs))

foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' f z xs = (foldlS f z (stream xs))

foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' f z xs = (foldrS f z (stream xs))

enumFromTo' :: Int -> Int -> [Int]
enumFromTo' m n = unstream (enumFromToS m n)

mapMapF :: [Int] -> [Int]
mapMapF xs = map' (subtract 1) (map' (+1) xs)

mapFilterF :: [Int] -> [Int]
mapFilterF xs = map' (+1) (filter' odd xs)

filterMapF :: [Int] -> [Int]
filterMapF xs = filter' odd (map' (+1) xs)

filterFilterF :: [Int] -> [Int]
filterFilterF xs = filter' odd (filter' (\ x -> x `mod` 3 /= 0) xs)

main :: IO ()
main = do let testL = P.enumFromTo 1 5000000 :: [Int]
          seqList testL
          let smallTestL = L.take 1500000 testL
          seqList smallTestL
          run 5 testL trans_tests
          putStr "\n"
          run 5 (1 :: Int,5000000 :: Int) prod_tests
          putStr "\n"
          run 5 smallTestL con_tests
          putStr "\n"
          run 5 (1 :: Int,1500000 :: Int) con_prod_tests

trans_tests :: [(String, [F [Int]])]
trans_tests = [ ("map . map"                        , [Flist mapMapF] )
              , ("map . filter"                     , [Flist mapFilterF])
              , ("filter . filter"                  , [Flist filterFilterF] )
              , ("filter . map"                     , [Flist filterMapF])
              ]

mapEnumFromToF (x,y) = map' (+1) (enumFromTo' x y)

filterEnumFromToF (x,y) = filter' odd (enumFromTo' x y)

mapFilterEnumFromToF (x,y) = map' (+1) (filter' odd (enumFromTo' x y))

filterMapEnumFromToF (x,y) = filter' odd (map' (+1) (enumFromTo' x y))

prod_tests = [ ("map . enumFromTo"                , [Flist mapEnumFromToF])
             , ("filter . enumFromTo"             , [Flist filterEnumFromToF])
```

```
            , ("map . filter . enumFromTo"        , [Flist mapFilterEnumFromToF])
            , ("filter . map . enumFromTo"        , [Flist filterMapEnumFromToF])
            ]

foldrMapF xs = foldr' (+) 0 (map' (+1) xs)

foldrFilterF xs = foldr' (+) 0 (filter' odd xs)

foldrMapFilterF xs = foldr' (+) 0 (map' (+1) (filter' odd xs))

foldrFilterMapF xs = foldr' (+) 0 (filter' odd (map' (+1) xs))

foldlMapF xs = foldl' (+) 0 (map' (+1) xs)

foldlFilterF xs = foldl' (+) 0 (filter' odd xs)

foldlMapFilterF xs = foldl' (+) 0 (map' (+1) (filter' odd xs))

foldlFilterMapF xs = foldl' (+) 0 (filter' odd (map' (+1) xs))

con_tests = [ ("foldr . map"                    , [F foldrMapF] )
            , ("foldr . filter"                 , [F foldrFilterF] )
            , ("foldr . map . filter"           , [F foldrMapFilterF] )
            , ("foldr . filter . map"           , [F foldrFilterMapF] )
            , ("foldl . map"                    , [F foldlMapF] )
            , ("foldl . filter"                 , [F foldlFilterF] )
            , ("foldl . map . filter"           , [F foldlMapFilterF] )
            , ("foldl . filter . map"           , [F foldlFilterMapF] )
            ]

foldrEnumFromToF (x,y) = foldr' (+) 0 (enumFromTo' x y)
foldrMapEnumFromToF (x,y) = foldr' (+) 0 (map' (+1) (enumFromTo' x y))
foldrFilterEnumFromToF (x,y) = foldr' (+) 0 (filter' odd (enumFromTo' x y))
foldrMapFilterEnumFromToF (x,y) = foldr' (+) 0 (map' (+1) (filter' odd (enumFromTo' x y)))
foldrFilterMapEnumFromToF (x,y) = foldr' (+) 0 (filter' odd (map' (+1) (enumFromTo' x y)))

foldlEnumFromToF (x,y) = foldl' (+) 0 (enumFromTo' x y)
foldlMapEnumFromToF (x,y) = foldl' (+) 0 (map' (+1) (enumFromTo' x y))
foldlFilterEnumFromToF (x,y) = foldl' (+) 0 (filter' odd (enumFromTo' x y))
foldlMapFilterEnumFromToF (x,y) = foldl' (+) 0 (map' (+1) (filter' odd (enumFromTo' x y)))
foldlFilterMapEnumFromToF (x,y) = foldl' (+) 0 (filter' odd (map' (+1) (enumFromTo' x y)))

con_prod_tests = [ ("foldr . enumFromTo"               , [F foldrEnumFromToF] )
                 , ("foldr . map . enumFromTo"         , [F foldrMapEnumFromToF])
                 , ("foldr . filter . enumFromTo"      , [F foldrFilterEnumFromToF])
                 , ("foldr . map . filter . enumFromTo", [F foldrMapFilterEnumFromToF])
                 , ("foldr . filter . map . enumFromTo", [F foldrFilterMapEnumFromToF] )
                 , ("foldl . enumFromTo"               , [F foldlEnumFromToF])
                 , ("foldl . map . enumFromTo"         , [F foldlMapEnumFromToF])
                 , ("foldl . filter . enumFromTo"      , [F foldlFilterEnumFromToF])
                 , ("foldl . map . filter . enumFromTo", [F foldlMapFilterEnumFromToF])
                 , ("foldl . filter . map . enumFromTo", [F foldlFilterMapEnumFromToF])
                ]
```