# Eliminating Intermediate Lists in pH

by

Jacob B. Schwartz

S.B., Massachusetts Institute of Technology, 1999

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2000



Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 12, 2000

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arvind
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

# Eliminating Intermediate Lists in pH

by

Jacob B. Schwartz

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2000, in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

The pervasive use of lists as intermediate data structures in functional programming can cause time and space inefficiency in naïvely compiled programs. Using compiler optimizations to remove intermediate lists has come to be known as *deforestation*. Many techniques for deforestation have been proposed, although few consider their impact on parallel languages. This thesis investigates a particular deforestation technique based on hylomorphisms and the category theory of types and describes how the technique is implemented in the compiler for pH, an implicitly-parallel dialect of Haskell.

## Resumo

La trapenetrema uzado de listoj kiel interaj datumstrukturoj en funkcia programado povas kaŭzi tempan kaj spacan malefikecojn en naive kompilerataj programoj. Uzi kompilerajn optimigojn por forigi interajn listojn nomiĝis *senarbigo*. Multaj teknikoj por senarbigi estas proponitaj, kvankam malmultaj konsideras sian efikon sur paralelaj lingvoj. Tiu tezo esploras specifan senarbigan teknikon bazita sur hilomorfioj kaj la kategorio-teorio pri tipoj kaj priskribas kiel la tekniko efektiviĝas en la kompilero por pH, implicite paralela dialekto de Haskell.

Thesis Supervisor: Arvind
Title: Professor of Computer Science and Engineering

## Acknowledgments

I would like to thank my thesis advisor Arvind for allowing me to do interesting research.

I would like to thank fellow graduate student and pH researcher Jan-Willem Maessen for providing feedback on this research and on drafts of this paper and for graciously answering every little question that I came to him with.

I would also like to thank fellow graduate student Mieszko Lis for his proofreading help.

Above all, I would like to thank my family for helping me reach this milestone and for putting up with my disappearance while I was writing this paper.

# Contents


**1 Introduction** **7**

1.1 The Problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.2 The Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**2 Background** **11**

2.1 `foldr/build` Deforestation . . . . . . . . . . . . . . . . . . . . . 11

2.1.1 Consuming Lists . . . . . . . . . . . . . . . . . . . . . . . 12

2.1.2 Producing Lists . . . . . . . . . . . . . . . . . . . . . . . 13

2.1.3 The `foldr/build` Rule . . . . . . . . . . . . . . . . . . . . 14

2.2 Parallel List Traversal . . . . . . . . . . . . . . . . . . . . . . . 15

2.2.1 Consuming Lists with `reduce/map` . . . . . . . . . . . . . . 17

2.2.2 `map/reduce` Optimization . . . . . . . . . . . . . . . . . . . 17

2.2.3 Producing Lists with `unfold` . . . . . . . . . . . . . . . . . 18

2.2.4 Other optimizations . . . . . . . . . . . . . . . . . . . . . 20

2.3 Calculational Fusion . . . . . . . . . . . . . . . . . . . . . . . . 23

2.3.1 Bananas, Lenses, and Envelopes . . . . . . . . . . . . . . . . 24

2.3.2 Algebraic Data Types . . . . . . . . . . . . . . . . . . . . . 26

2.3.3 Morphisms Over Generic Types . . . . . . . . . . . . . . . . 29

2.3.4 Acid Rain . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

2.3.5 Hylomorphisms in Triplet Form . . . . . . . . . . . . . . . . 32

2.3.6 Hylomorphism Fusion . . . . . . . . . . . . . . . . . . . . . 33

**3 Implementing Hylo Fusion** **36**

3.1 HYLO . . . 36

3.2 Representation in pH . . . 38

3.3 Deriving Hylomorphisms . . . 42

3.4 Inlining Hylomorphisms . . . 45

3.5 Restructuring Hylomorphisms . . . 46

3.5.1 Restructuring $\phi$ . . . 46

3.5.2 Restructuring $\psi$ . . . 51

3.6 Fusing Hylomorphisms . . . 53

3.6.1 Detecting $in_F$ . . . 55

3.6.2 Detecting $out_F$ . . . 58

3.6.3 Reconciling $in_F$ and $out_F$ . . . 58

3.6.4 Fusion . . . 59

3.6.5 Finding Compositions of Hylomorphisms . . . 60

3.7 Deriving $\tau$ and $\sigma$ . . . 64

3.7.1 Deriving $\tau$ . . . 65

3.7.2 Deriving $\sigma$ . . . 69

3.8 Revising the Restructuring Algorithms . . . 70

3.8.1 Recursive Restructuring of $\phi$ . . . 72

3.8.2 Recursive Restructuring of $\psi$ . . . 74

3.9 Examples . . . 75

3.9.1 Deforesting Lists . . . 75

3.9.2 Deforesting Trees . . . 78

**4 Related Work** **83**

**5 Conclusions** **85**

5.1 Preserving Parallelism . . . 87

# List of Figures


2-1 `map/reduce` optimization rules . . . . . . . . . . . . . . . . . . . . . . 19

3-1 pH Intermediate Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . 40

3-2 Revised Algorithm $\mathcal{E}$ . . . . . . . . . . . . . . . . . . . . . . . . . . 50

3-3 Revised Algorithm $\mathcal{S}_{\psi}$ . . . . . . . . . . . . . . . . . . . . . . . . . 54

3-4 Finding Fusible Compositions (Part I) . . . . . . . . . . . . . . . . . . 62

3-5 Finding Fusible Compositions (Part II) . . . . . . . . . . . . . . . . . . 63

3-6 Recursive Restructuring Algorithm $\mathcal{R}_{\phi}$ . . . . . . . . . . . . . . . . . . 73

3-7 Recursive Restructuring Algorithm $\mathcal{R}_{\psi}$ . . . . . . . . . . . . . . . . . . 74

# Chapter 1

# Introduction

## 1.1 The Problem

Common practice in functional programming is to use a compositional style. This style encourages programmers to break algorithms down into simple functions which can be written and analyzed separately. The individual functions are easier to reason about, thus making programming and debugging easier. This style also allows for greater code reuse.

Lists provide a "glue" which binds separate functions together into a single algorithm. They are the intermediate data-structures which are generated by one function and passed to the next to be consumed. In a naïve implementation of this process, most intermediate lists would be constructed, traversed once, and then discarded. This is horribly inefficient in space (and time), when the function can be more simply computed in a single iterative loop.

Consider the classic example of computing the sum of the first $n$ squares. In the functional language *Haskell* [15], one might write the computation as follows:

```
(sum (map square [1..n]))
```

Here, `sum` and `map` are functions defined in the extensive Haskell prelude (library), `square` represents for simplicity a function which squares an integer, and `[1..n]` is syntax for the list of integers from 1 to $n$. A straightforward compilation of this example would generate the list of integers from 1 to $n$ and apply `(map square)` to the list, thus generating a second list (of the the squares from 1 to $n^2$), to be then consumed by `sum` to finally produce the output. The generation of the two intermediate structures is wasteful, since we could accomplish the same computation in a single loop. Specifically, we could rewrite the composition of functions as a single function which generates no lists at all:

```
sumOfSquares n = if (n > 1)
                 then (square n) + (sumOfSquares (n-1))
                 else (square n)
```

We want to continue to use the compositional style of programming, so rather than make programmers change their ways, the solution is to make smarter compilers which can recognize when intermediate data-structures are wasteful and remove them. This process has come to be known, humorously, as *deforestation*.

## 1.2 The Thesis

The original goal of my research was to rewrite the traversal optimization pass of the pH compiler [20]. The pass was written by Jan-Willem Maessen using the deforestation rules he derived in his master's thesis [17]. It worked with higher-order representations of the code to be optimized rather than working directly on the code. There were known bugs in the pass which were difficult to fix because the code was so convoluted. In addition to cleaning up the compiler, it was hoped that rewriting the pass would allow me to tease out implicit optimizations being performed in the pass along with the deforestation transformations. These implicit optimizations (described in Section 2.2.4) are "clean up" rules which were necessary in the original implementation to ensure that the transformed code output by the deforestation pass was in its most efficient form. Some of the optimizations, however, might be useful on other parts of the code. Separating out the strictly deforestation rules would not only allow the general optimizations to be moved to their own compiler pass (or to an existing pass with other simplifying optimizations), but would reduce the complexity of the traversal optimization pass, making it simpler to read and debug.

The course of the research changed when I discovered a new deforestation technique which seemed more powerful than Maessen's scheme. The technique, proposed by Takano and Meijer in [24], is an extension of the `foldr/build` deforestation to other data types. They express recursion with the notation from the infamous "bananas and lenses" paper [19] and specifically use hylomorphisms as their canonical form.

Papers about the technique made miraculous claims about its power, including the ability to automatically (and efficiently) derive the canonical form from recursive definitions, the ability to deforest recursion over multiple data structures, and the ability to extend to arbitrary algebraic data types. These claims, coupled with the fact that other researchers in functional programming were working with hylomorphisms, made the option of using this technique in the pH compiler enticing. My research then shifted to discovering how the hylomorphism technique handled parallelism and whether it could be extended to work with the pH compiler.

After researching the hylomorphism technique, the place for my thesis within other deforestation work seemed clear. In Chapter 2, I describe three methods of deforestation. The first method is the traditional deforestation technique used in serial languages, such as Haskell. The second method is that devised by Maessen and implemented in the pH compiler and is simply an extension of the first method for parallel languages. The third method described is the hylomorphism fusion of Takano and Meijer, which is itself just a generalization of the traditional deforestation (first method). While the hylomorphism method does much to improve the power of traditional deforestation, it does not concern itself at all with parallelization. My work, then, would be to extend hylomorphism fusion for use in parallel languages, in the same way that Maessen extended traditional deforestation.

My plan for this project was to implement an existing serial hylomorphism fusion system in the hope that it would provide insight into a system for parallel fusion. When I began, the only presentation of an automatic system for hylomorphism fusion was Onoue, Hu, Iwasaki, and Takeichi's paper "A Calculational Fusion System HYLO" [21]. Implementing the algorithms in that paper proved to be a significant project in itself, leaving little time to explore parallelism. This thesis, then, discusses the implementation of that system and leaves the extension to a parallel system as future research. In Chapter 3, I discuss the HYLO algorithms in greater detail than the original paper, providing corrections and pointing out open questions along the way. I even present an important algorithm which was missing from the HYLO system. In Chapters 4 and 5, I conclude with a few thoughts on parallelism and pointers to papers that might be helpful in extending the HYLO system in various directions.

# Chapter 2

# Background

This chapter presents previous work on deforestation which is relevant to the work being done on the pH compiler. In particular, three techniques are described: traditional `foldr/build` deforestation (which is implemented in the Glasgow Haskell compiler [7]), `reduce/map` optimization (which was the first deforestation technique implemented in the pH compiler), and calculational fusion using hylomorphisms (which is the basis for the work in this thesis). An overview of the various methods for removing intermediate data structures can be found in [9]. A short history of deforestation (and the general problem of intermediate data structures) is presented in [16].

## 2.1 `foldr/build` Deforestation

As far as this thesis is concerned, the first major step in that history is the work by Gill, Launchbury, and Peyton Jones [10]. They proposed a system for transformation which requires that modules generate and consume lists in specific ways. The form they used, which I will describe below, is too restrictive for the programmer to be required to write all functions in that form. It is still a useful method, however, because one can implement the functions in the standard Haskell prelude in this form and reasonably expect programmers to write most of their programs with prelude functions. Automatic deforestation can then proceed on any function which is composed of these standard functions, including the sum of squares example.

### 2.1.1 Consuming Lists

The work of Gill et al. builds on the Bird-Meertens formalism [2]. Bird devised an algebra for describing operations on lists, from which algorithms can be derived. By proving theorems in this algebra, one can say something about the algorithms they represent. Gill et al. are specifically interested in using transformations in the algebra (which can be proven to be correct) to remove intermediate lists.

Bird describes a set of useful operations, which he calls reduction operators and which can convert lists into other kinds of values. The first reduction operator he presents is `reduce`, defined as follows:

$$\texttt{reduce}\ (\oplus)\ z\ \texttt{[]} = z$$
$$\texttt{reduce}\ (\oplus)\ z\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

The first argument is a binary operation $\oplus$ (on a specific data type) which, for `reduce` to be safe, must be associative and have $z$, the second argument, as its (left and right) identity. That is:

$$a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c$$
$$a \oplus z \equiv z \oplus a \equiv a$$

We can define `reduce` more constructively as follows (where `++` is the append operator for lists):

$$\begin{array}{lcl} \texttt{reduce}\ (\oplus)\ \texttt{z []} & = & \texttt{z} \\ \texttt{reduce}\ (\oplus)\ \texttt{z [e]} & = & \texttt{e} \\ \texttt{reduce}\ (\oplus)\ \texttt{z (a ++ b)} & = & \texttt{(reduce}\ (\oplus)\ \texttt{z a)} \oplus \texttt{(reduce}\ (\oplus)\ \texttt{z b)} \end{array}$$

The typical next step at this point is to disambiguate the definition of `reduce` by picking an arbitrary direction for the computation of the reduction to proceed. Gill et al. choose to represent `reduce` as a right fold, but either direction is perfectly legal:

```
reduce f z = foldl f z
reduce f z = foldr f z
```

where the fold operations are defined as follows:

$$\texttt{foldr} \ \oplus \ z \ [x_1, x_2, \ldots, x_n] = x_1 \oplus (x_2 \oplus (\cdots \oplus (x_n \oplus z)))$$
$$\texttt{foldl} \ \oplus \ z \ [x_1, x_2, \ldots, x_n] = (((z \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n$$

As mentioned before, we can define all of the list-consuming functions in the Haskell prelude as folds. For instance, in the sum of squares example, we could define `sum` and `map` as follows:

```
sum xs = foldr (+) 0 xs
map f xs = foldr (\ a b -> ((f a):b)) [] xs
```

Many other functions can be represented in this manner. Of interesting note are the following (where `(:)` is the `cons` operator):

```
xs ++ ys = foldr (:) ys xs
foldl f z xs = (foldr (\ b g a -> g (f a b)) id xs) z
```

### 2.1.2 Producing Lists

The Gill et al. approach also requires that list production be standardized. They note that the effect of `foldr f z` on a list is to replace the *cons* between each element in the list with the function `f` and to replace the *nil* at the end of the list with `z`. They then reason that if the construction of lists were abstracted over *cons* and *nil*, then the effect of applying fold to a list could be achieved by applying the abstracted list to `f` and `z`. They invented a function `build` to represent this idea:

```
build g = g (:) []
```

As an example, consider the list of integers from 1 to $n$ used in the sum of squares example. We could create that list using the following function:

```
from a b = if (a > b)
           then []
           else (a:(from (a+1) b))
```

which can be defined in terms of `build` as follows:

```
from a b  = build (from' a b)
from' a b = (\ c n -> if (a > b)
                      then n
                      else (c a (from' (a+1) b c n)))
```

All functions which generate lists should abstract them in this way if the compiler is to deforest compositions involving those functions. We cannot expect the programmer to use `build` in his programs, but we can expect the prelude functions to be written in this way (as we did with `foldr`). For example:

```
map f xs = build (\ c n -> foldr (\ a b -> c (f a) b) n xs)
```

### 2.1.3 The `foldr/build` Rule

With list consumption and production standardized, we can now use a single transformation rule to remove intermediate lists. That rule is:

$$\texttt{foldr}\ f\ z\ (\texttt{build}\ g) = g\ f\ z$$

This rule is the heart of the Gill, Launchbury, Peyton Jones paper. It is significant enough to be the only function highlighted with a box in that paper! To see how it works, consider the sum of squares example, rewritten using `foldr` and `build`. The expression

```
sum (map square (from 1 k))
```

will be desugared by the compiler into

```
foldr (+) 0 (build
  (\ c n -> foldr (\ a b -> c (square a) b) n (build
    (from' 1 k))))
```

In this scheme, functions which consume a list and return another list in the standard forms must themselves be of the form `build`...`foldr`. Thus, when we compose several such functions we produce an expression which has alternating `build` and `foldr`, as seen in the above equation. The `foldr/build` rule then tells us that we can remove all the intermediate occurrences of `foldr`...`build`, leaving an expression with a single `build` and a single `foldr` (assuming the composite function both consumes and produces a list). In the current example, the pairs which are deforested are

```
foldr (+) 0 (build
  (\ c n -> foldr (\ a b -> c (square a) b) n (build
    (from' 1 k))))
```

leaving the following deforested code:

```
(\ c n -> (from' 1 k) (\ a b -> c (square a) b) n) (+) 0
```

which we can $\beta$-reduce to produce

```
(from' 1 k) (\ a b -> (square a) + b) 0
```

## 2.2 Parallel List Traversal

The `foldr/build` deforestation works great, except that it requires list traversals be expressed using `foldr`. This imposes an arbitrary data-dependency on otherwise parallel computations, which serializes the computation. In a parallel compiler, we want the option of deciding how to traverse a list when there is no inherent "handedness" in the computation.

The solution implemented in the current pH compiler is due to Maessen [18]. He suggests that we back up to the point where Gill et al. choose to represent `reduce` as `foldr`, and instead stick with `reduce`.

As Maessen points out, `reduce` is not quite enough. There are still some operations which we cannot represent using only `reduce`. Maessen provides as an example, the following idea which we would like to express:

```
length (a ++ b) = (length a) + (length b)
```

If we compare the type of this function to the type of `reduce`, we can see what the problem is:

```
reduce :: (a -> a -> a) -> a -> [a] -> a
length :: [a] -> Int
```

What is needed is some way to transform the values of the elements of the list into values of the type to be returned. The solution is to introduce the `map` operator:

```
map f []       = []
map f [e]      = [(f e)]
map f (a ++ b) = (map f a) ++ (map f b)
```

Now we can represent `length` in terms of `reduce` and `map` as follows (where `const` is a function which returns the first of its two arguments and `.` is the composition operation for functions):

```
length = sum . map (const 1)
```

Both Bird [2] and Maessen [17] show that using the two operations `reduce` and `map` in this way we can express the homomorphism from the monoid of lists over type $\alpha$, given by the four-tuple $(\texttt{[}\alpha\texttt{]}, \texttt{++}, \texttt{[]}, \texttt{(:[])})$, to an arbitrary monoid $(\beta, \oplus, id_{\oplus}, i)$ as follows:

$$h\ xs = \texttt{reduce}\ \oplus\ id_{\oplus}\ \texttt{(map}\ i\ xs\texttt{)}$$

Thus, any data-type which is expressible as a monoid can be implemented with lists and list operations. Maessen [17] discusses how arrays and open lists are implemented this way in the pH compiler.

### 2.2.1 Consuming Lists with `reduce/map`

The way that Maessen avoids picking an arbitrary traversal direction is by using higher-order functions. He describes this process in three steps. First, the computation that will be done on each list element when it receives the data traveling down the "spine" of the traversal is packaged up. Then a higher-order function is used to string these packages together. These two steps are captured in the `reduce/map` structure. The final step is to call this higher-order function on the initial value of the traversal to actually perform the computation. Here are both fold operations represented in this form (where `id` is the identity function and `flip` on a function reverses the order of the first two arguments of that function):

```
foldr f z xs = (reduce (.) id (map f xs)) z
foldl f z xs = (reduce (flip (.)) id (map (flip f) xs)) z
```

Similarly, we can express `reduce` in this way:

```
reduce ⊕ z xs = reduce (\ l r t -> (l z) ⊕ (r z))
                id
                (map const xs)
                z
```

The optimizations that lead to deforestation will then assume that all list consumers are of the (canonical) form:

(`reduce` $a$ `id` (`map` $u$ *list*)) $t$

### 2.2.2 `map/reduce` Optimization

After showing that `reduce` and `map` can express the homomorphisms with respect to the append operator `++`, Bird then uses this to prove several promotion rules, which show how certain list operators can be "pushed" inwards through calls to `concat` (the reduction of a list by `++`). The promotion rules for `map` and `reduce` are:

$$
\begin{aligned}
\texttt{map f (concat } L\texttt{)} &= \texttt{concat (map (map f) } L\texttt{)}\\
\texttt{reduce } \oplus \texttt{ z (concat } L\texttt{)} &= \texttt{reduce } \oplus \texttt{ z (map (reduce } \oplus \texttt{ z) } L\texttt{)}
\end{aligned}
$$

Maessen derives some similar rules that illustrate the ordering assumptions present in `map` and `reduce`. The result of applying `map` to a list is unaffected by the ordering of that list. The same cannot be said for `reduce`, because the operation passed to it is not necessarily commutative. Maessen illustrates this idea with the following rules:

$$
\begin{aligned}
\texttt{map f (reverse } L\texttt{)} &= \texttt{reverse (map f } L\texttt{)}\\
\texttt{reduce } \oplus \texttt{ z (reverse } L\texttt{)} &= \texttt{reduce (flip } \oplus\texttt{) z } L
\end{aligned}
$$

When the operation of the `reduce` is commutative, the compiler would benefit from dropping `flip` in the above example. In order to tell the compiler that we do not care about the order of a list—when we are using it as a bag, for example—we can call the function `someOrder`. It takes a list as its argument and returns a permutation of that list. When the compiler reaches a call to `someOrder`, it drops any assumptions about ordering and then chooses the best ordering (by picking a traversal direction for each sublist) considering the traversal function it is constructing.

Having shown some of the rules which can be derived, Maessen puts it all together into a set of optimization rules (Figure 2-1), which he calls the `map/reduce` rules because the last three arguments to $\mathcal{LF}$ are the values of $a$, $u$, and $t$ in the canonical reduction given earlier using `map` and `reduce`, which is being applied to the expression in the double brackets.

If promotion rules were applied directly, there would still be redexes in the code after compilation. The `map/reduce` rules eliminate these redexes by building a higher-order representation of the computation. Specifically, it makes all calls to $a$ at compile time! This can be done because $a$ has a known regular structure.

### 2.2.3 Producing Lists with `unfold`

List generation needs to be standardized as well, if we are to eliminate lists and not just optimize their traversal. The current pH compiler provides a function `unfold` for

$$
\begin{array}{lcl}
\mathcal{LE}[\![\texttt{[}E_1\texttt{,}\ E_2\texttt{,}\ \ldots E_n\texttt{]}]\!] & = & \mathcal{LF}[\![\texttt{[}E_1\texttt{,}\ E_2\texttt{,}\ \ldots E_n\texttt{]}]\!]\ a_{list}\ u_{list}\ t_{list} \\
\mathcal{LE}[\![E_1\texttt{++}E_2]\!] & = & \mathcal{LF}[\![E_1\texttt{++}E_2]\!]\ a_{list}\ u_{list}\ t_{list} \\
 & & \qquad\qquad \textit{etc.}\text{ for other list expressions} \\
\mathcal{LE}[\![\texttt{reduce}\ f\ t\ L]\!] & = & \mathcal{LE}[\![\texttt{let z = }t\texttt{ in} \\
 & & \quad \texttt{in}\ \mathcal{LF}[\![L]\!]\ (\lambda\, l\, r\, t\, .\, f\ (l\ \texttt{z})\ (r\ \texttt{z}))\ \texttt{const}\ t]\!] \\
\mathcal{LE}[\![\texttt{foldr}\ f\ t\ L]\!] & = & \mathcal{LE}[\![\mathcal{LF}[\![L]\!]\ \texttt{(.)}\ f\ t]\!] \\
\mathcal{LE}[\![\texttt{foldl}\ f\ t\ L]\!] & = & \mathcal{LE}[\![\mathcal{LF}[\![L]\!]\ \texttt{(flip (.))}\ \texttt{(flip}\ f\texttt{)}\ t]\!]
\end{array}
$$

$$
\begin{array}{llcl}
\mathcal{LF}[\![\texttt{[}E_1\texttt{,}\ E_2\texttt{,}\ \ldots E_n\texttt{]}]\!] & a\ u\ t & = & ((u\ E_1)\ `a`\ (u\ E_2)\ `a`\ \ldots\ (u\ E_n))\ t \\
\mathcal{LF}[\![E_1\texttt{++}E_2]\!] & a\ u\ t & = & (\mathcal{LF}[\![E_1]\!]\ a\ u\ `a`\ \mathcal{LF}[\![E_2]\!]\ a\ u)\ t \\
\mathcal{LF}[\![\texttt{[]}]\!] & a\ u\ t & = & t \\
\mathcal{LF}[\![A\texttt{:}L]\!] & a\ u\ t & = & (u\ A\ `a`\ \mathcal{LF}[\![L]\!]\ a\ u)\ t \\
\mathcal{LF}[\![\texttt{map}\ f\ L]\!] & a\ u\ t & = & \mathcal{LF}[\![L]\!]\ a\ (u\ \cdot\ f)\ t \\
\mathcal{LF}[\![\texttt{concat}\ L]\!] & a\ u\ t & = & \mathcal{LF}[\![L]\!]\ a\ (\lambda\ l\ .\ \mathcal{LF}[\![l]\!]\ a\ u)\ t \\
\mathcal{LF}[\![\texttt{reverse}\ L]\!] & a\ u\ t & = & \mathcal{LF}[\![L]\!]\ (\texttt{swap}\ a)\ u\ t \\
\mathcal{LF}[\![\texttt{someOrder}\ L]\!] & a\ u\ t & = & \mathcal{LF}[\![L]\!]\ (\texttt{bothways}\ a)\ u\ t
\end{array}
$$

Figure 2-1: `map`/`reduce` optimization rules

the programmer to use (again, as long as the programmer sticks to list comprehensions and prelude functions, the use of this function will be hidden from him):

```
unfold p f v = h v
         where h v | p v       = []
                   | otherwise = (a:(h b))
                        where (a,b) = f v
```

This definition was arrived at because it is flexible enough that, when transformed with some additional rules, it generates different code depending on how it is consumed.

A similar function `synthesize` is described in [17]. It operates in the reverse manner of `reduce` in the way that `unfold` is the reverse of `foldr` and `foldl`.

We can now add two final rules:

$\mathcal{LF}[\![\texttt{unfold}\ P\ \ F\ \ V]\!]\ a\ u\ t\ =$

$$
\begin{array}{l}
\texttt{let h v acc | } \mathcal{LE}[\![P]\!] \texttt{ = acc} \\
\qquad\qquad\texttt{| otherwise = } (a\ (u\ \ e)\ \texttt{(h v')})\texttt{ acc} \\
\qquad\qquad\qquad\texttt{where (e,v') = } \mathcal{LE}[\![F]\!]\texttt{ v} \\
\texttt{in\ \ h } \mathcal{LE}[\![E]\!]\ t
\end{array}
$$

Otherwise:

$\mathcal{LF}[\![E]\!]\ a\ u\ t\ =$

$$
\begin{array}{l}
\texttt{let h []\ \ \ \ \ acc = acc} \\
\qquad\texttt{h (e:es) acc = } (a\ (u\ \ e)\ \texttt{(h es)})\texttt{ acc} \\
\texttt{in\ \ h } \mathcal{LE}[\![E]\!]\ t
\end{array}
$$

A more sophisticated set of transformations to replace these last two is given in [18]. Those transformations take into account that the structure of $a$ is known, where this naïve set of transformations does not. The result is code which builds up the list in a way that is best suited to the function which will consume it (defaulting to an iterative form when possible).

### 2.2.4 Other optimizations

As Gill notices in [9], there are optimizations other than straightforward deforestation which are necessary for Maessen's transformations to produce proper output. These optimizations are implicit in Maessen's implementation of deforestatation in the pH compiler. Gill specifically mentions arity analysis as one such optimization, since the same mechanism is needed in his implementation of `foldr/build` deforestation. Constant argument removal is another optimization used in Maessen's implementation.

As an example, consider what the compiler does when it encounters the following code (our infamous sum of squares example):

```
reduce (+) 0 (map (\ x -> x * x) xs)
```

Maessen's implementation sees this code, generates representations for $a$, $u$, and $t$, and runs the higher-order function to generate the final output—a loop, which would look something like this:

```
let h xs t = case xs of
               [] -> t
               (x:xt) -> (x * x) + (h xt t)
in  h xs 0
```

As we mentioned, there are some hidden optimizations required to produce this output. In Maessen's implementation, it is unclear whether these optimizations are being applied correctly. Their implicit presence in the optimization pass makes it difficult to reason about the correctness. A cleaner implementation would separate these rules from those which are strictly deforestation rules. For example, if we apply only the deforestation rules, the process might proceed as follows. First, the code is transformed into the canonical form:

```
_reduce (\ l r t -> (l t) + (r t))
        (\ t -> t)
        (_map (\ x t -> x * x) xs)
        0
```

Then the traversal optimization pass does its computation and outputs the following loop code:

```
let h xs = case xs of
             [] -> (\ t -> t)
             (x:xt) -> (\ l r t -> (l t) + (r t))
                         ((\ x t -> x * x) x)
                         (h xt)
in  h xs 0
```

This is different from the output given in the first example. The difference is that there is still much simplification that can be performed on the code. However, this is not simplification that the traversal optimizations should be concerned with. Trying to make this simplification part of the traversal optimization pass is part of the reason why the code is so convoluted. Instead, general code-simplifying passes should be used to shape the loop into a form that is most efficient for compilation.

Continuing with our example, we can apply $\beta$-reduction rules to achieve the following code:

```
let h xs = case xs of
             [] -> (\ t -> t)
             (x:xt) -> let f = h xt
                       in  (\ t -> (x * x) + (f t))
in  h xs 0
```

This is where we apply the arity analysis that Gill mentions. The remaining step is to notice that the function `f` is always applied to one argument. Thus we can $\eta$-abstract by adding an additional argument to the definition of `f`:

```
let h xs = case xs of
             [] -> (\ t -> t)
             (x:xt) -> let f t = h xt t
                       in  (\ t -> (x * x) + (f t))
in  h xs 0
```

Now we notice that the function `h` is always applied to at least two arguments, so it is safe to $\eta$-abstract the definition of `h`:

```
let h xs t = (case xs of
               [] -> (\ t -> t)
               (x:xt) -> let f t = h xt t
                         in  (\ t -> (x * x) + (f t))) t
in  h xs 0
```

Now we can perform some final $\beta$-reduction and other simplifying transformations to achieve the result of optimization in the first example. Without the arity analysis to raise the arity of `h`, the compiler would not realize that the function can be computed nicely as a tail-recursive loop.

It is worth noting that further optimization is still possible. Since the argument `t` never changes in recursive calls to `h`, it is safe to lift it out of the recursion. Since the function is only called once (and thus always called with the same value of `t`), we can go even further and replace all instances of `t` with the constant value:

```
let h xs = case xs of
             [] -> 0
             (x:xt) -> (x * x) + (h xt)
in  h xs
```

This optimization, like arity analysis, is straightforward.

## 2.3 Calculational Fusion

A third form of deforestation was introduced by Takano and Meijer in a paper called "Shortcut Deforestation in Calculational Form" [24]. The term *calculational* refers to the philosophy that one should *calculate* programs from their descriptions in the way a mathematician calculates the answer to a numerical problem [19]. It is easy to discuss statements about simple operators, like addition, multiplication, and composition; but in order to propose and prove statements about whole programs, one will eventually have to analyze recursion, which is not so easily done in the general case. Meijer, Fokkinga, and Paterson liken this situation to the use of `goto` statements, which are difficult to reason about and so have been replaced in modern languages by structured control flow primitives such as conditionals and `while`-loops which make reasoning feasible.

Using category theory, Meijer et al. extend Bird's calculational framework for recursion on lists into a calculus for describing recursion on arbitrary, inductively-defined data types. They propose a set of higher-order functions for representing certain recursion schemes, over which they can prove laws and which are powerful enough to describe the kinds of recursion seen in functional programs. These representations are the basis for Takano and Meijer's deforestation scheme.

### 2.3.1 Bananas, Lenses, and Envelopes

Before presenting the calculational framework over arbitrary data types, it is worth examining the specific case for lists. This will reduce the conceptual leap required to move from the `foldr/build` framework to the more general form.

Meijer et al. describe three forms of recursion. The first are functions from lists (a recursive data type) to some other arbitrary data type. Such functions destruct lists and are called *catamorphisms*, from the Greek preposition $\kappa\alpha\tau\alpha$ meaning "downwards." The second form of recursion are functions which construct lists, and thus are functions from an arbitrary data type to lists. Such functions are called *anamorphisms*, from the Greek preposition $\alpha\nu\alpha$ meaning "upwards." The third type are functions from one arbitrary data type to another arbitrary data type, but with a call structure in the shape of a list. These functions are called *hylomorphisms*, from the Aristotelian philosophy that form and matter are one, and $\upsilon\lambda o$ meaning "dust" or "matter" [19].

#### Catamorphisms

A list catamorphism is a function `h` of the form:

```
h :: [A] -> B
h []     = b
h (a:as) = a ⊕ (h as)
```

This should look familiar as it is the definiton of a fold. The function `h` can be expressed in Bird's notation as:

```
h = foldr ⊕ b
```

In the notation proposed by Meijer et al., list catamorphisms are written using "banana" brackets (so named for their shape):

$$h = (\!|b, \oplus|\!)$$

### Anamorphisms

A list anamorphism is a function of the form:

```
h :: B -> [A]
h b | (p b) = []
h b         = let (a,b') = g b
              in  (a:(h b'))
```

where `p` is a predicate that takes a value of type `B` and returns a boolean.

List anamorphisms construct a list and are represented in Bird's framework by the `unfold` function:

```
h = unfold p g
```

In the extended framework, list anamorphisms are written using a pair of brackets which looks like concave lenses:

$$h = [\!(g, p)\!]$$

### Hylomorphisms

A list hylomorphism is a recursive function whose call structure is isomorphic to a list—that is, a linear recursive function. Such functions have the form:

```
h :: A -> B
h a | (p a) = c
h a         = let (b,a') = g a
              in  b ⊕ (h a')
```

Hylomorphisms are indicated in the extended notation by double brackets, which Meijer et al. refer to as "envelopes":

$$h = [\![(c, \oplus), (g, p)]\!]$$

Notice that the form for hylomorphisms is the same form as for anamorphisms except that the list constructors *nil* and *cons* have been replaced by $\mathtt{c}$ and $\oplus$ respectively. This means that anamorphisms are just a special case of hylomorphisms. Similarly, catamorphisms are hylomorphisms with specific values for $\mathtt{p}$ and $\mathtt{g}$. This is because hylomorphisms represent the composition of a catamorphism and an anamorphism (or a fold and an unfold in Bird's framework):

$$[\![(\mathsf{c}, \oplus), (\mathsf{g}, \mathsf{p})]\!] \;=\; (\!|\mathsf{c}, \oplus|\!) \;\circ\; [\!(\mathsf{g}, \mathsf{p})\!]$$

### 2.3.2 Algebraic Data Types

Having described the three patterns of recursion in terms of cons-lists, we would now like to extend these ideas to arbitrary data types. In order to do this, we need to be able to discuss data types (and functions which operate on them) in a systematic way. Fortunately, there exists a theory of data types which we can use. This theory is based in category theory and defines recursive data types (also called algebraic data types) as the least fixed points of functors.

The following sections will briefly describe the algebra of data types. For a more thorough treatment of category theory and its applications to programming, see [1]. The category used in this work is $\mathcal{CPO}$, the category of complete partial orders with continuous functions. As we will see, this has the benefit that the carriers of initial algebras and final co-algebras coincide.

#### Functors

There are several basic functors we will need in order to define data types. These functors are *id* (identity), $\underline{A}$ (constants), $\times$ (product), $+$ (separated sum), and $A_\perp$ (strictify or lifting).

The product functor is defined as follows for both types and functions:

$$\begin{aligned} A \times B &= \{(a, b) \mid a \in A, b \in B\} \\ (f \times g)\ (a, b) &= (f\ a, g\ b) \end{aligned}$$

We also define the related combinators left projection, right projection, and split:

$$\begin{aligned} exl\ (a,b) &= a \\ exr\ (a,b) &= b \\ (f \vartriangle g)\ a &= (f\ a, g\ a) \end{aligned}$$

The relationship between these combinators and the product functor is characterized by the equation:

$$f \times g \;=\; (f \circ exl) \vartriangle (g \circ exr)$$

The separated sum functor is defined as follows for both types and functions:

$$\begin{aligned} A + B &= (\{0\} \times A \ \cup\ \{1\} \times B)_\perp \\ (f+g)\ \perp &= \perp \\ (f+g)\ (0,a) &= (0, f\ a) \\ (f+g)\ (1,b) &= (1, g\ b) \end{aligned}$$

The related combinators left injection, right injection, and junc are defined as follows:

$$\begin{aligned} inl\ a &= (0,a) \\ inr\ b &= (1,b) \\ (f \triangledown g)\ \perp &= \perp \\ (f \triangledown g)\ (0,a) &= f\ a \\ (f \triangledown g)\ (1,b) &= g\ b \end{aligned}$$

The relationship between these combinators and the separated sum is characterized by the equation:

$$f + g \;=\; (inl \circ f) \triangledown (inr \circ g)$$

### Initial Fixed Points of Functors

For our analysis, it is necessary that data types are defined by functors whose operation on functions are continuous. As it turns out, all functors in $\mathcal{CPO}$ which are built up from the basic functors just described satisfy this continuity condition. Having described the basic functors, we are now ready to define data types as the initial fixed points of functors.

Let $\mathsf{F}$ be an endofunctor on category $\mathcal{C}$. That is, $\mathsf{F}$ is a functor from $\mathcal{C}$ to $\mathcal{C}$. We can define several concepts based on this functor. An $\mathsf{F}$-*algebra* is a strict function of type $\mathsf{F}A \rightarrow A$. Dually, an $\mathsf{F}$-*co-algebra* is a (not necessarily strict) function of type $A \rightarrow \mathsf{F}A$. In both cases, we say that the set $A$ is the *carrier* of the algebra. An $\mathsf{F}$-*homomorphism* $h : A \rightarrow B$ from $\mathsf{F}$-algebra $\phi : \mathsf{F}A \rightarrow A$ to $\mathsf{F}$-algebra $\psi : \mathsf{F}B \rightarrow B$ is a function which satisfies the equation:

$$h \circ \phi \quad = \quad \psi \circ \mathsf{F}h$$

This is expressed more concisely as $h : \phi \rightarrow_F \psi$. We define the category $\mathcal{ALG}(\mathsf{F})$ as the category whose objects are $\mathsf{F}$-algebras and whose morphisms are $\mathsf{F}$-homomorphisms. Dually, $\mathcal{COALG}(\mathsf{F})$ is the category whose objects are $\mathsf{F}$-co-algebras and whose morphisms are $\mathsf{F}$-co-homomorphisms.

Now we see why the default category is $\mathcal{CPO}$. In this category, $\mathcal{ALG}(\mathsf{F})$ has an initial object and $\mathcal{COALG}(\mathsf{F})$ has a final object, and the carriers of both algebras coincide. Specifically, the initial object of $\mathcal{ALG}(\mathsf{F})$ is the $\mathsf{F}$-algebra $in_F : \mathsf{F}\mu\mathsf{F} \rightarrow \mu\mathsf{F}$ and the final object of $\mathcal{COALG}(\mathsf{F})$ is the $\mathsf{F}$-co-algebra $out_F : \mu\mathsf{F} \rightarrow \mathsf{F}\mu\mathsf{F}$, where $\mu$ is the fixed point operator that satisfies the equation $\mu h = h(\mu h)$. The algebras $in_F$ and $out_F$ are inverses of each other and therefore determine the isomorphism $\mu\mathsf{F} \cong \mathsf{F}\mu\mathsf{F}$ in $\mathcal{C}$. The type $\mu\mathsf{F}$ is the *algebraic data type* defined by the functor $\mathsf{F}$.

An example is definitely in order.

### Examples

These are examples are taken from [24].

Consider the following recursive type declaration in Haskell:

```
data Nat = Zero | Succ Nat
```

This declaration defines a functor $\mathsf{N} = \underline{\perp} + id$. That is, $\mathsf{N}A = \perp + A$ and $\mathsf{N}h = id_\perp + h$. It also defines the data type $nat = \mu\mathsf{N}$ and an initial $\mathsf{N}$-algebra $in_N = Zero \triangledown Succ : \mathsf{N}nat \to nat$, where $\perp$ is the terminal object in $\mathcal{C}$ and $Zero : \perp \to nat$ is a constant.

Now consider the parameterized declaration for lists of type $A$:

```
data List A = Nil | Cons A (List A)
```

This declaration defines a functor $\mathsf{L}_A = \underline{\perp} + \underline{A} \times id$, which by the definitions of the basic functors means that $\mathsf{L}_A B = \perp + (A \times B)$ and $\mathsf{L}_A h = id_\perp + (id_A \times h)$. As before, the data type is defined by the carrier $\mu\mathsf{L}_A$ of the initial $\mathsf{L}_A$-algebra $in_{L_A} = Nil \triangledown Cons : \mathsf{L}_A(listA) \to (listA)$.

We can also construct a final $\mathsf{L}_A$-co-algebra as follows:

$$out_{L_A} \quad = \quad (id_\perp + \mathtt{hd} \vartriangle \mathtt{tl}) \circ \mathtt{is\_nil}? : (listA) \to \mathsf{L}_A(listA)$$

Here $\mathtt{p}?$ injects a value $\mathtt{x}$ of type $A$ into the type $A + A$ depending on the result of $\mathtt{p}$ $\mathtt{x}$. One could therefore think of the definition of $out_{L_A}$ as corresponding to:

```
\ x -> if (is_nil x) then ⊥ else (hd x, tl x)
```

### 2.3.3 Morphisms Over Generic Types

#### Catamorphisms and Anamorphisms

The fact that $in_F$ is the initial object of $\mathcal{ALG}(\mathsf{F})$ means by definition that for any $\mathsf{F}$-algebra $\phi : \mathsf{F}A \to A$, there is a unique $\mathsf{F}$-homomorphism $h : in_F \to_F \phi$. This $\mathsf{F}$-homomorphism is just a catamorphism over the type $\mu\mathsf{F}$! We denote this catamorphism by $(\!|\phi|\!)_F$.

Dually, for $out_F$ to be the final object in $\mathcal{COALG}(\mathsf{F})$ means that for every $\mathsf{F}$-co-algebra $\psi : A \to \mathsf{F}A$, there is a unique $\mathsf{F}$-co-homomorphism $h : \psi \to_F out_F$, which is an anamorphism over the type $\mu\mathsf{F}$ and is denoted by $[\!(\psi)\!]_F$.

We can also define these two morphisms as least fixed points:

$$
\begin{aligned}
(\!|\_|\!)_F &: (\mathsf{F}A \to A) \to \mu\mathsf{F} \to A \\
(\!|\phi|\!)_F &= \mu(\lambda f.\ \phi \circ \mathsf{F}f \circ out_F) \\
[\!(\_)\!]_F &: (A \to \mathsf{F}A) \to A \to \mu\mathsf{F} \\
[\!(\psi)\!]_F &= \mu(\lambda f.\ in_F \circ \mathsf{F}f \circ \psi)
\end{aligned}
$$

From these fixed point definitions, we can derive the following useful equations:

$$
\begin{aligned}
(\!|\phi|\!)_F &= \phi \circ \mathsf{F}(\!|\phi|\!)_F \circ out_F \\
(\!|\phi|\!)_F \circ in_F &= \phi \circ \mathsf{F}(\!|\phi|\!)_F \\
[\!(\psi)\!]_F &= in_F \circ \mathsf{F}(\!|\psi|\!)_F \circ \psi \\
out_F \circ [\!(\psi)\!]_F &= \mathsf{F}(\!|\psi|\!)_F \circ \psi
\end{aligned}
$$

### Hylomorphisms

As was mentioned before in the specific case of lists, hylomorphisms are the composition of catamorphisms with anamorphisms. This is still true over arbitrary types. We denote hylomorphisms by:

$$
[\![\phi, \psi]\!]_F = (\!|\phi|\!)_F \circ [\!(\psi)\!]_F
$$

As with the other morphisms, we can also represent hylomorphisms as least fixed points:

$$
\begin{aligned}
[\![\_,\_]\!]_F &: (\mathsf{F}A \to A) \times (B \to \mathsf{F}B) \to B \to A \\
[\![\phi, \psi]\!]_F &= \mu(\lambda f.\ \phi \circ \mathsf{F}f \circ \psi)
\end{aligned}
$$

It should be obvious from the fixed point definitions that catamorphisms and anamorphisms are just special cases of hylomorphisms:

$$
\begin{aligned}
(\!|\phi|\!)_F &= [\![\phi, out_F]\!]_F \\
[\!(\psi)\!]_F &= [\![in_F, \psi]\!]_F
\end{aligned}
$$

Many useful laws for program calculation can be derived for hylomorphisms. One such law is called HyloShift and is expressed as follows:

$$\eta : \mathsf{F} \to \mathsf{G} \;\Rightarrow\; [\![\phi \circ \eta, \psi]\!]_F \;=\; [\![\phi, \eta \circ \psi]\!]_G$$

This law demonstrates that natural transformations can be moved between the two parameters of a hylomorphism. It will allow us to introduce a new notation for hylomorphisms in section 2.3.5.

### 2.3.4 Acid Rain

The goal of all of this work to structure recursion is that we will be able to fuse compositions of recursive functions. This is performed by a pair of rules which Meijer, following the tradition of humorous names, has dubbed the Acid Rain theorem.

Takano and Meijer first introduce Acid Rain as a single free [27] theorem:

$$g : \forall A \,.\, (\mathsf{F}A \to A) \to A \;\Rightarrow\; (\!|\phi|\!)_F \; (g \; in_F) = g \; \phi$$

This rule should make sense as an extension of the traditional `foldr/build` rule to all algebraic data types. That is, it is simply saying that a catamorphism (fold) over the recursive type defined by $\mathsf{F}$ composed with a function which is parameterized by the constructors of that data type (build) is equivalent to applying the parameterized function on the parameter of the catamorphism.

Takano and Meijer suggest that it is easier to apply this rule if it is expressed in terms of functions, so they present the Acid Rain rule again:

$$g : \forall A \,.\, (\mathsf{F}A \to A) \to B \to A \;\Rightarrow\; (\!|\phi|\!)_F \circ (g \; in_F) = g \; \phi$$

Takano and Meijer then note that since we are working in category theory, it makes sense to take the dual of this rule:

$$h : \forall A \,.\, (A \to \mathsf{F}A) \to A \to B \;\Rightarrow\; (h \; out_F) \circ [\!(\psi)\!]_F = h \; \psi$$

This rule does not have an equivalent in the work on traditional deforestation, however it is not hard to see that such an equivalent could exist. Gill [9] suggests that in the traditional framework this dual could be called the `unbuild/unfold` rule. As with `build`, `unbuild` would not have a Hindley-Milner type.

Thus we have two Acid Rain rules—one for catamorphisms and one for anamorphisms.

We are not finished yet, however. While we now have rules for fusing catamorphisms and anamorphisms, it is not clear how to go about applying these rules in an automatic system. We need methods for both finding reducible expressions and for deciding what order to apply the rules when redexes overlap. Takano and Meijer suggest that the first step to resolving these problems is to use hylomorphisms as the canonical representation for recursion. This seems like a natural choice since catamorphisms and anamorphisms are just special cases of hylomorphisms and previous work has shown that most reasonable functions can be represented as hylomorphisms [3].

### 2.3.5 Hylomorphisms in Triplet Form

Even with hylomorphisms as the canonical form, there is still a problem. Some functions can be represented as catamorphisms on their input types and anamorphisms on their output types. For example:

$$\begin{array}{rcl} \texttt{length} & = & (\!|Zero \triangledown (Succ \circ exr)|\!)_{L_A} \\ & = & [\!( (id_\perp + \texttt{tl}) \circ \texttt{is\_nil?} )\!]_N \end{array}$$

And in general, for any natural transformation $\eta : \mathsf{F} \to \mathsf{G}$, the HyloShift rule tells us that the following catamorphism and anamorphism are equivalent:

$$\begin{array}{rcl} (\!|in_G \circ \eta|\!)_F & = & [\![in_G \circ \eta, out_F]\!]_F \quad = \\ & = & [\![in_G, \eta \circ out_F]\!]_G \quad = \quad [\!(\eta \circ out_F)\!]_G \end{array}$$

If such hylomorphisms appear in a program we are attempting to deforest, we may miss an application of the Acid Rain theorem for catamorphisms because the function appears in anamorphism form, or vice versa. In order not to miss such opportunities, we would need to apply the HyloShift rule at every step. This is a lot of work to do, which we can avoid by using a new notation for hylomorphisms which keeps the natural transformation as a third parameter to the hylomorphism, separate from the other two parameters.

Specifically, the new least fixed point definition for hylomorphisms is:

$$\begin{aligned} [\![\_,\_,\_]\!]_{G,F} &: \forall A, B \,.\, (\mathsf{G}A \to A) \times (\mathsf{F} \to \mathsf{G}) \times (B \to \mathsf{F}B) \to (B \to A) \\ [\![\phi, \eta, \psi]\!]_{G,F} &= \mu(\lambda f.\ \phi \circ \eta \circ \mathsf{F}f \circ \psi) \end{aligned}$$

With this notation, we can represent natural transformations in a neutral form:

$$(\!|in_G \circ \eta|\!)_F \;=\; [\![in_G, \eta, out_F]\!]_{G,F} \;=\; [\!(\eta \circ out_F)\!]_G$$

Specifically, we can represent the function `length` as follows:

$$\texttt{length} \;=\; [\![in_N, id + exr, out_{L_A}]\!]_{N,L_A}$$

Takano and Meijer claim that with this notation, it becomes easier to judge whether a hylomorphism is a catamorphism or an anamorphism: If the third parameter of the hylomorphism is $out_F$, then it is an $\mathsf{F}$-catamorphism; if the first parameter is $in_G$, then it is a $\mathsf{G}$-anamorphism.

### 2.3.6 Hylomorphism Fusion

We can now restate the Acid Rain rules in terms of the triplet notation for hylomorphisms.

The Acid Rain rule for catamorphisms becomes the Cata-HyloFusion rule:

$$\begin{gathered} \tau : \forall A \,.\, (\mathsf{F}A \to A) \to \mathsf{F}A \to A \;\Rightarrow \\ [\![\phi, \eta_1, out_F]\!]_{G,F} \circ [\![\tau\ in_F, \eta_2, \psi]\!]_{F,L} \;=\; [\![\tau\ (\phi \circ \eta_1), \eta_2, \psi]\!]_{F,L} \end{gathered}$$

The Acid Rain rule for anamorphisms becomes the Hylo-AnaFusion rule:

$$\begin{gathered} \sigma : \forall A \,.\, (A \to \mathsf{F}A) \to A \to \mathsf{F}A \;\Rightarrow \\ [\![\phi, \eta_1, \sigma\ out_F]\!]_{G,F} \circ [\![in_F, \eta_2, \psi]\!]_{F,L} \;=\; [\![\phi, \eta_1, \sigma\ (\eta_2 \circ \psi)]\!]_{F,L} \end{gathered}$$

**Transformation Algorithm**

The transformation algorithm proposed by Takano and Meijer is simply to repetitively apply the hylomorphism fusion laws until there are no redexes left in the program. The one non-trivial aspect of this algorithm is the reduction strategy that decides the order in which redexes are transformed. Despite the use of hylomorphisms as the canonical form, the order in which the Acid Rain rules are applied can change the amount of deforestation possible.

Because there are two kinds of redexes (Cata-Hylo and Hylo-Ana), there are four different ways in which two redexes can overlap:

1. Two Cata-Hylo redexes overlap:
   The reduction of the left redex does not destroy the right one.

2. Two Hylo-Ana redexes overlap:
   The reduction of the right redex does not destroy the left one.

3. A Cata-Hylo redex on the left overlaps with a Hylo-Ana redex on the right:
   The reduction of either redex does not destroy the other.

4. A Hylo-Ana redex on the left overlaps with a Cata-Hylo redex on the right:
   The reduction of either redex does destroy the other.

From these cases, Takano and Meijer conclude that it is important to reduce a series of redexes of the same kind (that is, cases 1 and 2) in the right order. They propose the following reduction strategy to maximize the opportunities for deforestation:

1. Reduce all maximal Cata-Hylo redex chains from left to right.

2. Reduce all maximal Hylo-Ana redex chains from right to left.

3. Simplify the inside of each hylomorphism using reduction rules for the basic functors and combinators.

4. If there still exist any redexes for HyloFusion rules, return to step 1.

It was noticed by Paterson [23] that the Acid Rain theorems could each be generalized to a form that would not make overlapping redexes sensitive to the order of reduction. His rules, which also appear in [21], differ from the originals in that they allow the intermediate functors of two composed hylomorphisms to be different as long as the $\phi$ ($\psi$) of one can be reduced to $\tau\ in$ ($\sigma\ out$) of the other. The rule for Cata-HyloFusion is:

$$\begin{gathered}\tau : \forall A\,.\,(\mathsf{F}A \to A) \to \mathsf{F}'A \to A \;\Rightarrow \\ [\![\phi, \eta_1, out_F]\!]_{G,F} \circ [\![\tau\ in_F, \eta_2, \psi]\!]_{F',L} \;=\; [\![\tau\ (\phi \circ \eta_1), \eta_2, \psi]\!]_{F',L}\end{gathered}$$

With this rule, and a similar rule for Hylo-AnaFusion, it is not necessary to reduce chains of the same redex in a particular order.

# Chapter 3

# Implementing Hylo Fusion

The first step towards implementing a parallel hylomorphism fusion pass in the pH compiler is to implement an existing non-parallel system. The only practical implementation of the hylomorphism fusion theory was presented by Onoue, Hu, Iwasaki, and Takeichi in [21]. This chapter describes how the algorithms in that paper are implemented in an optimization pass for the pH compiler.

## 3.1 HYLO

As Onoue et al. point out, the major difficulty in implementing the hylomorphism theory is that the transformation laws have so far been developed as aids for program calculation by hand and not by machine. To develop an automatic system for program calculation, one needs to re-express the transformation laws in a constructive form. The work by Onoue et al. attempts to do this. They developed an automatic fusion system called HYLO and give, in the paper, all the algorithms used by this system.

The second difficulty, according to Onoue et al., is developing a system for practical use. As we will see in their implementation, sometimes it is impossible to give a constructive algorithm for all possible inputs. In these cases, Onoue et al. settle for algorithms which cover only the forms of input one might reasonably expect in practice.

The basic steps of the HYLO system are as follows:

1. Derive Hylomorphisms from Recursive Definitions

   Hylomorphisms are not in the source language, only in the intermediate syntax used by the HYLO system. Thus, the system cannot expect programmers to write their recursion in this way. Further, the HYLO system does not assume that functions in a standard prelude are defined as hylomorphisms. The HYLO system follows the lead of Launchbury and Sheard [16] and derives hylomorphisms from arbitrary recursive definitions. In actuality, not all recursive definitions can be turned into hylomorphisms, but Onoue et al. argue that their algorithm covers a sufficient range of inputs that appear in practice.

2. Restructure Hylomorphisms to expose Data Production and Consumption

   As we saw in the previous chapter, for fusion of hylomorphisms to proceed by the Acid Rain laws, $\phi$ and $\psi$ need to be of the form $\tau\ in$ and $\sigma\ out$. To put hylomorphisms in this form, the HYLO system first extracts computations from $\phi$ and $\psi$ that can be moved to $\eta$. This process, called restructuring, is performed by two algorithms, one for removing computation from $\phi$ and one for removing computation from $\psi$. At this point, if $\phi$ and $\psi$ are not already *in* or *out*, the HYLO system attempts to derive $\tau$ and $\sigma$ such that $\phi = \tau\ in$ and $\psi = \sigma\ out$. Again, one algorithm is given for $\phi$ and one for $\psi$.

3. Apply Acid Rain

   The Acid Rain law is applied to fuse adjacent hylomorphisms. After fusion, it may be necessary to restructure the resulting hylomorphism to allow fusion with other hylomorphisms. The fusion may also have brought nested hylomorphisms together, requiring restructuring and application of the fusion law inside of hylomorphisms. Thus steps 2 and 3 are repeated until no fusible hylomorphisms remain.

4. Inline Remaining Hylomorphisms

   Any hylomorphisms still in the program must be replaced by a recursive definition. Onoue et al. do not give an algorithm for this process except to say that it is the inverse process of hylomorphism derivation.

## 3.2 Representation in pH

Even though Onoue et al. are concerned with developing constructive algorithms for calculational fusion, their presentation is still in the abstract notation of category theory. When deriving and manipulating hylomorphisms in the pH compiler, it is helpful not to stray too far from the intermediate syntax that the compiler pass is given. At the same time, while we do not want to give up the syntax in which the program is already represented, we also do not want to have to bring the category theory into this syntax by defining operators in the syntax for concepts like separated sum. The solution is to decide what information from a function is necessary to compute the hylomorphism, and to store only that information.

Examining the HYLO algorithms, one finds that the hylomorphism $[\![\phi, \eta, \psi]\!]_{F,G}$ is always of the form:

$$\begin{array}{lcl}
\mathsf{F} & = & \mathsf{F}_1 + \cdots + \mathsf{F}_n \\
\mathsf{F}_i & = & \Gamma(v_{i_1}) \times \cdots \times \Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} \\
\mathsf{G} & = & \mathsf{G}_1 + \cdots + \mathsf{G}_n \\
\mathsf{G}_i & = & \Gamma(v_{i_1}) \times \cdots \times \Gamma(v_{i_{m_i}}) \times I_1 \times \cdots \times I_{l_i} \\
\phi & = & \phi_1 \triangledown \cdots \triangledown \phi_n \\
\phi_i & = & \lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ t_{\phi_i} \\
\eta & = & \eta_1 + \cdots + \eta_n \\
\eta_i & = & \lambda(v_{i_1}, \ldots, v_{i_{m_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ (t_{i_1}, \ldots, t_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}) \\
\psi & = & \lambda v_s.\ \mathsf{case}\ t_0\ \mathsf{of}\ p_1 \rightarrow (1, t_{\psi_1}); \cdots; p_n \rightarrow (n, t_{\psi_n}) \\
t_{\psi_i} & = & (t_{i_1}, \ldots, t_{i_{m_i}}, t'_{i_1}, \ldots, t'_{i_{l_i}})
\end{array}$$

Here, variables are represented by $v$, expressions by $t$, and $\Gamma$ is the constant functor that returns the type of its argument. The functors $\mathsf{F}_i$ and $\mathsf{G}_i$ represent tuples whose first values are regular variables with the given type ($\Gamma(v)$) and whose remaining values are recursive values ($I$). That is, they are values of the recursive data type. For example, for the list data type, the functor corresponding to `(x:xs)` would be $\Gamma(x) \times I$. The variables corresponding to an $I$ in the functor are marked with a prime to indicate that they are recursive variables.

As it turns out, the specific types returned by each $\Gamma$ are unused in the HYLO algorithms. The functors are only needed to indicate which variables are recursive and which are not. If we keep track of this inside $\phi$, $\eta$, and $\psi$, then we do not need the functors in our representation.

All we need to represent in order to represent a hylomorphism are the three components from the triplet notation: $\phi$, $\eta$, and $\psi$. For $\phi$ and $\eta$, we keep a list of $n$ elements, which are the representations for the individual $\phi_i$ and $\eta_i$. Each $\phi_i$ is a triple whose elements are a list of the non-recursive variables $v$, a list of the recursive variables $v'$, and the expression $t_{\phi_i}$. Each $\eta_i$ is also a triple whose elements are a list of the non-recursive variables $v$, a list of the recursive variables $v'$, and a list of the expressions $t$ that make up the first part of the tuple that $\eta_i$ returns. The second part of the tuple is just the recursive variables, and since we already have them in the representation, there is no need to store them again. Finally, $\psi$ is a 4-tuple consisting of the variable $v_s$ over which the hylomorphism inducts, the expression $t_0$ given to the case-statement, a list of $n$ 2-tuples representing the patterns $p$ by indicating the constructor and its variables, and a list of $n$ 2-tuples representing the tuples $t_{\psi_i}$ by indicating the recursive values separately from the non-recursive. To put things more concretely, here are the type declarations for the three components of a hylomorphism:

```
type Phi = [([Var],[Var],Expr)]
type Eta = [([Var],[Var],[Expr])]
type Psi = (Var,Expr,[(Constr,[Var])],[([Expr],[Expr])])
```

This definition was not purely derived from the category theory definition of a hylomorphism; it co-evolved with the algorithms that use it, so some of the reasons for this representation might not become clear until we introduce the algorithms. This definition also depends a bit on the intermediate syntax used in the optimization passes of the pH compiler. The syntax also influences most of the algorithms that will follow, so we present it here in Figure 3-1.

This isn't the whole hylomorphism representation, yet. While developing the algorithms for finding fusible hylomorphisms, it became clear that some additional

```
data Expr = Apply Var [Expr]
          | Lambda [Var] Expr
          | Constr ConstrID [Expr]
          | Case Expr [(ConstrID, [Var], Expr)] (Var,Expr)
          | LetRec [Def] Expr
          | Let Var Expr Expr

data Def  = Def Var Expr
          | Par [Def]
          | Seq [Def]
```

*Variables are represented as applications with no arguments and constants are represented as constructors with no arguments.*

Figure 3-1: pH Intermediate Syntax

information is needed. Technically, a hylomorphism is just the sum of the three components, but most functions which reduce to hylomorphisms actually have additional parameters besides the recursive argument $v_s$ in the $\psi$ component of the hylomorphism. Thus, we don't often find hylomorphisms composed directly with hylomorphisms. Instead, we find hylomorphisms composed with $\lambda$-abstractions and let-statements which have a hylomorphism at their core. If we incorporate these "wrapper" statements into our hylomorphism representation, we can work directly with compositions of hylomorphisms. We incorporate this information by appending a context to the representation. To make working with the context easier, we don't store an actual expression for the context, but instead store a list of surrounding statements using the following data type:

```
data HyloContext = Apply e
                 | Lambda [Var]
                 | Let [Def]
                 | Case (Expr -> Expr)
```

The first three disjuncts are necessary for implementing the fusion system. Hylomorphisms like `map` have extra arguments beyond the recursive argument that need to be recorded. This is done with the `Lambda` disjunct. As the hylomorphism is applied, those $\lambda$-variables become bound to values, which are represented with the `Let` disjunct. Let-statements surrounding a hylomorphism can also be incorporated directly into the representation as `Let` values. Since let-statements, $\lambda$-abstractions, and applications can be interwoven in the code, this information needs to be captured in a single list to preserve the order. The final argument to a hylomorphism, the recursive argument, also needs to be stored. This is the sole purpose of the `Apply` disjunct.

The fourth disjunct is a special case that we have added to handle a common occurrence inside the pH compiler. The compiler is aggressive about transforming functions into the following form:

```
f tuple xs = case tuple of
               (a,b) -> case xs of
                          [] -> ...
                          (x:xs) -> ...
```

Deconstructing the non-recursive arguments before the recursive arguments is not a form that the algorithm for deriving hylomorphisms expects. To solve this problem, we store the case-statement as part of the context. We store in the `Case` disjunct a function which takes an expression (in our case, the hylomorphism) and outputs that expression with the case-statement added around it. This simplifies the inlining of the hylomorphism later on. It also allows us to capture a larger group of case-statements. Using information from a previous pass of the compiler, we can detect whether all but one branch of a case-statement results in bottom. In such cases, we do not need to push code into the non-terminating branches, so we can take shortcuts and consider the case-statement almost like a let-binding. When we encounter such case-statements, we can separate them into the non-bottom expression and the context for that expression, which we can store as a `Case` disjunct of the `HyloContext` data type.

The final representation of a hylomorphism in our system is thus:

```
type Hylo = (Phi,Eta,Psi,[HyloContext])
```

These descriptions of hylomorphisms should make more sense after we have seen a few algorithms. In particular, the process for deriving hylomorphisms is quite revealing.

## 3.3 Deriving Hylomorphisms

The algorithm for deriving hylomorphisms from recursive definitions is first presented by Hu et al. in [11] and is expanded upon in the comprehensive paper by Onoue et al. [21]. The algorithm derives hylomorphisms from functions of the form:

$$f = \lambda v_s.\ \texttt{case}\ t_0\ \texttt{of}\ p_1 \to t_1\,;\cdots;p_n \to t_n$$

The trick is to replace each $t_i$ with an equivalent expression $g_i\ t'_i$. This produces a new definition for the function:

$$f = \lambda v_s.\ \texttt{case}\ t_0\ \texttt{of}\ p_1 \to g_1\ t'_1\,;\cdots;p_n \to g_n\ t'_n$$

We can then lift all the $g_i$s out of the case-statement to produce a compositional definition for the function:

$$f = (g_1 \triangledown \cdots \triangledown g_n) \circ (\lambda v_s.\ \texttt{case}\ t_0\ \texttt{of}\ p_1 \to (1, t'_1)\,;\cdots;p_n \to (n, t'_n))$$

Now, if the individual $g_i$s can be expressed as $\phi_i \circ F_i\ f$ where $F_i$ is a functor, then we can show that:

$$g_1 \triangledown \cdots \triangledown g_n = (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ (F_1 + \cdots + F_n)\ f$$

Substituting this into the definition of the function $f$ we arrive at:

$$f = (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ (F_1 + \cdots + F_n)\ f \circ (\lambda v_s.\ \texttt{case}\ t_0\ \texttt{of}\ p_1 \to (1, t'_1)\,;\cdots;p_n \to (n, t'_n))$$

And by the definition of a hylomorphism, we have that:

$$f = [\![\phi, id, \psi]\!]_{F,F}$$

This is nice, but how do we go about finding $\phi_i$, $F_i$, and $t'_i$ for each $t_i$? According to Hu et al. [11] this is done in five steps:

1. Identify all the recursive calls to $f$ in $t_i$, and call them $f\ t_{i_1}, \ldots, f\ t_{i_{l_i}}$.
2. Find all the free variables in $t_i$ but not in $t_{i_1}, \ldots, t_{i_{l_i}}$, and call them $v_{i_1}, \ldots, v_{i_{k_i}}$.
3. Define $t'_i$ by tupling all the arguments of the recursive calls obtained in step 1 and the free variables obtained in step 2. That is,

$$t'_i = (v_{i_1}, \ldots, v_{i_{k_i}}, t_{i_1}, \ldots, t_{i_{l_i}}).$$

4. Define $F_i$ according to the construction of $t'_i$ by

$$F_i = \Gamma(v_{i_1}) \times \cdots \times \Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i},$$

   where $I_1 = \cdots = I_{l_i} = I$ and $\Gamma$ returns the type of the given variable.

5. Define $\phi_i$ by abstracting all the recursive function calls in $t_i$ by

$$\phi_i = \lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).t_i[f\ t_{i_1}/v'_{i_1}, \ldots, f\ t_{i_{l_i}}/v'_{i_{l_i}}],$$

   where $v'_{i_1}, \ldots, v'_{i_{l_i}}$ are new variables introduced for replacing occurrences of $f\ t_{i_1}, \ldots, f\ t_{i_{l_i}}$ in $t_i$.

Hu et al. give a formal, constructive algorithm for this process on a simplified language and then provide a proof of its correctness. This algorithm is expanded to a more complicated language by Onoue et al. The algorithm, called $\mathcal{A}$, uses an auxiliary algorithm $\mathcal{D}$ which returns a triple containing the free variables found in step 2, a set of associations $\{(v'_{i_1}, f\ t_{i_1}), \ldots, (v'_{i_{l_i}}, f\ t_{i_{l_i}})\}$ found in step 1, and $t_i$ with each occurrence of $f$ replaced by its associated new variable $v'$. Algorithm $\mathcal{D}$ takes as input a list of bound variables (which it builds up as it makes recursive calls to itself). When $\mathcal{D}$ recurses down to a variable, it checks to see if that variable is in the list of bound variables or in some nebulous "*global_vars*" and if the variable appears in neither then it is deemed to be a free variable and output in the first part of the tuple. This is, however, not constructive. To solve this problem, we realized that the free variables in $t_i$ with which we are concerned are simply the variables in the pattern $p_i$. Thus, we pass these variables to the algorithm $\mathcal{D}$ which then considers a variable to be free only if it appears in this list.

The algorithm by Onoue et al. only works for specific recursive functions $f$. It first requires that $f$ inducts over its last argument. The position is not important, because we can rearrange the arguments of a function to make the inductive argument the last one, but it does mean that the function $f$ must induct over a single argument. Further, it is required that recursive calls to $f$ be saturated and that all other arguments be constant in the recursive calls. In this way, the non-inductive arguments can be lifted out to reduce $f$ to a single-argument function as expected by the simplified algorithm given by Hu et al.

The pH intermediate syntax allows for default cases, which are not accounted for in the given derivation algorithm. Case-statements with default cases would need to be converted to complete case-statements, by duplicating the default case for each missing constructor, in order for derivation, and thus fusion, to proceed. This conversion can result in code duplication, so some examination might be necessary to determine whether the possible fusion is worth duplicating code. Our current implementation simply ignores case-statements with default cases which have not been converted by earlier passes of the compiler.

Implementing the derivation algorithm for pH was straight-forward, but there is still one issue looming. The intermediate syntax used in the pH compiler does not allow nested constructors in the patterns of case-statements. Patterns in pH with nested constructors result in nested case-statements in the intermediate syntax. The derivation algorithm would need to recognize such cases and produce a representation that could be handled by the other algorithms in the HYLO system. This potential problem has not been explored as part of this project, although several solutions do seem possible.

See Section 3.9 for examples of the derivation algorithm on sample programs.

## 3.4 Inlining Hylomorphisms

Inlining is the reverse problem of deriving hylomorphisms. It involves taking a hylomorphism in triplet form (or 4-tuple) and producing a recursive definition in the pH intermediate syntax. Onoue et al. dismiss this problem as simply the inverse of the derivation algorithm and move on. For the sake of completeness, here is how inlining is implemented in the pH compiler.

Given a function $f = [\![\phi, \eta, \psi]\!]$ (if you want to inline a hylomorphism which is not named, of course, you simply give it a fresh identifier), compose $\phi$ and $\eta$ into a single $\phi'$, producing a hylomorphism with the identity function as its middle parameter:

$$
\begin{aligned}
& f = [\![\phi', id, \psi]\!] \\
& \mathit{where} \quad \phi' = \phi_1 \triangledown \cdots \triangledown \phi_n \\
& \qquad\quad\;\; \phi_i = \lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ (t_{i_1}, \ldots, t_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}) \\
& \qquad\quad\;\; \psi = \lambda v_s.\ \texttt{case}\ t_0\ \texttt{of}\ p_1 \to (1, t'_1); \cdots; p_n \to (n, t'_n) \\
& \qquad\quad\;\; t'_i = (u_{i_1}, \ldots, u_{i_{k_i}}, u'_{i_1}, \ldots, u'_{i_{l_i}})
\end{aligned}
$$

This is now in the form that the derivation algorithm outputs. To perform the inverse operation, we need to derive the original $t_i$ for each $\phi_i$ and associated $t'_i$ from $\psi$. This is done by replacing all occurrences of the non-inductive variables $v_{i_1}, \ldots, v_{i_{k_i}}$ in $t_{i_1}, \ldots, t_{i_{k_i}}$ with the values $u_{i_1}, \ldots, u_{i_{k_i}}$ from $\psi$ and replacing all the inductive variables $v'_{i_1}, \ldots, v'_{i_{l_i}}$ with the function $f$ applied to the variables $u'_{i_1}, \ldots, u'_{i_{l_i}}$ from $\psi$. More succinctly:

$$t_i = \phi_i\ (u_{i_1}, \ldots, u_{i_{k_i}}, f\ u'_{i_1}, \ldots, f\ u'_{i_{l_i}}).$$

Seeing this equation, it is now clear that it can be derived from the equation in step 5 of the hylomorphism derivation algorithm. The final output of inlining is, of course:

$$f = \lambda v_s.\ \texttt{case}\ t_0\ \texttt{of}\ p_1 \to t_1; \cdots; p_n \to t_n$$

Remember, though, that we are carrying around context information for the hylomorphism, including any other arguments to the function $f$ and any values they might have been bound to along the way. This information needs to be output as well. Creating the context is straightforward, using the appropriate intermediate syntax statement for each disjunct of the context type. Some simplifying transformations might be necessary to optimize a naïvely generated context.

## 3.5 Restructuring Hylomorphisms

Restructuring is the process by which computation from $\phi$ or $\psi$ is transfered to $\eta$. This is an important part of the fusion process because it helps us discover when a hylomorphism is simply a catamorphism or anamorphism. Restructuring can also bring together hylomorphisms inside separate components of a larger hylomorphism so that they may be fused. Onoue et al. claim that it also helps reduce $\phi$ and $\psi$ to forms from which $\tau$ and $\sigma$ can more easily be derived. The algorithm as given by Onoue et al., however, seems to sometimes remove the possibility of derivation. This problem is discussed in Section 3.7.1.

Separate algorithms are needed for restructuring $\phi$ and restructuring $\psi$.

### 3.5.1 Restructuring $\phi$

Onoue et al. give a constructive description of an algorithm $\mathcal{S}_\phi$ to restructure a function $\phi$ into $\phi = \phi' \circ \eta_\phi$. This algorithm makes use of an auxiliary algorithm $\mathcal{E}$ to detect the maximal subterms of $\phi$ which can be abstracted out of the bodies $t_i$ of the $\phi_i$. The subterms which can be abstracted are those which do not contain recursive variables and whose free variables are a subset of the pattern variables. This last condition is important because we might encounter let-statements and $\lambda$-abstractions inside $t_i$ and we do not want, for instance, to abstract out the `x+1` in `\x.x+1`.

Here is an example of what we would like to happen, taken from Onoue et al. Given the function `map`, we derive a hylomorphic representation:

$$\texttt{map}\ \ g = [\![ Nil \triangledown \lambda(a, v_1').\ Cons(g\ a, v_1'), id, out ]\!]$$

In this hylomorphism, $\phi_1 = \lambda().\ Nil$ and $\phi_2 = \lambda(a, v_1').\ Cons(\underline{g\ a}, v_1')$. Here, $v_1'$ is a recursive variable and, according to Onoue et al., there is only one maximal non-recursive subterm as shown underlined. Following through with the restructuring algorithm would result in the following composition:

$$\begin{aligned}
&\phi_2 = \phi_2' \circ \eta_{\phi_2} \\
&\text{where } \phi_2' = \lambda(u_{2_1}, v_1').\ Cons(u_{2_1}, v_1') \\
&\qquad\ \ \eta_{\phi_2} = \lambda(a, v_1').\ (g\ a, v_1')
\end{aligned}$$

We now notice that $\phi' = \phi_1 \triangledown \phi_2'$ is the *in* function for lists, making `map` available for deforestation in more situations.

The attentive reader will note that *Nil* is not underlined. Why is it not abstracted out of $\phi_1$ by the restructuring algorithm? If we follow the definition for $\mathcal{E}$ given above, and, in fact, the algorithm as given by Onoue et al., then *Nil* ought to be returned as a maximal subterm not containing any recursive variables. The definition as given performs a legitimate transformation, moving as much work as possible from $\phi$ into $\eta$. However, Onoue et al. expect the restructuring algorithm to leave $\phi$ in *in* form when such a form exists. If the algorithm abstracts *Nil* in the case above, it would not return $\phi'$ in the *in* form. In some sense, it would take too much computation out of $\phi$. If we are to use the restructuring algorithm for the purpose of uncovering *in* form, we need to change algorithm $\mathcal{E}$ to recognize these cases and not mark them for abstraction. In order to do this, however, we ourselves need to have some notion of what the possible cases are. So far we have only one example.

A first thought might be to prevent the abstraction of all constants (constructors with no arguments), but this can't be the whole picture. As a counter example, consider the composition `length xs = sum . map (\x. 1) xs`. We could imagine deriving a hylomorphism for the `map` in which $\phi_2 = \lambda(a, as).\ Cons(1, as)$ or even $\phi_2 = \lambda(as).\ Cons(1, as)$. If we identify the constant as a subterm to be abstracted, we can recover the anamorphic aspect of the hylomorphism. Further, we should not only consider constructors with no arguments, but all constructors with no recursive arguments. For example, the *in* form for a hylomorphism over a tree data type with data in the leaves might be $(\lambda(a).\ Leaf(a)) \triangledown (\lambda(l, r).\ Node(l, r))$. The restructuring algorithm as given would attempt to move $Leaf(a)$ into $\eta$, which is a perfectly legal transformation, but destroys the *in* form that we would like to preserve.

It seems that what we want to do is not remove constructors from the top level. That is, if the body of a $\phi_i$ is a constructor, we should not abstract away the entire body. In our implementation, we do allow total abstraction of the arguments to a top-level constructor. This is acceptable because our implementation does not deal with recursive structures formed from nested constructors (see Section 3.3). This is a bit of a hack, though. What we really want to do is recognize when the top-level constructor is of the type over which the hylomorphism recurses. When we do not allow nested constructors, type correctness guarantees us that our hack will work. However, if our implementation needed to look for a larger structure of constructors, it would need some way of knowing which constructors were part of the recursive structure and which ones could be abstracted away.

Another possible solution to over-eager restructuring is to say that the problem is not with algorithm $\mathcal{E}$ but in our method for identifying whether a $\phi$ ($\psi$) is in *in* (*out*) form. The definitions for *in* and *out* aren't that flexible, though. Perhaps what we really mean to say is that we can let the restructuring algorithm run rampant if we add some better checks to the fusion side. If a function fails the check for the *in* and *out* forms, we can then run an algorithm to check if the function is in a form which can be easily converted to that form. In a sense, we already have such an algorithm in the algorithms which derive $\tau$ and $\sigma$, but it would be inefficient to derive $\tau$ or $\sigma$ when an *in* or *out* form exists, and it would reduce the number of cases where fusion is possible.

Restricting the amount of abstraction performed by the restructuring algorithm is easy enough, so that is what we have done in the pH compiler. Our algorithm takes a boolean input $p$ which indicates whether the expression being examined is at the top-level. The case for constructors uses this value to determine whether to abstract the entire expression.

There are additional differences between our algorithm and the one given by Onoue et al. that are worth mentioning. The original algorithm $\mathcal{E}$ did not live up to the verbal explanation given above. That is, it failed to properly compute the subterms to be abstracted by allowing subterms with bound variables to be marked for abstraction. This required a non-trivial fix to the algorithm, because Onoue et al. used the returned list of subterms to signal whether any recursive variables were found in that subterm. In many cases, their algorithm $\mathcal{E}$ reasons that an expression can be abstracted if and only if all of its subterms can be abstracted. This is not the case with the `\x.x+1` example given above. We do not want to abstract out `x+1`, but when we consider the entire expression then it no longer has any free variables and we can abstract the whole expression out of $\phi$. To relieve the subterm list of its double purpose, a boolean output was added to indicate whether any recursive variables were found. An additional input, $s_b$, is also needed to list the bound variables which cannot occur in a subterm to be abstracted.

The algorithm we use is given in Figure 3-2. This revised algorithm takes as input an expression, a boolean indicating whether the expression is top-level, a list of the recursive variables, and a list of the bound variables. It returns a triple containing a list of associations of fresh variables to the subexpressions that they replace, the original expression with the subexpressions replaced by the new variables, and a boolean indicating whether the expression contains any references to the recursive variables.

The remaining differences between this description and the one given by Onoue et al. result from the algorithms operating on different languages. For one, the pH intermediate syntax does not contain a special representation for hylomorphisms, so our implementation does not contain a case for them. It would be possible for our compiler to put a marker around inlined hylomorphisms and thus have a special case for them in our algorithms. However, we currently make the assumption that the algorithms are correct whether the nested hylomorphisms are inlined or not. This may not be so safe of an assumption when we also consider that Onoue et al. do not have cases in their algorithm for recursive let-statements, prefering instead to allow recursion in their core syntax only in top-level definitions. Our implementation handles recursive let-statements, potentially with mutually recursive definitions. We

$$
\begin{array}{rcl}
\mathcal{E}[\![v]\!]\ \_\ s_r\ s_b & = & \text{if } v \in s_r \text{ then } (\{\}, v, \texttt{False}) \\
& & \qquad\qquad \text{else if } v \in s_b \text{ then } (\{\}, v, \texttt{True}) \\
& & \qquad\qquad\qquad\qquad \text{else } (\{(u, v)\}, u, \texttt{True})
\end{array}
$$

$$
\mathcal{E}[\![l]\!]\ \_\ s_r\ s_b \;=\; (\{(u, l)\}, u, \texttt{True})
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![\lambda v.t]\!]\ \_\ s_r\ s_b & = & \text{if } b \text{ then } (\{(u, \lambda v.t)\}, u, \texttt{True}) \\
& & \quad \text{else } (w, \lambda v.t', \texttt{False}) \\
& & \text{where } (w, t', b) = \mathcal{E}[\![t]\!]\ \texttt{False}\ s_r\ s_b \cup \{v\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{case}\ t_0\ \texttt{of}\ p_1 \rightarrow t_1; \cdots; p_n \rightarrow t_n]\!]\ \_\ s_r\ s_b = \\
\qquad \text{if } \forall i.\ b_i \text{ then } (\{(u, \texttt{case}\ t_0\ \texttt{of}\ p_1 \rightarrow t_1; \cdots; p_n \rightarrow t_n)\}, u, \texttt{True}) \\
\qquad\qquad \text{else } (w_0 \cup \ldots \cup w_n, \texttt{case}\ t'_0\ \texttt{of}\ p_1 \rightarrow t'_1; \cdots; p_n \rightarrow t'_n, \texttt{False}) \\
\qquad \text{where } (w_0, t'_0, b_0) = \mathcal{E}[\![t_0]\!]\ \texttt{False}\ s_r\ s_b \\
\qquad\qquad\quad (w_i, t'_i, b_i) = \mathcal{E}[\![t_i]\!]\ \texttt{False}\ s_r\ s_b \cup \mathit{Vars}(p_i)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![v\ t_1\ \ldots\ t_n]\!]\ \_\ s_r\ s_b = \\
\qquad \text{if } \forall i.\ b_i \text{ then } (\{(u, v\ t_1\ \ldots\ t_n)\}, u, \texttt{True}) \\
\qquad\qquad \text{else } (w_0 \cup \ldots \cup w_n, t'_0\ t'_1\ \ldots\ t'_n, \texttt{False}) \\
\qquad \text{where } (w_0, t'_0, b_0) = \mathcal{E}[\![v]\!]\ \texttt{False}\ s_r\ s_b \\
\qquad\qquad\quad (w_i, t'_i, b_i) = \mathcal{E}[\![t_i]\!]\ \texttt{False}\ s_r\ s_b
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![C\ t_1\ \ldots\ t_n]\!]\ p\ s_r\ s_b = \\
\qquad \text{if } \neg p \wedge \forall i.\ b_i \text{ then } (\{(u, C\ t_1\ \ldots\ t_n)\}, u, \texttt{True}) \\
\qquad\qquad\quad \text{else } (w_1 \cup \ldots \cup w_n, C\ t'_1\ \ldots\ t'_n, \texttt{False}) \\
\qquad \text{where } (w_i, t'_i, b_i) = \mathcal{E}[\![t_i]\!]\ \texttt{False}\ s_r\ s_b
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{let}\ v = t_1\ \texttt{in}\ t_0]\!]\ \_\ s_r\ s_b = \\
\qquad \text{if } b_0 \wedge b_1 \text{ then } (\{(u, \texttt{let}\ v = t_1\ \texttt{in}\ t_0)\}, u, \texttt{True}) \\
\qquad\qquad \text{else } (w_0 \cup w_1, \texttt{let}\ v = t'_1\ \texttt{in}\ t'_0, \texttt{False}) \\
\qquad \text{where } (w_0, t'_0, b_0) = \mathcal{E}[\![t_0]\!]\ \texttt{False}\ s_r\ s_b \cup \{v\} \\
\qquad\qquad\quad (w_1, t'_1, b_1) = \mathcal{E}[\![t_1]\!]\ \texttt{False}\ s_r\ s_b
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{letrec}\ v_1 = t_1; \cdots; v_n = t_n\ \texttt{in}\ t_0]\!]\ \_\ s_r\ s_b = \\
\qquad \text{if } \forall i.\ b_i \text{ then } (\{(u, \texttt{letrec}\ v_1 = t_1; \cdots; v_n = t_n\ \texttt{in}\ t_0)\}, u, \texttt{True}) \\
\qquad\qquad \text{else } (w_0 \cup \ldots \cup w_n, \texttt{letrec}\ v_1 = t'_1; \cdots; v_n = t'_n\ \texttt{in}\ t'_0, \texttt{False}) \\
\qquad \text{where } (w_0, t'_0, b_0) = \mathcal{E}[\![t_0]\!]\ \texttt{False}\ s_r\ s_b \cup \{v_1, \ldots, v_n\} \\
\qquad\qquad\quad (w_i, t'_i, b_i) = \mathcal{E}[\![t_i]\!]\ \texttt{False}\ s_r\ s_b
\end{array}
$$

$$
\begin{array}{rcl}
u & = & \text{fresh variable} \\
\mathit{Vars}(p) & = & \text{the variables of a pattern}
\end{array}
$$

Figure 3-2: Revised Algorithm $\mathcal{E}$

assume our extensions to be correct.

### 3.5.2 Restructuring $\psi$

The restructuring algorithm for $\psi$ should take functions of the form:

$$\psi = \lambda v_s.\ \text{case } t_0 \text{ of } p_1 \to (1, t_{\psi_1}); \cdots; p_n \to (n, t_{\psi_n})$$

and extract any natural transformations which can be shifted into the $\eta$ portion of the hylomorphism. The restructuring algorithm $\mathcal{S}_\psi$ given by Onoue et al. uses an auxiliary algorithm $\mathcal{FV}$ to determine the free variables in each $t_{\psi_i}$ which are not recursive variables (and not globally available to all parts of the hylomorphism). The algorithm then creates a new term $t'_{\psi_i}$ which is a tuple of those free variables and the recursive variables. The original expressions in $t_{\psi_i}$ are substituted into $\eta$. In this way, the new $t'_{\psi_i}$ just passes along the free variables needed to compute whatever more complicated expressions appeared originally, while the computation is done in $\eta$. Before restructuring, $\psi$ contained $t_{\psi_i}$s which were tuples of arbitrary expressions. After restructuring, the new $\psi'$ contains $t'_{\psi_i}$s which are tuples whose non-recursive elements (corresponding to $\Gamma$ in the functor) are only variables. If the recursive elements as well are only variables, then $\psi'$ has the potential for being in *out* form.

The algorithm $\mathcal{S}_\psi$ suffers from the same over-eagerness problem as $\mathcal{S}_\phi$: it does not return $\psi'$ in *out* form in some cases when it exists. What happens is that $t_{\psi_i}$ may not use all the variables available to it from the patterm $p_i$ so the restructured $t'_{\psi_i}$ could be missing some of the non-recursive variables which we require for $\psi$ to be in *out* form. We are free to add these variables into $t'_{\psi_i}$ but we have to be sure that we are not adding recursive variables, which could also prevent $\psi$ from being in *out* form. As the algorithm for detecting the *out* form is not clearly defined, it is possible that our problem lies with the detection algorithm. As with $\mathcal{S}_\phi$, we could loosen our definition of the *out* form, making the output of the restructuring algorithm more acceptable. We could even have the detection algorithm perform some transformations on the less-optimal cases to put them into a form suitable for fusing. We have chosen to place the burden on our restructuring algorithm.

The restructuring algorithm implemented in the pH compiler uses an auxiliary algorithm $\mathcal{FV}$ to collect the free variables. Note, however, that the free variables of $t_{\psi_i}$ which are not already available to $\eta$ are only the variables in the pattern $p_i$ (and that proper *out* form uses all of them), so it might be possible to derive the *out* form without looking for the free variables which are actually used. When it can be determined that the *out* form is not possible, though, it is best to resort to the free-variables method rather than clutter up the function with unnecessary variables. So once our implementation has retrieved the free (non-global) variables from the non-recursive elements of $t_{\psi_i}$, it examines the recursive elements to make sure they are all variables. If any of them are complex expressions, then we know that $\psi$ cannot be restructured into *out* form so we follow the method of Onoue et al. and use only the free variables in the output. If the recursive elements are all variables, then we perform a second test to see if *out* form is possible. We compare the variables appearing in the recursive parts of the tuple against the free variables list. If any variables are shared between them, then the $\psi$ cannot be restructured into *out* form, so we resort to using only the free variables. If the lists are disjoint, then we attempt to create the *out* form by adding back any pattern variables which do not appear in either list. These variables are added to the non-recursive part of the tuple.

If the algorithm reaches this final stage where all the variables are used, the output is not guaranteed to be in *out* form because the recursive variables could appear multiple times in the tuple. What we hope is that the algorithm will produce an output in *out* form if it exists. This turns out not to be the case for this algorithm, and we show a fix for this in Section 3.8. Further, the functor for which the output is in *out* form might not be the same as the functor of the hylomorphism with which we hope to fuse it. We deal with these issues later when we present the detection and fusion algorithms.

As with the derivation algorithm, the description given by Onoue et al. for this restructuring algorithm also contains a reference to *global_vars*. When the algorithm $\mathcal{FV}$ encounters a variable, it considers it to be a free variable of interest if it is not an element of *global_vars*. Perhaps Onoue et al. are only considering hylomorphisms in top-level definitions or they have their algorithms collect the context for each hylomorphism. Our implementation, however, does not attempt to construct this list. Instead, we pass the auxiliary algorithm a list containing the variables from the pattern plus $v_s$ and we consider a variable to be a free variable of interest if it is contained in this list.

The algorithm used in the pH compiler is given formally in Figure 3-3. The auxiliary algorithm $\mathcal{FV}$ takes an expression and a list of potential free variables and returns a list of those variables which occur in the expression.

In the algorithm given in the figure, we intersect $s_f$ with the list containing $v_s$ and the recursive variables. This is to check whether the non-recursive part of the tuple makes reference to any recursive variables. We include $v_s$ because it can be replaced by a construction involving all of the pattern variables. If any of these variables are recursive, then an occurrence of $v_s$ means that the expression uses recursive variables. What the figure does not show is that in disjuncts where there are no recursive variables, an occurrence of $v_s$ is acceptible and this check should not be performed. In those cases, we can produce the *out* form by making all of the pattern variables non-recursive inputs and replacing occurrences of $v_s$ with a construction of these variables.

## 3.6 Fusing Hylomorphisms

Given the composition of two hylomorphisms, $[\![\phi_1, \eta_1, \psi_1]\!] \circ [\![\phi_2, \eta_2, \psi_2]\!]$, we want to determine whether fusion is possible and if so, apply the appropriate fusion rule to produce a single hylomorphism.

There are three forms that we attempt to fuse:

1. Cata-Ana Fusion

$$[\![\phi_1, \eta_1, out_F]\!]_{G,F} \circ [\![in_F, \eta_2, \psi_2]\!]_{F,H}$$

   This is just a special case of the two following rules, but it is easy to detect in the process of detecting the other forms and the fusion law for this case is much

$$
\begin{array}{l}
\mathcal{S}_\psi[\![ [\![\phi, \eta_1 + \cdots + \eta_n, \psi]\!]_{G,F} ]\!] = [\![\phi, \eta'_1 + \cdots + \eta'_n, \psi']\!]_{G,F'} \\
\text{where } \lambda v_s.\ \text{case } t_0 \text{ of } p_1 \rightarrow (1, t_1); \cdots; p_n \rightarrow (n, t_n) = \psi \\
\qquad (t_{i_1}, \ldots, t_{i_{k_i}}, tt_{i_1}, \ldots, tt_{i_{l_i}}) = t_i \\
\qquad F_1 + \cdots + F_n = F \\
\qquad \Gamma(t_{i_1}) \times \cdots \times \Gamma(t_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} = F_i \\
\qquad \{v_{i_1}, \ldots, v_{i_{m_i}}\} = \text{if } \mathit{IsVar}(tt_{i_1}) \wedge \cdots \wedge \mathit{IsVar}(tt_{i_{l_i}}) \\
\qquad\qquad\qquad\qquad \text{then if } s_f \cap \{v_s, tt_{i_1}, \ldots, tt_{i_{l_1}}\} = \{\} \\
\qquad\qquad\qquad\qquad\qquad \text{then } \mathit{Vars}(p_i) - \{tt_{i_1}, \ldots, tt_{i_{l_1}}\} \\
\qquad\qquad\qquad\qquad\qquad \text{else } s_f \\
\qquad\qquad\qquad\qquad \text{else } s_f \\
\qquad\qquad\qquad\qquad \text{where } s_f = \cup_{j=1}^{k_i} (\mathcal{FV}[\![t_{i_j}]\!]\ \mathit{Vars}(p_i) \cup \{v_s\}) \\
\qquad \eta'_i = \lambda(u_{i_1}, \ldots, u_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{m_i}}).\ e_i[s] \\
\qquad t'_i = (t_{i_1}, \ldots, t_{i_{k_i}}, tt'_{i_1}, \ldots, tt'_{i_{m_i}}) \\
\qquad \psi' = \lambda v_s.\ \text{case } t_0 \text{ of } p_1 \rightarrow (1, t'_1); \cdots; p_n \rightarrow (n, t'_n) \\
\qquad F'_i = \Gamma(t_{i_1}) \times \cdots \times \Gamma(t_{i_{k_i}}) \times I_1 \times \cdots \times I_{m_i} \\
\qquad F' = F'_1 + \cdots + F'_n
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{FV}[\![v]\!]\ s_p & = & \text{if } v \in s_p \text{ then } \{\text{v}\} \text{ else } \{\} \\[1em]
\mathcal{FV}[\![l]\!]\ s_p & = & \{\} \\[1em]
\mathcal{FV}[\![\lambda v.t]\!]\ s_p & = & \mathcal{FV}[\![t]\!]\ s_p \\[1em]
\mathcal{FV}[\![\texttt{case}\ t_0\ \texttt{of}\ p_1 \rightarrow t_1; \cdots; p_n \rightarrow t_n]\!]\ s_p & = & s_0 \cup \cdots \cup s_n \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\ s_p \\[1em]
\mathcal{FV}[\![v\ t_1\ \ldots\ t_n]\!]\ s_p & = & s_0 \cup \cdots \cup s_n \\
 & & \text{where } s_0 = \mathcal{FV}[\![v]\!]\ s_p \\
 & & \qquad\quad s_i = \mathcal{FV}[\![t_i]\!]\ s_p \\[1em]
\mathcal{FV}[\![C\ t_1\ \ldots\ t_n]\!]\ s_p & = & s_1 \cup \cdots \cup s_n \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\ s_p \\[1em]
\mathcal{FV}[\![\texttt{let}\ v = t_1\ \texttt{in}\ t_0]\!]\ s_p & = & s_0 \cup s_1 \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\ s_p \\[1em]
\mathcal{FV}[\![\texttt{letrec}\ v_1 = t_1; \cdots; v_n = t_n\ \texttt{in}\ t_0]\!]\ s_p & = & s_0 \cup \cdots \cup s_n \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\ s_p
\end{array}
$$

Figure 3-3: Revised Algorithm $\mathcal{S}_\psi$

simpler. For deforestation of lists, the case we are most concerned with, this is the only fusion rule that is needed for reasonable inputs. (Only functions which recursively generate the tail *and* the head need to have $\tau$ derived to be fused.)

2. Cata-Hylo Fusion

$$[\![\phi_1, \eta_1, out_F]\!]_{G,F} \circ [\![\tau\ in_F, \eta_2, \psi_2]\!]_{F',H}$$

   This form can be fused with the Cata-HyloFusion rule from Section 2.3.6.

3. Hylo-Ana Fusion

$$[\![\phi_1, \eta_1, \sigma\ out_F]\!]_{G,F'} \circ [\![in_F, \eta_2, \psi_2]\!]_{F,H}$$

   This form can be fused with the Hylo-AnaFusion rule from Section 2.3.6.

To determine whether a composition is in one of these forms, we first restructure the hylomorphisms by applying $\mathcal{S}_\psi$ to the first hylomorphism and $\mathcal{S}_\phi$ to the second hylomorphism to produce $[\![\phi_1, \eta'_1, \psi'_1]\!] \circ [\![\phi'_2, \eta'_2, \psi_2]\!]$. Then we test whether $\psi'_1$ is in *out* form and $\phi'_2$ in *in* form. If so, then we further test that they are *in* and *out* for the same functor! (This is explained further in Section 3.6.3.) If the functors match, then we can apply the Cata-Ana fusion rule. If not, it might still be possible to derive $\tau$ or $\sigma$ and proceed with the other fusion rules. If only one of $\phi'_1$ or $\psi'_2$ is in the proper form, then we attempt to derive $\tau$ or $\sigma$ from the other to produce a deforestable composition. If neither $\psi'_1$ nor $\phi'_2$ is in the proper form, or if a $\tau$ or $\sigma$ cannot be derived, then our implementation gives up.

### 3.6.1 Detecting $in_F$

Our implementation considers a function $\phi$ to be in *in* form if all of the $t_{\phi_i}$s are constructors whose arguments, if any, are all variables. The order of the variables is not considered, because the ordering can be affected through natural transformations which can be moved into $\eta$ to produce the order which best suits us (meaning, the order which matches the $\psi$ of the hylomorphism with which we are attempting to fuse). What is important is which variables are recursive and which are non-recursive. This is something we do not check for in this stage, because, regardless, the function is in *in* form. The placement of non-recursive and recursive variables changes the functor for which it is the *in* function. If this functor, call it $F$ does not match the functor $F'$ of the hylomorphism is it composed with, then we might need to derive a $\tau$ for which $in_F = \tau\ in_{F'}$.

For the list data structure,

$$in = (\lambda().\ Nil) \triangledown (\lambda(u, v).\ Cons(u, v))$$

Detecting this *in* form is deceptively easy, because the first constructor has no arguments and the second constructor has one non-recursive argument and one recursive argument, which can't easily be mixed up: it can only have two recursive elements in crazy functions which recurse to produce the head and it can't have only non-recursive variables because then it wouldn't be a recursive function! Hylomorphisms which operate on more complicated structures, however, like trees, whose constructors contain multiple recursive variables, can have $\phi$ in *in* form, but over a different functor, not the proper functor for the data type, depending on the distribution of non-recursive and recursive variables. In these cases, it becomes important to compare the $\phi$ from one hylomorphism with the $\psi$ from the other hylomorphism to see if simple reordering of the variables will reconcile the functors or if it is necessary to derive $\tau$ or $\sigma$.

As an example of when two composed hylomorphisms might be in *in* and *out* form, but for different functors, consider the following functions:

```
data Tree t = Leaf | Branch t (Tree t) (Tree t)

mapTree f Leaf = Leaf
mapTree f (Branch n l r) = Branch (f n) (mapTree f l) (mapTree f r)

mapLeft f Leaf = Leaf
mapLeft f (Branch n l r) = Branch (f n) (mapLeft l) r

makeTree [] = Leaf
```

```
makeTree (x:xs) = Branch x (Leaf) (makeTree xs)
```

The derived hylomorphisms for these functions are

$$\begin{aligned}
&\texttt{mapTree} = \lambda f.\ [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).\mathit{Leaf} \ \triangledown\ \lambda((u_1),(v_1,v_2)).\mathit{Branch}((f\ \ u_1), v_1, v_2) \\
&\qquad\qquad \psi = \lambda v_s.\texttt{case}\ \ v_s\ \ \texttt{of} \\
&\qquad\qquad\qquad\qquad \mathit{Leaf} \to (1,((),())) \\
&\qquad\qquad\qquad\qquad \mathit{Branch}(n,l,r) \to (2,((n),(l,r)))
\end{aligned}$$

$$\begin{aligned}
&\texttt{mapLeft} = \lambda f.\ [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).\mathit{Leaf} \ \triangledown\ \lambda((u_1,u_2),(v_1)).\mathit{Branch}((f\ \ u_1), v_1, u_2) \\
&\qquad\qquad \psi = \lambda v_s.\texttt{case}\ \ v_s\ \ \texttt{of} \\
&\qquad\qquad\qquad\qquad \mathit{Leaf} \to (1,((),())) \\
&\qquad\qquad\qquad\qquad \mathit{Branch}(n,l,r) \to (2,((n,r),(l)))
\end{aligned}$$

$$\begin{aligned}
&\texttt{makeTree} = [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).\mathit{Leaf} \ \triangledown\ \lambda((u_1),(v_1)).\mathit{Branch}(u_1, \mathit{Leaf}, v_1) \\
&\qquad\qquad \psi = \lambda v_s.\texttt{case}\ \ v_s\ \ \texttt{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nil} \to (1,((),())) \\
&\qquad\qquad\qquad\qquad \mathit{Cons}(x,xs) \to (2,((x),(xs)))
\end{aligned}$$

It should be clear that `mapTree` and `mapLeft` each have a $\phi$ which is in *in* form, but for different functors, because they have different numbers of recursive variables in the `Branch` case. The first function recurses over the last two arguments to the constructor, while the second function recurses only over the middle argument. Similarly, the derived hylomorphism for `makeTree` has only one recursive variable and therefore cannot be directly fused with either of the previous functions, whose $\psi$ are in *out* form but have two recursive variables. That is, `makeTree` (after restructuring) has the functor $F_2 = \Gamma(\texttt{n}) + \Gamma(\texttt{Leaf}) + I$, which does not match the functor from `mapTree`, $F_2' = \Gamma(\texttt{n}) + I + I$. It would be necessary to derive $\tau$ in order to deforest the composition.

### 3.6.2 Detecting $out_F$

Our implementation considers a function $\psi$ to be in *out* form if every $t_{\psi_i}$ is a tuple of the variables in the associated $p_i$, with each pattern variable appearing exactly once. As with detecting *in*, the order of the variables is unimportant (although we can enforce an arbitrary order here and in the restructuring algorithms), rather the important information is which pattern variables are in the recursive part of the tuple and which are in the non-recursive part.

### 3.6.3 Reconciling $in_F$ and $out_F$

As we have mentioned, once the hylomorphisms in a composition have been identified as catamorphism and anamorphism, it is still necessary to determine whether they are acting over the same functor. This can be done rather easily in our implementation by matching the constructor in each $t_{\phi_i}$ with a pattern $p_i$ and associated $t_{\psi_i}$ from $\psi$, checking first, of course, that the number of cases $n$ is the same in both hylomorphisms! If the cases from $\phi$ and $\psi$ can be matched up one-to-one, then it is necessary to check that the variables in each case match up. This means that if the $j$-th variable in the constructor from $t_{\phi_i}$ is supplied by the $k$-th element of the input to $\phi_i$, then the $j$-th variable of the same constructor in $p_i$ must be the $k$-th element in the output $t_{\psi_i}$. Further, that variable should be either recursive in both places or non-recursive in both places. If in one instance it is a non-recursive variable and in the other it is a recursive variable, then the hylomorphisms are not operating on the same functor.

We can guarantee that the variables will be in the same order by enforcing an order in the output of the restructuring algorithms. It is easy enough to have these algorithms return hylomorphisms whose inputs to each $\phi_i$ and outputs in each $t_{\psi_i}$ are variables given in the order they appear in the constructors, but grouped into non-recursive and recursive, of course. What we are really concerned with, then, in the reconciling algorithm, is the distribution of variables into these two groups, a property which is intrinsic to the hylomorphisms.

It is worth noting that given a constructor, the pH compiler can retrieve the entire set of constructors for that data type and can determine the type signature for each constructor. Thus, it is possible to tell which arguments to the constructors are of the same type as the output of the constructors. From this it is possible to generate the true *in* and *out* for that data type, and to detect whether a given $\phi$ or $\psi$ is in that form. As we have seen, however, the functor of a $\phi$ or $\psi$ is only important in relation to the $\phi$ or $\psi$ to which it is being fused. There are, in fact, cases where two hylomorphisms are not in *in* and *out* form based on the functor $F$ for the intermediate data type, but there exists a related functor $F'$ for which one hylomorphism is in $in_{F'}$ ($out_{F'}$) form and the other is in $\sigma\ out_{F'}$ ($\tau\ in_{F'}$) form. In such cases, fusion can proceed through the functor $F'$, when it could not have proceeded had we required that everything be expressed in terms of $F$.

### 3.6.4 Fusion

If the cases of the hylomorphisms can be matched and the variables agree in all cases, then it simply remains to reorder the cases of one hylomorphism to match the other before fusion can be performed. Fusion itself is simple composition of functions to produce a single hylomorphism.

Since we have appended additional information to the hylomorphism representation, there is one new step to the fusion process, which is to combine the contexts from the composed hylomorphisms to become the context for the hylomorphism resulting from fusion. Except for $\lambda$-abstraction, the statements we have allowed in the context have had the property that we can push surrounding code inside the expression without worry. Thus, to fuse composed hylomorphisms, we simply push the outermost hylomorphism inside the context of the first, and perform the fusion there. This is possible only if the inner hylomorphism is fully applied, and thus does not have any surrounding $\lambda$-abstractions. The context for the resulting hylomorphism is then just the concatenation of the context lists of the composed hylomorphisms.

### 3.6.5 Finding Compositions of Hylomorphisms

As mentioned in Section 2.3.6, the current Acid Rain rules relieve the requirements that overlapping compositions be performed in a particular order. This suggests that any reduction strategy is acceptable, which is nearly true. There are still a few issues to think about, though.

For one, we will show in Section 3.7.2 that the algorithm for deriving $\sigma$ might remove the possibility of later fusion, and if so should be performed last. This is easy to say, but it also means that we need to be careful about the order that we fuse inside the elements of a let-statement. We may want to fuse the definitions of functions before fusing instances of them inside the body of the let-statement so as to avoid duplicating work if we need to fuse multiple instances inside the body. But if the definition consists of multiple composed hylomorphisms whose outermost hylomorphism will require deriving $\sigma$, then fusing the entire definition first might produce a hylomorphism that cannot be later fused with other hylomorphisms to which it might be applied.

Another issue arises if we try to reduce applications of hylomorphisms to constructors while we are performing fusion reductions. That is, if we encounter a recursive list function being applied to the list `[1,2,3,4]`, then we can find the case in the hylomorphism for `(x:xs)` and apply it to `1` and `[2,3,4]`. Since it is a recursive function, the result of the application is likely to contain another application of the hylomorphism—in this case, to the tail of the list—which we can continue to reduce. The problem arises if there is a composition of hylomorphisms being applied to the constructor. While we could reduce the applications one at a time, it is likely to be more efficient if we fuse all the hylomorphisms first and perform a single reduction on that hylomorphism applied to the constructor. The first step of this reduction, as we have said, is likely to produce applications of the hylomorphism on the recursive arguments of the constructor. It might also have introduced new fusible compositions as well which, by the same reasoning, should be performed before continuing to unroll the hylomorphism on the constant data structure.

Our reduction algorithm does not provide any special insight, but we present it here for completeness. Our algorithm makes two passes. First, it derives hylomorphisms for all of the recursive functions in the program, producing a table which associates the names of the functions with their hylomorphism representations. This table is then used by a second pass which finds fusible compositions and performs the fusion.

This second pass recurses to the deepest parts of the program and works its way out. Each recursive call returns whether the subexpression it has processed so far is a hylomorphism, allowing the caller to decide if the larger expression is a fusible composition of hylomorphisms and, if so, to return the fused hylomorphism. The table of hylomorphisms is necessary so that when a variable is encountered, it can be looked up in the table to see if it is bound to a hylomorphism. This way, a composition of variables which are bound to hylomorphisms can be fused without having to inline every hylomorphism. If the hylomorphisms bound to variables in a composition cannot be fused, they are not inlined.

The algorithm for finding and performing fusion is given in Figures 3-4 and 3-5. It examines an expression and returns the deforested expression plus a signal indicating whether the expression represents a hylomorphism and what that hylomorphism is. This way, when the algorithm is operating on an application of a hylomorphism, it can call itself recursively on the last argument to see if that argument is a hylomorphism. If so, then we have a composition of hylomorphisms which we should attempt to fuse. If fusion is successful, then we inline the result and return that as the new expression, while at the same time returning the hylomorphism representation and a signal that the expression being returned is a hylomorphism. The signal also becomes useful in the let-statement, $\lambda$-abstraction, and case-statement cases where, if the body is a hylomorphism, then the entire expression is that hylomorphism with its context adjusted to reflect the additional statement. As mentioned, this algorithm requires a hylomorphism table in order to determine which identifiers represent hylomorphisms. As we see in the case for recursive let-statements, if recursively applying the algorithm to the definition of a variable results in a hylomorphism, then the hylomorphism table

$$\mathcal{J}[\![v]\!]\ ht \;=\; \text{if } (v,h) \in ht \text{ then } (\texttt{Just}\ \ h, v) \text{ else } (\texttt{Nothing}, v)$$

$$\mathcal{J}[\![l]\!]\ ht \;=\; (\texttt{Nothing}, l)$$

$$\begin{array}{rcl}
\mathcal{J}[\![\lambda v.t]\!]\ ht & = & \text{case } h \text{ of} \\
& & \quad \texttt{Just}\ h' \to \text{if } (\mathit{GetArity}\ h') > \mathit{0} \\
& & \qquad\qquad\quad \text{then } (\texttt{Just}\ \ (\mathit{AddLambda}\ \ h'\ \ v), \lambda v.t') \\
& & \qquad\qquad\quad \text{else } (\texttt{Nothing}, \lambda v.t') \\
& & \quad \texttt{Nothing} \to (\texttt{Nothing}, \lambda v.t') \\
& & \text{where } (h, t') = \mathcal{J}[\![t]\!]\ ht
\end{array}$$

$$\begin{array}{l}
\mathcal{J}[\![\texttt{case}\ \ t_0\ \ \texttt{of}\ \ p_1 \to t_1; \cdots; p_n \to t_n]\!]\ ht = \\
\qquad \text{case } \mathit{SingleBranch}(\texttt{case}\ \ t'_0\ \ \texttt{of}\ \ p_1 \to t_1; \cdots; p_n \to t_n) \text{ of} \\
\qquad\quad \texttt{Nothing} \to (\texttt{Nothing}, \texttt{case}\ \ t'_0\ \ \texttt{of}\ \ p_1 \to t'_1; \cdots; p_n \to t'_n) \\
\qquad\quad \texttt{Just}\ (\mathit{context}, e) \to \\
\qquad\qquad \text{case } h \text{ of} \\
\qquad\qquad\quad \texttt{Nothing} \to (\texttt{Nothing}, \texttt{case}\ \ t'_0\ \ \texttt{of}\ \ p_1 \to t'_1; \cdots; p_n \to t'_n) \\
\qquad\qquad\quad \texttt{Just}\ \ h' \to (\texttt{Just}\ \ h'', \texttt{case}\ \ t'_0\ \ \texttt{of}\ \ p_1 \to t'_1; \cdots; p_n \to t'_n) \\
\qquad \text{where } (_, t'_i) = \mathcal{J}[\![t_i]\!]\ ht \\
\qquad\qquad\quad (h, e') = \mathcal{J}[\![e]\!]\ ht \\
\qquad\qquad\quad h'' = \mathit{AddCase}\ \ h'\ \ \mathit{context}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{J}[\![v\ t_1\ \ldots\ t_n]\!]\ ht & = & \text{if } (v,h) \in ht \\
& & \text{then if } (l > n) \\
& & \qquad \text{then } (\texttt{Just}\ \ h', v\ t'_1\ \ldots\ t'_n) \\
& & \qquad \text{else if } (n = l) \\
& & \qquad\quad \text{then case } h_n \text{ of} \\
& & \qquad\qquad\quad \texttt{Nothing} \to (\texttt{Just}\ \ h', v\ t'_1\ \ldots\ t'_n) \\
& & \qquad\qquad\quad \texttt{Just}\ \ h'_n \to \\
& & \qquad\qquad\qquad \text{case } h_{f_1} \text{ of} \\
& & \qquad\qquad\qquad\quad \texttt{Nothing} \to (\texttt{Just}\ \ h', v\ t'_1\ \ldots\ t'_n) \\
& & \qquad\qquad\qquad\quad \texttt{Just}\ \ h'' \to (\texttt{Just}\ \ h'', \mathit{Inline}\ \ h'') \\
& & \qquad\quad \text{else case } h_l \text{ of} \\
& & \qquad\qquad\quad \texttt{Nothing} \to (\texttt{Nothing}, v\ t'_1\ \ldots\ t'_n) \\
& & \qquad\qquad\quad \texttt{Just}\ \ h'_l \to \\
& & \qquad\qquad\qquad \text{case } h_{f_2} \text{ of} \\
& & \qquad\qquad\qquad\quad \texttt{Nothing} \to (\texttt{Nothing}, v\ t'_1\ \ldots\ t'_n) \\
& & \qquad\qquad\qquad\quad \texttt{Just}\ \ h'' \to \\
& & \qquad\qquad\qquad\qquad (\texttt{Nothing}, (\mathit{Inline}\ \ h'')\ t'_{l+1}\ \ldots\ t'_n) \\
& & \text{else } (\texttt{Nothing}, v\ t'_1\ \ldots\ t'_n) \\
& & \text{where } (h_i, t'_i) = \mathcal{J}[\![t_i]\!]\ hs \\
& & \qquad\quad l = \mathit{GetArity}\ \ h \\
& & \qquad\quad h' = \mathit{AddValues}\ \ h\ \{t'_1, \ldots, t'_n\} \\
& & \qquad\quad h_{f_1} = \mathit{FuseHylos}\ \ (\mathit{AddValues}\ \ h\ \{t'_1, \ldots, t'_{n-1}\})\ h'_n \\
& & \qquad\quad h_{f_2} = \mathit{FuseHylos}\ \ (\mathit{AddValues}\ \ h\ \{t'_1, \ldots, t'_{l-1}\})\ h'_l
\end{array}$$

Figure 3-4: Finding Fusible Compositions (Part I)

$$
\begin{aligned}
\mathcal{J}[\![C\ t_1\ \ldots\ t_n]\!]\ ht &= (\texttt{Nothing}, C\ t_1'\ \ldots\ t_n') \\
&\quad \text{where } (h_i, t_i') = \mathcal{J}[\![t_i]\!]\ hs
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{J}[\![\texttt{let}\ v = t_1\ \texttt{in}\ t_0]\!]\ ht &= \text{case } h_0 \text{ of} \\
&\qquad \texttt{Nothing} \to (\texttt{Nothing}, \texttt{let}\ v = t_1'\ \texttt{in}\ t_0') \\
&\qquad \texttt{Just}\ h \to (\texttt{Just}\ h', \texttt{let}\ v = t_1'\ \texttt{in}\ t_0') \\
&\quad \text{where } (h_1, t_1') = \mathcal{J}[\![t_1]\!]\ ht \\
&\qquad\qquad (h_0, t_0') = \mathcal{J}[\![t_0]\!]\ ht \cup \{h_1\} \\
&\qquad\qquad h' = \mathit{AddLet}\ h\ \{(v, t_1')\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{J}[\![\texttt{letrec}\ v_1 = t_1; \cdots; v_n = t_n\ \texttt{in}\ t_0]\!]\ ht = \\
&\qquad \text{case } h_0 \text{ of} \\
&\qquad\quad \texttt{Nothing} \to (\texttt{Nothing}, \texttt{letrec}\ v_1 = t_1'; \cdots; v_n = t_n'\ \texttt{in}\ t_0') \\
&\qquad\quad \texttt{Just}\ h \to (\texttt{Just}\ h', \texttt{letrec}\ v_1 = t_1'; \cdots; v_n = t_n'\ \texttt{in}\ t_0') \\
&\qquad \text{where } (h_1, t_1') = \mathcal{J}[\![t_1]\!]\ ht \\
&\qquad\qquad (h_i, t_i') = \mathcal{J}[\![t_i]\!]\ ht \cup \{(v_1, h_1), \ldots, (v_{i-1}, h_{i-1})\} \\
&\qquad\qquad (h_0, t_0') = \mathcal{J}[\![t_0]\!]\ ht \cup \{(v_1, h_1), \ldots, (v_n, h_n)\} \\
&\qquad\qquad h' = \mathit{AddLet}\ h\ \{(v_1, t_1'), \ldots, (v_n, t_n')\}
\end{aligned}
$$

Figure 3-5: Finding Fusible Compositions (Part II)

needs to be updated for the scope of that variable to indicate the new hylomorphism that the variable represents.

A problem with this algorithm is that it requires that an identifier be bound to a hylomorphism in order to be inlined for fusion. If an identifier represents a composition of functions, the last of which is a hylomorphism, then our algorithm will not realize that fusion is possible in cases where the identifier is being applied to a hylomorphism. If pre-processing does not inline all such cases for us, a more clever method for identifying when to inline is needed.

For the same reasons, the case for recursive let-statements may not catch all compositions of hylomorphisms. The table of hylomorphisms is threaded serially through the recursive calls on the bindings so that as a definition is discovered to be a hylomorphism, it can be fused in later definitions. With mutually recursive definitions, it may be necessary to process the definitions several times before all possible fusion has been performed. We currently assume this case to be rare or impossible, and only process the definitions once.

Another issue to note is in the $\lambda$-abstraction case where we check the arity of the hylomorphism before adding to the context. The case for application, where fusion is performed, operates on the assumption that if a hylomorphism has a non-zero arity, then it has not yet been applied to its recursive argument. Thus, to make this assumption correct, we have to check the arity of hylomorphisms in the $\lambda$-abstraction case and not add to the context of a hylomorphism which is fully applied. However, more inlining could be performed if we relaxed this restriction. To do this, we would need to alter the case for applications to not only check the arity of the hylomorphism but also to look for the existence of an `Apply` disjunct in the context. If one does not occur in the context, then we are free to use the arity information to determine the recursive argument.

## 3.7 Deriving $\tau$ and $\sigma$

Sometimes simple restructuring will not produce a catamorphism or anamorphism. In those cases, we can still proceed with fusion if we can show that $\phi$ ($\psi$) is of the form $\tau\, in$ ($\sigma\, out$). That is, if we can show that the computation can be abstracted over the data constructors. This derivation is necessary in cases where computation is being performed on the recursive elements of the data structure. For $\phi$, this can mean that a hylomorphism is being applied to a recursive variable. For $\psi$, it means that further pattern matching is being performed beyond the first level of constructors. Onoue et al. give examples to illustrate when derivation is needed and what the results are.

The algorithm for deriving $\tau$ was straightforward and easy to write. The algorithm for $\sigma$ has not been implemented in our system. It will likely require more tinkering to perform the algorithm in the pH framework because the algorithm as given involves the construction of patterns which is not as easy as the high-level description makes it look. In the same way that our implementation does not explicitly produce functions with separated sums, it is likely that a special representation will be needed for $\sigma$ which captures all the information necessary for applying it to an actual $\psi$, but without creating a real function.

Both algorithms take a $\phi$ ($\psi$) from which to derive and the functor of the $\psi$ ($\phi$) to which we want to fuse. In our implementation for deriving $\tau$, we pass the entire $\psi$ to the algorithm rather than simply extracting the functor. The algorithm for deriving $\tau$ makes use of the *out* form for the functor it has been passed, and because we only attempt to derive when the $\psi$ of the other hylomorphism is in *out* form, by passing the $\psi$ to the algorithm we can have a specimen of this form handy rather than have to create it from the functor.

### 3.7.1 Deriving $\tau$

The algorithm $\mathcal{F}$ given by Onoue et al. for deriving $\tau$ requires that its input be in a canonical form or else it cannot derive $\tau$ for that input. A function $\phi$ is in this canonical form if, for each $\phi_i$, the body $t_i$ is in one of the following forms:

1. a variable bound by $\phi_i$
2. a constructor application in which each of the arguments to the constructor is in one of these four canonical forms
3. an application of a hylomorphism to one of the recursive variables of $\phi_i$, where the $\phi$ of the hylomorphism is in canonical form and the $\psi$ is in *out* form
4. a function application where the function is, according to Onoue et al., a "global function" (meaning that it can be lifted out of the hylomorphism or already is) and where no argument to the function contains a reference to any recursive variable

Onoue et al. claim that this form is not as restrictive as it might look. Restructuring, they say, removes much of the computation not related to the building of the final data type. In fact, after restructuring, expressions of the fourth kind have been extracted from $\phi$ so we do not even need to consider that case in our algorithm. Onoue et al. dismiss let-statements, case-statements, and non-recursive functions as things which can be removed by pre-processing. They suggest that case-statements can be translated into hylomorphisms, although it is not clear whether they are simply observing that most case-statements are removed when we convert recursive functions to hylomorphisms (and remember that some recursive functions just don't fit the form from which we can derive, so some case-statements will remain) or whether they are further suggesting that case-statements not in recursive functions can be converted into non-recursive hylomorphisms! In any event, the proposed algorithm works for a large number of reasonable cases. The question of whether still more cases can be squeezed out of the system is a matter for further research.

In order to derive a $\tau$ satisfying $\phi = \tau \; in_F$, one needs the $\phi$ from which to derive and the functor $F$. We perform the derivation because we are hoping to fuse the hylomorphism containing this $\phi$ with a hylomorphism whose $\psi$ is in $out_F$ form. It is this functor $F$, from $\psi$, that we pass to the deriving algorithm. In fact, rather than deriving the functor, for which as we have said we do not have a representation, we simply pass the entire $\psi$ to the deriving function. From $\psi$ we can produce a list of the constructors in its patterns. Using this list, we can create a set of fresh variables which can be used to abstract out occurrences of those constructors in $\phi$. Further, we can determine which are the recursive and non-recursive arguments to the constructors, which is the information we needed from the functor $F$.

With this information, we straightforwardly implement the algorithm given by Onoue et al. We do take a shortcut, perhaps at the sake of readability, by checking that $\phi$ is in the canonical form at the same time that we are deriving $\tau$. If we find that $\phi$ is not in canonical form, we simply throw away the information we have gathered so far.

It is worth noting that the algorithm $\mathcal{F}$ uses an auxiliary algorithm $\mathcal{F}'$ which operates on the $t_i$ of each $\phi_i$. The auxiliary algorithm as given in [21] relies on information from $\mathcal{F}$ without explicitly mentioning it as arguments to the algorithm. This is acceptable if $\mathcal{F}'$ is defined in the scope of $\mathcal{F}$, but it is worth clarifying what information is being used because it does create a slight ambiguity in the case of $\mathcal{F}'$ on a single variable. The result of the algorithm depends on whether the variable is a recursive variable. Since there can be nested hylomorphisms and thus recursive calls to $\mathcal{F}$, it is unclear which recursive variables should be considered. Should recursive calls to $\mathcal{F}$ build up a list of recursive variables or should it only consider recursive variables from the current hylomorphism? We choose to only consider the current hylomorphism. Changing our system to the other interpretation would be trivial.

The algorithm $\mathcal{F}$ abstracts away constructors leaving a function $\tau$ which takes in the cases of a $\phi$ and places them where the constructors used to be. Inside $\tau$, in order to abstract constructors from structures which will not be built until runtime, it is necessary to use a catamorphism $(\!|c_1 \triangledown \cdots \triangledown c_n|\!) = [\![c_1 \triangledown \cdots \triangledown c_n, id, out]\!]$ which serves to perform the abstraction. In our implementation, we construct this catamorphism once, from $\psi$, and insert it into $\tau$ wherever it is needed. When $\tau$ is applied to a $\phi$, each variable $c_i$ is replaced by the expression $\phi_i$. Thus, the catamorphism becomes $[\![\phi, id, out]\!]$. We could save a little work if we added an additional abstraction variable to $\tau$ when it is derived, and used this new variable in place of the catamorphism. Then, in our function which applies $\tau$ to the $\phi$ from another hylomorphism, we can simply substitute the whole hylomorphism in place of this new variable.

In this algorithm it is important to recognize hylomorphisms to decide whether a $\phi$ is in canonical form. If we find a global variable applied to a recursive variable, we have to know the hylomorphism to which it is bound in order to consider it in canonical form and derive $\tau$. Thus, our implementation passes the hylomorphism table to algorithm $\mathcal{F}$. When a hylomorphism is identified, it is abstracted by recursive application of the algorithm. The result is a new hylomorphism. In the case where the hylomorphism is indicated by a global variable, the new hylomorphism needs to be inlined with a new name. In the case where the hylomorphism is inlined, we simply need to rewrite the hylomorphism, and a new name is not necessary. However, when we apply $\tau$ to a $\phi$ and produce the fused hylomorphism, new internal hylomorphisms will not appear in the hylomorphism table and any hylomorphisms inlined under their original names will now have incorrect entries in the table. Since we rely on the information in this table, particularly if we will need to derive $\tau$ again, the hylomorphism derivation pass should be performed on the fused hylomorphism to correct the entries.

### Restructuring as an aid to deriving $\tau$

Onoue et al. claim that the restructuring algorithm helps simplify the derivation of $\tau$ and $\sigma$. However, it appears that the algorithm for restructuring $\phi$ can actually transform a hylomorphism into a form from which we cannot derive $\tau$. This was discovered when our system could not derive $\tau$ for the following example from Onoue et al.:

```
data Tree = Leaf | Node Int Tree Tree


foo = \xs. case xs of
                 []     -> Leaf
                 (a:as) -> Node (double a) (squareNodes (foo as)) (foo as)
```

where `squareNodes` is a hylomorphism that squares the values in a tree. Our system derives the following $\phi$ for this function:

$$\phi = \lambda().\ Leaf \triangledown \lambda(v, v_1', v_2').\ Node(double\ v, squareNodes\ v_1', v_2')$$

This is in the canonical form and thus our system can apply the algorithm for deriving $\tau$. However, if we restructure the hylomorphism first, as Onoue et al. suggest, we produce a hylomorphism with the following $\phi$:

$$\phi = \lambda().\ Leaf \triangledown \lambda(v_1, v_2, v_1', v_2').\ Node(v_1, v_2\ v_1', v_2')$$

Here, the hylomorphism *squareNodes* has been identified as a subexpression which can be moved into $\eta$. Whether the function is referenced by an identifier or is inlined into the definition of `foo`, it will be replaced with a non-recursive variable by the restructuring algorithm. This leaves the expression $v_2\ v_1'$ which is not in canonical form. The canonical form requires that only hylomorphisms are applied to recursive variables. Since $\phi$ is not in canonical form, we cannot derive $\tau$ and thus cannot perform the fusion that was possible before the restructuring algorithm was applied.

To solve this problem, we can alter the reconstruction algorithm to preserve some computation in $\phi$. In fact, we only need to change the case of algorithm $\mathcal{E}$ for function application. In such cases, we check whether any of the arguments to the function contain recursive variables. If none of the arguments contain any references to the recursive variables, then we are free to abstract the entire application. If any of the arguments contains a recursive variable, then we have to decide how much restructuring we want to perform on the remaining arguments and the applied function. Because work is being done to a recursive variable, we know that the $\phi$ cannot be in *in* form, but we may still want to abstract away as much work as possible for other reasons, so the question is how much can we abstract before we cause problems for the $\tau$ derivation algorithm. It may not be possible to decide exactly how much can be abstracted and how much should be left, so we should err on the side of caution. Or we could decide to have multiple restructuring algorithms depending on the purposes for performing restructuring.

One possibility is that we carry around a table of hylomorphisms and look up the identifier of the function being applied. If the function being applied is a hylomorphism, then do not abstract it. We can even go so far as to consider the arity of the hylomorphism and determine whether the final argument to the hylomorphism contains a recursive variable. If the final argument does not contain a recursive variable, or if the hylomorphism is not being applied to enough arguments, we could choose to abstract it as normal. If we do not abstract the hylomorphism, we still have the option of abstracting any arguments which do not contain recursive variables.

More exploration is needed to decide which options are best. It is important to note, though, that restructuring is also used to bring internal hylomorphisms together for fusion. If our restructuring algorithms are too conservative in how much computation they leave in $\phi$ and $\psi$, we may miss opportunities for fusion inside hylomorphisms. The possibility of using separate restructuring algorithms should not be ignored.

### 3.7.2 Deriving $\sigma$

Onoue et al. present an algorithm $\mathcal{G}$ for deriving $\sigma$ from a $\psi$ of the form:

$$\lambda v_s.\ \texttt{case}\ \ v_s\ \ \texttt{of}\ p_1 \to t_1; \cdots; p_n \to t_n$$

The restriction here is that the case-statement operates on a single variable, the variable of the $\lambda$-abstraction.

The algorithm is intended to operate on hylomorphisms defined by nested case-statements, such as a list-consuming function which has a case for empty lists, a case for single-element lists, and a case for multiple-element lists. This algorithm is the dual of the algorithm for deriving $\tau$ in that it abstracts the deconstruction of the data structure. As Onoue et al. say, the implicit decomposition is made explicit by using the function $out_F$ to perform the decomposition.

The result of the algorithm, however, is a function $\sigma$ which, when applied to a $\psi$, produces a $\psi'$ whose case-statement branches on a complex expression. This new $\psi'$, it would seem, cannot be an input into algorithm $\mathcal{G}$. However, the complex expression involves case-statements nested inside the branching expressions of other case-statements, with the variable $v_s$ of the $\lambda$-abstraction at the core. It is possible that this nesting of case-statements could be transformed into a nested set of patterns branching purely on $v_s$, which would satisfy the form on which algorithm $\mathcal{G}$ operates.

## 3.8 Revising the Restructuring Algorithms

In the hylomorphism representation given in Section 3.2, we assume that the recursive variables in a natural transformation $\eta$ must be exactly those recursive variables in the input. Further, we assume that no recursive variables appear in the non-recursive elements of the output. That is, we assume that a derived hylomorphism starts with an identity $\eta$ which satisfies this form and that all the algorithms we apply to hylomorphisms preserve this feature of $\eta$. Thus, we reasoned, our representation needs only to record the input non-recursive variables, the input recursive variables, and the expressions for the non-recursive part of the output. Since the recursive variables should not appear in the output expressions, we can actually reduce our representation to merely keep a count of how many recursive variables are in the input of $\eta$ without actually giving them names. However, as we are about to show, the recursive output of $\eta$ is actually needed and therefore our representation should keep the named inputs and be extended to include a set of identifiers corresponding to the recursive outputs of $\eta$.

The need for a new $\eta$ became apparent when we attempted to test our system on the function `foo` (described in Section 3.7.1) used by Onoue et al. to illustrate the algorithm for deriving $\tau$. Onoue et al. claim that applying the algorithm for deriving hylomorphisms results in the following hylomorphism:

$$\begin{aligned} foo &= [\![\phi, id, out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\ &\qquad \text{where } \phi = \lambda().\ \mathit{Leaf} \triangledown \lambda(v, v').\ \mathit{Node}(\mathit{double}(v), \mathit{squareNodes}(v'), v') \end{aligned}$$

Our implementation of the algorithm, on the other hand, produces the following hylomorphism:

$$\begin{aligned} foo &= [\![\phi, id, \psi]\!]_{F_{T_A}, F_{T_A}} \\ &\qquad \text{where } \phi = \lambda().\ \mathit{Leaf} \triangledown \lambda(v, v_1', v_2').\ \mathit{Node}(\mathit{double}(v), \mathit{squareNodes}(v_1'), v_2') \\ &\qquad\qquad \psi = \lambda v_s.\ \text{case } v_s \text{ of } \mathit{Nil} \to (1, ()); \mathit{Cons}(a, as) \to (2, (a, as, as)) \end{aligned}$$

The results differ because of a disagreement over how to interpret the mathematical notation in the description of the derivation algorithm. We claim that their use of set union indicates that every occurrence of the recursive function should be replaced by a fresh variable, even if the function is being applied to an argument which has appeared before. Onoue et al. use a more complicated union procedure which assigns the same variable to multiple occurrences of the function on the same argument.

Both hylomorphisms inline to the same recursive function, so they are equivalent representations. The hylomorphism produced by Onoue et al. is a catamorphism over the functor for lists and therefore is readily fused with hylomorphisms over the list functor. Our hylomorphism operates over the functor for trees and would be an anamorphism if the function did not include a call to `squareNodes`. Thus, each form is open to different possibilities for fusion. But neither restructuring algorithm in the HYLO system can convert one form to the other because they do not affect the recursive elements. A transformation between the forms is needed if we hope to fully exploit the Acid Rain rules.

Certainly we can inline a hylomorphism and rederive using the other interpretation of the deriving algorithm, but it would be more efficient to have an algorithm which operates on the hylomorphism representation. Such algorithms do exist—one for $\phi$ and one for $\psi$.

For both parts of the hylomorphism, there is a general algorithm for restructuring which performs as much restructuring as possible. However, it is possible to implement simplified procedures which only perform restructuring if the output has a chance of being in *in* or *out* form. Recursive restructuring does not simplify the derivation of $\tau$ or $\sigma$ and does not bring internal hylomorphisms together. Its only purpose, then, is to reveal a catamorphism or anamorphism, so the simpler procedure is sufficient. For completeness, we describe both the general and simplified algorithms.

### 3.8.1 Recursive Restructuring of $\phi$

The middle component $\eta$ of a hylomorphism represents a natural transformation that can be shifted from one side of the hylomorphism to the other. In order for a function to be a natural transformation it cannot perform computation on the recursive variables. That is, the recursive elements of the output must be recursive variables, not expressions, and the non-recursive elements cannot contain references to the recursive variables. However, the transformation is free to ignore or duplicate recursive variables in the recursive elements of the output.

To perform recursive restructuring of $\phi$, we traverse each $\phi_i$ and collect the recursive variables that have appeared so far. When a recursive variable appears more than once, each new occurrence is replaced by a fresh identifier and the replacement is remembered in a list which associates the new identifier to the old identifier that it replaced. The new identifiers are added to the recursive inputs of $\phi_i$ and the recursive outputs of $\eta_i$ are similarly extended to include the values for these new inputs. The values are just duplicates of the values for the old variables that were replaced. The algorithm is given formally in Figure 3-6.

The algorithm as given threads the list of recursive variables $b$ from left to right in subexpressions. If the algorithm traverses from left to right in this way and keeps the list of recursive variables in the order that they are encountered, then the $\phi$ of the restructured hylomorphism will have its variables in the order that we enforced in

$$\mathcal{R}_\phi[\![ [\![\phi_1 \triangledown \cdots \triangledown \phi_n, \eta_1 + \cdots + \eta_n, \psi]\!]_{G,F} ]\!] = [\![\phi_1' \triangledown \cdots \triangledown \phi_n', \eta_1' + \cdots + \eta_n', \psi]\!]_{G',F}$$

where $\lambda(u_{i_1}, \ldots, u_{i_{k_i}}, v_{i_1}, \ldots, v_{i_{l_i}}).\ t_i = \phi_i$

$$\begin{aligned}
&G_1 + \cdots + G_n = G \\
&\Gamma(u_{i_1}) \times \cdots \times \Gamma(u_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} = G_i \\
&\lambda(\ldots).\ (t_{i_1}, \ldots, t_{i_{k_i}}, tt_{i_1}, \ldots, tt_{i_{l_i}}) = \eta_i \\
&(\{(v_{i_1}', tt_{i_1}'), \ldots, (v_{i_{m_i}}', tt_{i_{m_i}}')\}, t_i') = \mathcal{H}[\![t_i]\!]\ \{(v_{i_1}, tt_{i_1}), \ldots, (v_{i_{l_i}}, tt_{i_{l_i}})\}\ \{\} \\
&G_i' = \Gamma(u_{i_1}) \times \cdots \times \Gamma(u_{i_{k_i}}) \times I_1 \times \cdots \times I_{m_i} \\
&G' = G_1' + \cdots + G_n' \\
&\phi_i' = \lambda(u_{i_1}, \ldots, u_{i_{k_i}}, v_{i_1}', \ldots, v_{i_{m_i}}').\ t_i' \\
&\eta_i' = \lambda(\ldots).\ (t_{i_1}, \ldots, t_{i_{k_i}}, tt_{i_1}', \ldots, tt_{i_{m_i}}')
\end{aligned}$$

Assume that $u$ is a fresh variable in the following:

$$\begin{aligned}
\mathcal{H}[\![v]\!]\ s_r\ b \quad &= \quad \text{if } (v, e) \in s_r \\
&\qquad \text{then if } (v, _) \in b \\
&\qquad\qquad \text{then } (\{(u, e)\} \cup b, u) \\
&\qquad\qquad \text{else } (\{(v, e)\} \cup b, v) \\
&\qquad \text{else } (b, v) \\
\mathcal{H}[\![l]\!]\ s_r\ b \quad &= \quad (b, l) \\
\mathcal{H}[\![\lambda v.t]\!]\ s_r\ b \quad &= \quad (b', \lambda v.t') \\
&\qquad \text{where } (b', t') = \mathcal{H}[\![t]\!]\ s_r\ b \\
\mathcal{H}[\![\texttt{case}\ t_0\ \texttt{of}\ p_1 \to t_1; \cdots; p_n \to t_n]\!]\ s_r\ b &= \\
&\qquad (b_n', \texttt{case}\ t_0'\ \texttt{of}\ p_1 \to t_1'; \cdots; p_n \to t_n') \\
&\qquad \text{where } (b_0', t_0') = \mathcal{H}[\![t_0]\!]\ s_r\ b \\
&\qquad\qquad (b_i', t_i') = \mathcal{H}[\![t_i]\!]\ s_r\ b_{i-1}' \\
\mathcal{H}[\![v\ t_1\ \ldots\ t_n]\!]\ s_r\ b \quad &= \quad (b_n', v'\ t_1'\ \ldots\ t_n') \\
&\qquad \text{where } (b_0', v') = \mathcal{H}[\![v]\!]\ s_r\ b \\
&\qquad\qquad (b_i', t_i') = \mathcal{H}[\![t_i]\!]\ s_r\ b_{i-1}' \\
\mathcal{H}[\![C\ t_1\ \ldots\ t_n]\!]\ s_r\ b \quad &= \quad (b_n', C\ t_1'\ \ldots\ t_n') \\
&\qquad \text{where } (b_1', t_1') = \mathcal{H}[\![t_1]\!]\ s_r\ b \\
&\qquad\qquad (b_i', t_i') = \mathcal{H}[\![t_i]\!]\ s_r\ b_{i-1}' \\
\mathcal{H}[\![\texttt{let}\ v = t_1\ \texttt{in}\ t_0]\!]\ s_r\ b \quad &= \quad (b_1', \texttt{let}\ v = t_1'\ \texttt{in}\ t_0') \\
&\qquad \text{where } (b_0', t_0') = \mathcal{H}[\![t_0]\!]\ s_r\ b \\
&\qquad\qquad (b_1', t_1') = \mathcal{H}[\![t_1]\!]\ s_r\ b_0' \\
\mathcal{H}[\![\texttt{letrec}\ v_1 = t_1; \cdots; v_n = t_n\ \texttt{in}\ t_0]\!]\ s_r\ b &= \\
&\qquad (b', \texttt{letrec}\ v_1 = t_1'; \cdots; v_n = t_n'\ \texttt{in}\ t_0') \\
&\qquad \text{where } (b_1', t_1') = \mathcal{H}[\![t_1]\!]\ s_r\ b \\
&\qquad\qquad (b_i', t_i') = \mathcal{H}[\![t_i]\!]\ s_r\ b_{i-1}' \\
&\qquad\qquad (b', t_0') = \mathcal{H}[\![t_0]\!]\ s_r\ b_n'
\end{aligned}$$

Figure 3-6: Recursive Restructuring Algorithm $\mathcal{R}_\phi$

$$
\begin{array}{l}
\mathcal{R}_\psi[\![ [\![\phi, \eta_i + \cdots + \eta_n, \psi]\!]_{G,F} ]\!] = [\![\phi, \eta'_1 + \cdots + \eta'_n, \psi']\!]_{G,F'} \\
\text{where } \lambda v_s.\ \text{case } t_0 \text{ of } p_1 \to (1, t_1); \cdots; p_n \to (n, t_n) = \psi \\
\qquad (t_{i_1}, \ldots, t_{i_{k_i}}, tt_{i_1}, \ldots, tt_{i_{l_i}}) = t_i \\
\qquad F_1 + \cdots + F_n = F \\
\qquad \Gamma(t_{i_1}) \times \cdots \times \Gamma(t_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} = F_i \\
\qquad \lambda(u_{i_1}, \ldots, u_{i_{k_i}}, v_{i_1}, \ldots, v_{i_{l_i}}).\ e_i = \eta_i \\
\qquad (\{(tt'_{i_1}, v'_{i_1}), \ldots, (tt'_{i_{m_i}}, v'_{i_{m_i}})\}, s) = \mathcal{I}[\![\{(tt_{i_1}, v_{i_1}), \ldots, (tt_{i_{l_i}}, v_{i_{l_i}})\}]\!]\ \{\}\ \{\} \\
\qquad \eta'_i = \lambda(u_{i_1}, \ldots, u_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{m_i}}).\ e_i[s] \\
\qquad t'_i = (t_{i_1}, \ldots, t_{i_{k_i}}, tt'_{i_1}, \ldots, tt'_{i_{m_i}}) \\
\qquad \psi' = \lambda v_s.\ \text{case } t_0 \text{ of } p_1 \to (1, t'_1); \cdots; p_n \to (n, t'_n) \\
\qquad F'_i = \Gamma(t_{i_1}) \times \cdots \times \Gamma(t_{i_{k_i}}) \times I_1 \times \cdots \times I_{m_i} \\
\qquad F' = F'_1 + \cdots + F'_n
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{I}[\![\{\}]\!]\ b\ s & = & (b, s) \\
\mathcal{I}[\![\{(t,v)\} \cup p]\!]\ b\ s & = & \text{if } (t, v') \in b \text{ then } \mathcal{I}[\![p]\!]\ b\ \{(v, v')\} \cup s \\
 & & \qquad\qquad\quad \text{else } \mathcal{I}[\![p]\!]\ \{(t,v)\} \cup b\ s
\end{array}
$$

Figure 3-7: Recursive Restructuring Algorithm $\mathcal{R}_\psi$

the traditional restructuring algorithm and in the *in* form detection algorithm. Thus, in our algorithm to find fusible hylomorphisms, before testing for *in* form, we should perform both non-recursive and recursive restructuring, after which the hylomorphism will have $\phi$ in *in* form if it is in fact a catamorphism.

Since recursive restructuring only helps in determining if a hylomorphism is a catamorphism or anamorphism, we can simplify the algorithm to abort if its input has no chance of being in *in* form. Specifically, if the body $t_i$ of each $\phi_i$ is not a constructor with all variable arguments, then it is pointless to perform recursive restructuring. Thus, we only need the cases from algorithm $\mathcal{H}$ for constructors and for variables.

### 3.8.2 Recursive Restructuring of $\psi$

To perform recursive restructuring of $\psi$, we look for duplicate expressions in the recursive elements of each case $t_i$. Duplicate expressions are eliminated from $t_i$ and the corresponding inputs to $\eta_i$ are removed, with occurrences of the removed variables being replaced by the one remaining input variable receiving the same value. The algorithm is given formally in Figure 3-7.

As with the algorithm for $\phi$, we can simplify the algorithm to only operate on inputs which have a chance of being in *out* form. For a $\psi$ to be in *out* form, the recursive elements of $t_i$ must all be variables, and not expressions. Thus, we do not need to look for multiple occurrences of the same expression, only multiple occurrences of the same variable.

## 3.9 Examples

In the following examples, we use the new representation for hylomorphisms discussed in Section 3.8. This new representation differs from the one given in Section 3.2 in that we additionally record the recursive output of $\eta$, which we no longer assume to be identical to the recursive input. The type declaration for this component is then:

```
type Eta = [([Var],[Var],[Expr],[Var])]
```

### 3.9.1 Deforesting Lists

Consider the following program containing recursive functions on lists:

```
sum xs = case xs of
              [] -> 0
              (a:as) -> a + (sum as)


map g xs = case xs of
                [] -> []
                (a:as) -> ((g a):(map g as))


upto n m = case (m>n) of
                True -> []
                False -> (m:(upto n (m+1)))


square x = x*x
```

```
main = print (sum (map square (upto 10 1)))
```

Running the deforestation optimization on this program derives the following hylomorphisms:

$$\begin{aligned}
&\texttt{sum} = [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).0 \ \triangledown \ \lambda((u_1),(v_1)).(u_1 + v_1) \\
&\qquad\qquad \psi = \lambda v_s.\texttt{case}\ \ v_s\ \ \texttt{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nil} \rightarrow (\mathit{1}, ((),())) \\
&\qquad\qquad\qquad\qquad \mathit{Cons}(a, as) \rightarrow (\mathit{2}, ((a),(as)))
\end{aligned}$$

$$\begin{aligned}
&\texttt{map} = \lambda g.\ [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).\mathit{Nil} \ \triangledown \ \lambda((u_{\mathit{1}}),(v_{\mathit{1}})).\mathit{Cons}(g\ \ u_{\mathit{1}}, v_{\mathit{1}}) \\
&\qquad\qquad \psi = \lambda v_s.\texttt{case}\ \ v_s\ \ \texttt{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nil} \rightarrow (\mathit{1}, ((),())) \\
&\qquad\qquad\qquad\qquad \mathit{Cons}(a, as) \rightarrow (\mathit{2}, ((a),(as)))
\end{aligned}$$

$$\begin{aligned}
&\texttt{upto} = \lambda n.\ [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).\mathit{Nil} \ \triangledown \ \lambda((u_{\mathit{1}}),(v_{\mathit{1}})).\mathit{Cons}(u_{\mathit{1}}, v_{\mathit{1}}) \\
&\qquad\qquad \psi = \lambda v_s.\texttt{case}\ \ v_s > n\ \ \texttt{of} \\
&\qquad\qquad\qquad\qquad \mathit{False} \rightarrow (\mathit{1}, ((),())) \\
&\qquad\qquad\qquad\qquad \mathit{True} \rightarrow (\mathit{2}, ((v_s),(v_s + \mathit{1})))
\end{aligned}$$

These hylomorphisms are represented in our compiler as:

```
sum
 ([([], [], 0), ([u], [v], u + v)],
  [([], [], []), ([u], [v], [u], [v])],
  (xs, xs, [([], []), (:, [a, as])], [([], []), ([a], [as])]),
  [])


map
 ([([], [], []), ([u], [v], g u : v)],
  [([], [], []), ([u], [v], [u], [v])],
  (xs, xs, [([], []), (:, [a, as])], [([], []), ([a], [as])]),
```

```
  [Lambda g])
```

```
upto
 ([([], [], []), ([u], [v], u : v)],
  [([], [], []), ([u], [v], [u], [v])],
  (m, m > n, [(False, []), (True, [])], [([], []), ([m], [m + 1])]),
  [Lambda n])
```

To deforest `main`, we begin with the composition `(map square (upto 10 1))` which is a composition of a catamorphism with an anamorphism, so it can be fused into the following hylomorphism:

```
 ([([], [], []), ([u], [u], g u : v)],
  [([], [], []), ([u], [v], [u], [v])],
  (m, m > n, [(False, []), (True, [])], [([], []), ([m], [m + 1])]),
  [Let [Def g square, Def n 10], Apply 1])
```

In order to fuse the application of `sum` to this hylomorphism, it needs to be restructured into the following anamorphism:

```
 ([([], [], []), ([u], [u], u : v)],
  [([], [], []), ([u], [v], [g u], [v])],
  (m, m > n, [(False, []), (True, [])], [([], []), ([m], [m + 1])]),
  [Let [Def g square, Def n 10], Apply 1])
```

Now the $\phi$ of this hylomorphism is in *in* form and can be fused with the hylomorphism for `sum` which has $\psi$ in *out* form. This fusion produces the following hylomorphism:

```
 ([([], [], 0), ([u], [v], u + v)],
  [([], [], []), ([u], [v], [g u], [v])],
  (m, m > n, [(False, []), (True, [])], [([], []), ([m], [m + 1])]),
  [Let [Def g square, Def n 10], Apply 1])
```

This hylomorphism represents the fusion of the compositional definition of `main` and can be inlined into the following new expression for `main`:

```
let g = square
    n = 10
    hylo m = case (m > n) of
                 False -> 0
                 True -> (g m) + (hylo (m + 1))
in  hylo 1
```

This definition, with a single loop, is an improvement over the original definition with three loops and two intermediate lists.

### 3.9.2 Deforesting Trees

To illustrate how the system works on a data type other than lists, we use the following program on trees:

```
data Tree t = Leaf | Branch t (Tree t) (Tree t)

makeTree :: (t -> t) -> [t] -> Tree t
makeTree f1 [] = Leaf
makeTree f1 (x:xs) =
    Branch x (makeTree f1 xs) (mapTree f1 (makeTree f1 xs))

sumTree :: Tree Int -> Int
sumTree (Leaf) = 0
sumTree (Branch n l r) = n + (sumTree l) + (sumTree r)

mapTree :: (t -> t) -> Tree t -> Tree t
mapTree f2 (Leaf) = Leaf
mapTree f2 (Branch n l r) =
    Branch (f2 n) (mapTree f2 l) (mapTree f2 r)

upto :: Int -> Int -> [Int]
upto n m = case (m>n) of
```

```
                True -> []
                False -> (m:(upto n (m+1)))


square x = x * x


main = print (sumTree (mapTree (+1) (makeTree square (upto 8 1))))
```

When we run the deforestation optimization on this program, we derive the following hylomorphisms (plus a hylomorphism for `upto` which has not changed from the previous example):

$$\begin{aligned}
&\texttt{sumTree} = [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).0 \triangledown \lambda((u_1),(v_1,v_2)).(u_1+v_1+v_2) \\
&\qquad\quad \psi = \lambda v_s.\texttt{case } v_s \texttt{ of} \\
&\qquad\qquad\qquad \mathit{Leaf} \to (\mathit{1}, ((),())) \\
&\qquad\qquad\qquad \mathit{Branch}(n,l,r) \to (\mathit{2}, ((n),(l,r)))
\end{aligned}$$

$$\begin{aligned}
&\texttt{mapTree} = \lambda f.\ [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).\mathit{Leaf} \triangledown \lambda((u_{\mathit{1}}),(v_{\mathit{1}},v_{\mathit{2}})).\mathit{Branch}((f\ \ u_{\mathit{1}}), v_{\mathit{1}}, v_{\mathit{2}}) \\
&\qquad\quad \psi = \lambda v_s.\texttt{case } v_s \texttt{ of} \\
&\qquad\qquad\qquad \mathit{Leaf} \to (\mathit{1}, ((),())) \\
&\qquad\qquad\qquad \mathit{Branch}(n,l,r) \to (\mathit{2}, ((n),(l,r)))
\end{aligned}$$

$$\begin{aligned}
&\texttt{makeTree} = \lambda f.\ [\![\phi, id, \psi]\!] \\
&\quad \text{where } \phi = \lambda((),()).\mathit{Leaf} \triangledown \lambda((u_{\mathit{1}}),(v_{\mathit{1}},v_{\mathit{2}})).\mathit{Branch}(u_{\mathit{1}}, v_{\mathit{1}}, \mathit{mapTree}(f, v_{\mathit{2}})) \\
&\qquad\quad \psi = \lambda v_s.\texttt{case } v_s \texttt{ of} \\
&\qquad\qquad\qquad \mathit{Nil} \to (\mathit{1}, ((),())) \\
&\qquad\qquad\qquad \mathit{Cons}(a, as) \to (\mathit{2}, ((a),(as, as)))
\end{aligned}$$

These hylomorphisms are represented in our compiler as:

```
sumTree
 ([([], [], 0), ([u], [v1, v2], u + v1 + v2)],
  [([], [], []), ([u], [v1, v2], [u], [v1, v2])],
  (xs, xs, [(Leaf, []), (Branch, [n, l, r])],
```

```
   [([], []), ([n], [l, r])]),
  [])
```

```
mapTree
 ([([], [], Leaf), ([u], [v1, v2], Branch (f2 u) v1 v2)],
  [([], [], []), ([u], [v1, v2], [u], [v1, v2])],
  (xs, xs, [(Leaf, []), (Branch, [n, l, r])],
   [([], []), ([n], [l, r])]),
  [Lambda f2])
```

```
makeTree
 ([([], [], Leaf), ([u], [v1, v2], Branch u v1 (mapTree f1 v2))],
  [([], [], []), ([u], [v1, v2], [u], [v1, v2])],
  (xs, xs, [([], []), (:, [x, xs])], [([], []), ([x], [xs, xs])]),
  [Lambda f1])
```

To deforest this program, we begin with the composition (`makeTree square (upto 8 1)`) in `main`. As in the example for lists, the function `upto` is an anamorphism, but here the consuming hylomorphism is not immediately recognizable as a catamorphism. Recursive restructuring of the $\psi$ in `makeTree` is needed to remove the duplicate recursive variables. Once this is accomplished, the composition can be fused into the following hylomorphism:

```
([([], [], Leaf), ([u], [v1, v2], Branch u v1 (mapTree f1 v2))],
 [([], [], []), ([u], [v], [u], [v, v])],
 (m, m > n, [(False, []), (True, [])], [([], []), ([m], [m + 1])]),
 [Let [Def f1 square, Def n 8], Apply 1])
```

The $\eta$ of this hylomorphism duplicates the recursive variable, so recursive restructuring of $\phi$ is not needed. However, the hylomorphism is not a catamorphism, so $\tau$ will need to be derived if we are to fuse this hylomorphism with `mapTree`, which is an anamorphism. After deriving $\tau$ and performing the fusion, we produce the following hylomorphism:

```
([([], [], Leaf),
  ([u1, u2], [v1, v2],
   Branch (f2 u1) v1
          (let f3 = u2
               h xs = case xs of
                        Leaf -> Leaf
                        Branch n l r -> Branch (f2 (f3 n)) (h l) (h r)
           in  h v2))],
 [([], [], []), ([u], [v], [u, f1], [v, v])],
  (m, m > n, [(False, []), (True, [])], [([], []), ([m], [m + 1])]),
 [Let [Def f2 = (+1), Def f1 = square, Def n = 8], Apply 1])
```

For this hylomorphism to be fused with `sumTree`, we will again need to derive $\tau$. It is worth noting that the internal hylomorphism `mapTree` was altered in the deriving process and had to be inlined. Now when we go to derive $\tau$ again, we need to recognize the entire inlined expression as a hylomorphism being applied to a recursive variable, in order to determine that $\phi$ is in the canonical form required for deriving $\tau$. Our first attempt at this recognition process did not account for the surrounding let-statements and $\lambda$-abstractions that result from inlining. In this case, where the let-statement is binding a variable to a variable, it is easy to suggest that the hylomorphism-inlining algorithm inline these variable-to-variable let-statements and thus eliminate the problem. For more complicated let-bindings, the inlining algorithm could produce a hylomorphism with multiple non-recursive arguments and apply it to the bound values. However, we would rather produce hylomorphisms that have constant arguments lifted out of the loop and we can leave some simplifying to later parts of the compiler. Thus, we leave the inlining process unchanged and instead use a recognizing algorithm which gathers up the context statements and adds them to the internal representation of the hylomorphism.

Once we have derived $\tau$, we can apply the fusion rule and produce the following hylomorphism:

```
([([], [], 0),
```

```
  ([u1, u2], [v],
   u1 + v + (let f3 = u2
                 h xs = case xs of
                          Leaf -> 0
                          Branch n l r -> (f2 (f3 n)) + (h l) + (h r)
             in  h v))],
 [([], [], []), ([u], [v], [f2 u, f1])],
 (m, m > n, [(False, []), (True, [])], [([], []), ([m], [m + 1])]),
 [Let [Def f2 = (+1), Def f1 = square, Def n = 8], Apply 1])
```

which inlines into the following expression:

```
let f2 x = x + 1
    f1 = square
    n = 8
    h1 m =
      case (m > n) of
        False -> 0
        True -> (f2 m) + (h1 (m + 1)) +
                let f3 = f1
                    h2 a2 =
                      case a2 of
                        Leaf -> 0
                        Branch n l r -> (f2 (f3 n)) + (h2 l) + (h2 r)
                in  h2 (h1 (m + 1))
in  h1 1
```

This new definition for `main` is an improvement over the original which used three loops held together by an intermediate list and two intermediate trees. The new loop produces no lists or trees.

# Chapter 4

# Related Work

Olaf Chitil [5, 4] presents interesting work on how type inference can be used to derive `build` forms for most algebraic data types. In [5], he mentions Fegaras's [6] work on the use of free theorems [27] to derive the producer and consumer forms. These two systems, plus the hylomorphism system, all seem to be dancing around the same issue. It seems that a "unified theory of fusion" is needed.

As mentioned in the introduction, there exist extensions to the hylomorphism theory that make using hylomorphisms as the canonical form enticing. Alberto Pardo [22] has explored the fusion of recursive programs with effects, using two schemes he calls *monadic unfold* and *monadic hylomorphisms* and an updated Acid Rain theorem for fusing the new forms. Pardo claims that his extended rules can still perform traditional fusion, as purely functional hylomorphisms are special cases of monadic hylomorphisms with an identity monad component.

The HYLO authors have also proposed a few extensions to the theory. Recognizing that their system could not properly handle functions which induct over multiple data structures, like `zip`, they proposed an extension to the Acid Rain rules. Their extension meant the addition of a new rule to specifically handle such functions and not sharing the same form as the original Acid Rain rules. In later work [13], Iwasaki et al. extended the hylomorphism theory to use mutual hylomorphisms as a second canonical form, providing rules for operating on mutual hylomorphisms and for fusing with single recursive hylomorphisms. Unlike the previous extension, these new Acid

Rain rules followed the same form as the original. Iwasaki et al. show in that paper how functions which traverse over multiple data structures can be expressed in this new canonical form. Because the Acid Rain rules are not fundamentally changed, this new extension to the theory is preferred and is more powerful, allowing for the representation of mutual recursive functions.

Hylomorphisms are important to the HYLO authors because they see multiple calculational transformations being performed in a compiler [25]. One such transformation, based on mutual hylomorphisms, is Hu, Takeichi, and Chin's parallelizing transformation [12], which we discuss in the concluding remarks on parallelism in the next chapter.

As Gill [9] notes, Mark Jones [14] presents a framework for embedding the "bananas and lenses" notation directly in Haskell. Whether such representation can be used in a system, such as automatic deforestation, which manipulates these higher-order functions is still to be determined.

Despite the paper by Gibbons and Jones [8], unfold is still under-appreciated. Their paper discusses uses for unfold in functional programming, including its use in deforestation. Gibbons and Jones refer to the HYLO system as operating on functions which are compositions of `foldr` and `unfold`. They demonstrate the HYLO system on an example program developed in the course of the paper and show how it deforests to a simpler program provided by another source. It would be interesting to further explore the HYLO system in terms of direct manipulation of folds and unfolds, rather than hylomorphisms. Gill [9] provides a start in this direction by giving possible definitions for `unbuild` and `unfold` and the associated `unbuild/unfold` deforestation rule.

# Chapter 5

# Conclusions

The HYLO algorithms described by Onoue et al. have been implemented in the pH compiler. This experience showed that some modifications to the algorithms are necessary. The restructuring algorithms as given formally in [21] do not perform as described verbally. We revised the descriptions of these algorithms and provided descriptions for additional restructuring algorithms which were missing from the HYLO system.

We also showed that the restructuring algorithms are overloaded in the HYLO system. Onoue et al. use restructuring for three different purposes without recognizing that these purposes place potentially conflicting requirements on the output of the restructuring algorithms. First, restructuring is used to bring internal hylomorphisms together for fusion. Second, restructuring is used to reveal whether a hylomorphism is a catamorphism or anamorphism, to allow for fusion with other hylomorphisms. Third, restructuring is used to simplify a hylomorphism before $\tau$ or $\sigma$ is derived. In this paper, we discussed algorithms for all three purposes. Each is a valid application of the HyloShift rule, but they differ in how much computation is shifted. Restructuring for the purpose of fusing internal hylomorphisms, which requires that nearly all computation be shifted into $\eta$, is at odds with the restructuring prior to deriving $\tau$, which requires that some hylomorphisms be left unshifted. An implementation of the HYLO system should have separate algorithms for these purposes.

We also learned from this experience that operations on hylomorphisms should be relative. A function which operates on a data type described by a functor $\mathsf{F}$ can be expressed as a hylomorphism over a different, but related functor $\mathsf{F}'$. When attempting to fuse two hylomorphisms operating on the same data type but expressed over different functors, it is necessary to massage the hylomorphisms into forms on a common functor, either through restructuring or deriving $\tau$ or $\sigma$. It is not necessary, however, that the common functor be the functor $\mathsf{F}$ for the data type. In fact, requiring that the hylomorphisms be expressed over $\mathsf{F}$ restricts the number of cases in which fusion is possible. Thus, massaging a hylomorphism should be performed relative to the hylomorphism to which we hope to fuse.

Our implementation performs well on sample programs containing recursive functions on lists and trees. The success of this system on real world examples still needs to be explored. According to Olaf Chitil [5], Christian Tüffers's thesis [26] shows that the derivation algorithm used by the HYLO system can transform most definitions in real world programs into hylomorphisms. Thus, there is reason to believe that this system can succeed in practice.

Exploration will likely reveal that some issues common to deforestation optimizations, such as inlining and handling of wrapper computations, still need to be addressed in our implementation. As mentioned in Section 3.6.5 describing the algorithm for finding fusible compositions, there are many cases where fusion will be missed because our algorithm does not perform the necessary inlining. We believe that research on these issues in other systems will be applicable here. Maessen [17] discusses the desugaring used by his optimization and briefly addresses inlining; the majority of Gill's thesis [9] is discussion of optimizations which are necessary before the `foldr/build` deforestation pass can be effective, and which he explicitly claims are necessary for any calculational optimization to be practical; and Chitil [4] discusses how the worker/wrapper scheme suggested by Gill can be used to optimize inlining in his deforestation system.

## 5.1 Preserving Parallelism

One of the original goals of this project was to extend the HYLO system to an optimization which can fuse reductions without having to first serialize them into folds. In the course of the project, it became clear that implementing the current HYLO system would be a sufficient thesis in itself and that parallelism would have to be left for future work. We provide here a few thoughts on the current system and directions for future research.

The HYLO system as implemented does not introduce any ordering. However, the system can only perform deforestation on programs which are described in terms of folds. In the pH compiler, deforestation is relied on heavily to optimize the desugaring of list and array comprehensions. As mentioned in the description of Maessen's scheme, operations on arrays and open lists are represented as reductions on lists. Thus, the pH compiler relies heavily on deforestation to optimize the results of desugaring operations on these structures. For HYLO to be an effective optimization in the pH compiler, it would need to fuse reductions without introducing an arbitrary ordering to the program. Transforming reductions into folds is unacceptable. The HYLO system can still be useful for deforesting functions on user-defined types, for which reduction operators are not provided to the programmer.

Hu, Takeichi, and Chin's paper "Parallelization in Calculational Forms" [12] gives a calculational system for taking a program on cons-lists and transforming it into a program with functions on join-lists. That is, they create instances of `reduce` from folds. They call this technique *parallelization* and they perform the transformation using *mutumorphisms*, which are mutually recursive forms of Bird's list homomorphisms and which are related to the recursive hylomorphisms described in [13]. According to Hu et al., the point of transformations are for programmers to write horribly inefficient programs which are easy to understand, leaving it up to the compiler to derive an efficient equivalent program. In the same way that deforestation allows us to write modular programs which the compiler will fuse into a more efficient form, they say that parallelization allows us to write serial programs which the compiler will convert into parallel descriptions. Their philosophy is that calculational fusion should not be parallelized, but instead a parallelizing transformation should be performed after other calculational transformations have been applied. The benefit of their parallelizing step is that it generalizes to arbitrary data types rather than simply lists. The benefit of such a system in the pH compiler would be not only to recover parallelism after the deforestation pass, but also to automatically derive reductions over user-defined types.

Understanding the parallelizing optimization of Hu et al. might lead to insight into how the representation of hylomorphisms in the HYLO system could be be extended to capture parallel operators such as `reduce`. If hylomorphisms could be derived for functions which operate on join-lists, then the HYLO system could be used effectively in pH without having to arbitrarily transform reductions into folds. Currently, natural transformations like `map` can be expressed and deforested with minimal change to the system, because they do not require operating on the consumer and producer aspects of the hylomorphism.

Another possibility is to extend the HYLO system in the same way that Maessen extended the short cut deforestation of Gill et al. That is, to transform the recursive functions in a program into higher-order functions that build up the original function, which was either iterative or recursive. Then the task becomes to fuse these higher-order functions.

# Bibliography


[1] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, Hemel Hempstead, England, 1997.

[2] Richard S Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[3] Richard S. Bird and O. De Moor. Relational program derivation and context-free language recognition. In A. W. Roscoe, editor, *A Classical Mind: Essays dedicated to C.A.R. Hoare*, pages 17–35. Prentice Hall International, 1994.

[4] Olaf Chitil. Type-inference based short cut deforestation (nearly) without inlining. In *Draft Proceedings of the 11th International Workshop on Implementation of Functional Languages*, pages 17–32, 1999.

[5] Olaf Chitil. Type inference builds a short cut to deforestation. In *Proceedings of the 1999 International Conference on Functional Programming*, 1999.

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

[7] GHC Team. The glasgow haskell compiler user's guide, version 4.06, 2000.

[8] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the 1998 International Conference on Functional Programming*, 1998.

[9] Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, Glasgow University, January 1996.

[10] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Functional Programming and Computer Architecture*, pages 223–232. ACM SIGPLAN/SIGARCH, ACM, June 1993.

[11] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, Philadelphia, PA, 1996. ACM Press.

[12] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–328, 1998.

[13] Hiedya Iwasaki, Zhenjiang Hu, and Masato Takeichi. Towards manipulation of mutually recursive functions. In *Proceedings of the 3rd Fuji International Symposium on Functional and Logic Programming*, pages 61–79, 1998.

[14] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science 925, pages 97–136. Springer-Verlag, 1995.

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

[16] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 314–323, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[17] Jan-Willem Maessen. Eliminating intermediate lists in pH using local transformations. Master's thesis, MIT Department of Electrical Engineering and Computer Science, May 1994.

[18] Jan-Willem Maessen. Simplifying parallel list traversal. Technical Report CSG-Memo-370, Massachussets Institute of Technology, Laboratory for Computer Science, January 1995.

[19] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '91*, pages 124–144, 1991.

[20] R. S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J. Maessen, and Y. Zhou. pH language reference manual, version 1.0. Technical Report CSG-Memo-369, Massachussets Institute of Technology, Laboratory for Computer Science, January 1995.

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

[22] Albert Pardo. Fusion of recursive programs with computational effects. In *Proceedings of the 1st Workshop on Coalgebraic Methods in Computer Science*, 1998.

[23] R. A. Paterson, May 1999. Personal Communication.

[24] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.

[25] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 30(3), 1998.

[26] Christian Tüffers. Erweiterung des Glasgow-Haskell-Compilers um die Erkennung von Hylomorphismen. Master's thesis, RWTH Aachen, 1998.

[27] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, Sept 1989.