Monadic Short Cut Deforestation

Cecilia Manzino
Licenciatura en Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina
ceciliam@fceia.unr.edu.ar

December 2005

Supervised by:

Alberto Pardo
Instituto de Computación
Facultad de Ingeniería
Universidad de la República
Montevideo - Uruguay

Abstract

Functional programs are often constructed using a compositional style. This has the advantages of clarity and modularity. However, an intermediate data structure is created in every functional composition, giving rise in many cases to an efficiency problem. Deforestation is a technique that automatically removes intermediate data structures from programs. Various deforestation methods have been proposed. However, not all of them can be applied to programs with effects. In this thesis, we present some program transformations that can be used for the elimination of intermediate data structure from programs with effects.

Contents

1 Introduction

In functional programming it is common practice to write programs using a compositional style. In this approach a program is written by composing small functions using intermediate data structures to communicate them. For example, the function `any`, which tests if any of the elements of a list satisfies a predicate, may be defined as follows:

```
any p xs = or (map p xs)

  where or [] = False
        or (x:xs) = x || (or xs)

        map p [] = []
        map p (x:xs) = (p x): (map p xs)
```

In this definition `map` produces an intermediate list of booleans which is consumed by `or` to produce the final result.

Although this style of programming allows us to write clear and modular programs, the construction of intermediate data structures gives rise to an efficiency problem, since each element of these data structures must be allocated, examined and deallocated. If a strict evaluation strategy is used, the programs also require space to store the whole data structure.

A technique that automatically transforms a program into another that does not create intermediate data structures is called *deforestation* [Wad90]. Many methods have been proposed for this technique. Probably, the most successful one is "Shortcut Deforestation" [GLJ]. This method was incorporated in the Glasgow Haskell Compiler (GHC). It is based on a simple transformation, called `foldr/build`, which is used to remove the intermediate data structure created in the composition of functions written in terms of `foldr` and `build` . The functions `foldr` and `build` standardize the way in which lists are consumed and produced, respectively. In the case of function `any` we can define the functions `or` and `map` in terms of these functions and then apply the `foldr/build` transformation to eliminate the intermediate list, obtaining:

```
any p [] = False
any p (x:xs) = p x || any p xs
```

In [Sve02] a similar transformation was defined for the dual of `foldr`, called `destroy/unfoldr`. In this case, the functions `destroy` and `unfoldr` are used to represent functions that consume and produce lists, respectively. Both transformations can be generalized to any algebraic data type (see [TM95]).

The aim of this thesis is to present a transformation that extends shortcut deforestation to programs with effects, assuming that effects are modeled using monads.

An example of a program where we can apply such a transformation is the following.

```
asciiTable = putStr (map chr (enumFromTo 0 255))
```

This function prints all characters of the ASCII table. The intermediate lists produced by (`map chr (enumFromTo 0 255)`) and (`enumFromTo 0 255`) do not form part of the result. A more efficient version of `ascciTable`, which does not construct any intermediate list is the following:

```
asciiTable = putS' (\i -> if i>255 then Nothing
                            else Just (chr i, i+1)) 0

  where putS' f xs = case f xs of
                       Nothing -> return []
                       Just (c, cs) -> putChar c >> putS' f cs
```

We obtain the transformation by extending function `build` to produce monadic values and the way in which values are consumed in shortcut deforestation. We also present a generalization of each operator and transformation for any algebraic datatype.

The work is organized as follows. In section 2, we present the mathematical framework which serves as basis of this work. Section 3 reviews previous work on deforestation. Section 4 introduces monads. In section 5, we present shortcut deforestation for programs with effects and in section 6, we show how to corresponding transformation rules in the Glasgow Haskell Compiler and give some examples. Finally, section 7 presents some conclusions.

Throughout this work we will use a Haskell-like notation to write definitions and concepts.

2 Background

In this section we introduce the mathematical framework which provides the theoretical basis of this thesis. A more complete introduction to this theory can be found in [BdOM97, Fok92, Jeu93].

Throughout this work, definitions and concepts are given over the category *Cpo*, the category of complete partial orders with a least element $\perp$ and continuous functions. This choice will allow us to define some recursive operators as least fixed points of recursive equations. We will model types and programs in the following way: we interpret types as complete partial orders and programs as continuous functions.

We start presenting a construction that captures datatype declarations. Based on that construction, we define recursive operators associated with datatypes.

2.1 Functors

Functors are used to capture the signature of datatypes. We represent type constructors with functors on *Cpo*.

Definition 2.1 A functor consists of two components, denoted by F: a type constructor F, and a function $F : (a \to b) \to (F\ a \to F\ b)$, which satisfies the following equations:

$$
\begin{aligned}
&F\ id = id \\
&F\ (f \circ g) = F\ f \circ F\ g
\end{aligned}
$$

The first equation states that F preserves identities and the second that F preserves compositions.

As examples of functors we consider the basic functors: identity, constant, product and sum. These will permit us to construct a particular class of datatypes, called regular datatypes.

Definition 2.2 (Identity) The *identity* functor I, is defined as the identity on types and functions. That is,

$$\textbf{type}\ I\ a = a$$

$$\begin{aligned} &I :: (a \to b) \to (Ia \to Ib) \\ &I\ f = f \end{aligned}$$

Definition 2.3 (Constant) For any type t, the *constant* functor $\underline{t}$ is defined as follows:

$$\textbf{type}\ \underline{t}\ a = t$$

$$\begin{aligned} &\underline{t} :: (a \to b) \to (\underline{t}\ a \to \underline{t}\ b) \\ &\underline{t}\ f = id \end{aligned}$$

Definition 2.4 (Product) The *product* functor $(\times)$, which is a case of a bifunctor (a functor on two arguments), is defined as follows:

$$\textbf{data}\ a \times b = (a, b)$$

$$\begin{aligned} &(\times) :: (a \to c) \to (b \to d) \to (a \times b \to c \times d) \\ &(f \times g)\ (a, b) = (f\ a, f\ b) \end{aligned}$$

The following operators are related to the product functor:

$$\begin{aligned} &exl :: a \times b \to a \\ &exl\ (a, b) = a \end{aligned}$$

$$\begin{aligned} &exr :: a \times b \to b \\ &exr\ (a, b) = b \end{aligned}$$

$$\begin{aligned} &\triangle : (c \to a) \to (c \to b) \to c \to a \times b \\ &(f \triangle g)\ c = (f\ c,\ g\ c) \end{aligned}$$

The functions exl and exr are called left and right projections, respectively. They are used to inspect the elements of a product. The function $\triangle$ is called the split operation and is used to construct a product from an object.

The relationship between these operators and the product functor is given by the following equation:

$$f \times g = (f \circ exl) \triangle (g \circ exr)$$

Definition 2.5 (Sum) The *sum* bifunctor $(+)$ is defined by:

$$\textbf{data}\ a + b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

$$\begin{aligned}
&(+) :: (a \to c) \to (b \to d) \to (a + b \to c + d) \\
&(f + g)\ (Left\ a) = Left\ (f\ a) \\
&(f + g)\ (Right\ b) = Right\ (g\ b)
\end{aligned}$$

The following operators are related to this functor:

$$\begin{aligned}
&inl :: a \to a + b \\
&inl\ a = Left\ a \\
\\
&inr :: b \to a + b \\
&inr\ b = Right\ b \\
\\
&(\triangledown) :: (a \to c) \to (b \to c) \to (a + b \to c) \\
&(f \triangledown g)\ (Left\ a) = f\ a \\
&(f \triangledown g)\ (Right\ b) = g\ b
\end{aligned}$$

The functions inl and inr are called left and right injections. The function $(\triangledown)$ is called the junc operation. It combines the functions $f : a \to c$ and $g : b \to c$ into $(f \triangledown g)$, which applies f to left-tagged and g to right-tagged values.

The following equation gives the relationship between these operators and the sum functor:

$$f + g = inl \circ f \triangledown inr \circ g$$

Definition 2.6 Given two functors F and G, we define the functors $F \times G$ and $F + G$ as follows:

$$\begin{aligned}
&\textbf{type } (F \times G)\ a = F\ a \times G\ a \\
&(F \times G)\ f = F\ f \times G\ g
\end{aligned}$$

$$\begin{aligned}
&\textbf{type } (F + G)\ a = F\ a + G\ a \\
&(F + G)\ f = F\ f + G\ g
\end{aligned}$$

Definition 2.7 Regular functors are functors built from identities, constants, products, sums and type functors. They are generated by the following grammar:

$$F ::= I \mid \underline{t} \mid F \times F \mid F + F \mid D$$

where D stands for type functors, which are defined later.

Now, we present some examples of functors which are used to capture the signature of datatypes.

Example 2.1 The signature of the datatype of natural numbers,

$$\textbf{data } \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}^{1}$$

is captured by the functor N

$$\textbf{type } N\ a = () + a$$

$$\begin{aligned}
&N :: (a \to b) \to (N\ a \to N\ b) \\
&N\ f = id + f
\end{aligned}$$

Example 2.2 For the datatype of lists with elements of type a,

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

[1]Escribir interpretacin de barra

we can derive a functor L_a given by

$$\mathbf{type}\ L_a\ b = () + a \times b$$

$$L_a :: (b \to c) \to (L_a\ b \to L_a\ c)$$
$$L_a\ f = id + id \times f$$

Example 2.3 The signature of the datatype of leaf-labelled binary trees with elements of type a,

$$\mathbf{data}\ Btree\ a = Leaf\ a \mid Join\ (Btree\ a)\ (Btree\ a)$$

is captured by the functor B_a

$$\mathbf{type}\ B_a\ b = a + b \times b$$

$$B_a :: (b \to c) \to (B_a\ b \to B_a\ c)$$
$$B_a\ f = id + f \times f$$

2.2 Algebras, coalgebras and homomorphisms

Definition 2.8 Let F be a functor. An F-*algebra* is a function of type $F\ a \to a$. The type a is called the carrier of the F-algebra.

Definition 2.9 Given two F-algebras $h : F\ a \to a$ and $h' : F\ b \to b$, an F-*homomorphism* is a function $f : a \to b$ such that:

$$f \circ h = h' \circ F\ f$$

The dualization of algebras leads to the notion of coalgebra.

Definition 2.10 Given a functor F, a F-*coalgebra* is a function of type $a \to F\ a$. Like algebras, the type a is called the carrier of the F-coalgebra.

Definition 2.11 An F-*homomorphism* between the F-coalgebras $g : a \to F\ a$ and $g' : b \to F\ b$ is a function $f : a \to b$ that satisfies this equation:

$$g' \circ f = Ff \circ g$$

We often omit the prefix F for referring to algebras, coalgebras, and homomorphisms when it can be derived from the context.

2.3 Datatypes

In this section we define datatypes as canonical fixed points of functors.

Every regular datatype is understood as a solution of an equation $F\ x \cong x$, where F is the functor that captures its signature. The canonical solution to this equation is specified by a type μ_F together with an isomorphism given by an algebra $in_F : F\ \mu_F \to \mu_F$ and a coalgebra $out_F : \mu_F \to F\ \mu_F$. We say that μ_F is the dataype defined by the functor F. The algebra in_F encodes the constructors of the datatype, whereas its inverse out_F encodes the destructors.

In the following examples we define datatypes as canonical fixed points of the functors given in 2.1.

Example 2.1 The canonical solution to the type equation $() + x \cong x$ gives the datatype of natural numbers and the functions in_N and out_N,

$$\begin{aligned}
&in_N : N\ Nat \to Nat\\
&in_N = const\ Zero \triangledown Succ\\
\\
&out_N : Nat \to N\ Nat\\
&out_N\ Zero = Left\ ()\\
&out_N\ (Succ\ n) = Right\ n\\
\\
&const : a \to b \to a\\
&const\ a\ b = a
\end{aligned}$$

Example 2.2 The datatype of lists with elements of type a and the functions in_{L_a} and out_{L_a} are given by the canonical solution to the equation

$() + a \times x \cong x$,

$$
\begin{aligned}
&in_{L_a} : L_a\ (List\ a) \to List\ a \\
&in_{L_a} = const\ Nil \bigtriangledown uncurry\ Cons \\
\\
&out_{L_a} : List\ a \to L_a\ (List\ a) \\
&out_{L_a}\ Nil = Left\ () \\
&out_{L_a}\ (Cons\ a\ as) = Right\ (a, as) \\
\\
&uncurry : (a \to b \to c) \to (a \times b \to c) \\
&uncurry\ f\ (a, b) = f\ a\ b
\end{aligned}
$$

Example 2.3 The canonical solution to the type equation $B_a\ x \cong x$ gives the datatype of leaf-labelled binary trees and the functions in_{B_a} and out_{B_a},

$$
\begin{aligned}
&in_{B_a} : B_a\ (Btree\ a) \to Btree\ a \\
&in_{B_a} = const\ Leaf \bigtriangledown uncurry\ Join \\
\\
&out_{B_a} : Btree\ a \to B_a\ (Btree\ a) \\
&out_{B_a}\ Leaf\ a = Left\ a \\
&out_{B_a}\ (Join\ u\ t) = Right\ (u, t)
\end{aligned}
$$

2.4 Fold

Every datatype has associated a recursive operator called *fold*. This operator is used to represent functions defined by structural recursion.

The function fold, denoted by $fold_F\ (h)$, is defined as the least homomorphism between in_F and any other algebra $h : F\ a \to a$. By homomorphism's definition the following equation is satisfied:

$$fold_F\ (h) \circ in_F = h \circ F\ fold_F\ (h)$$

This equation states that fold consumes the input data structure replacing the constructors by the operations of the algebra h.

Since the function out_F is the inverse of in_F, we can transform the last equation to:

$$fold_F\ (h) = h \circ F\ fold_F\ (h) \circ out_F$$

Now, we show instances of this operator for some datatypes.

Example 2.1 For the datatype of natural numbers the fold operator is defined as:

$$\begin{aligned}
&fold_N\ : (N\ a \to a) \to \quad Nat \to a\\
&fold_N\ (h_1 \triangledown h_2)\ n = \textbf{case}\ n\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Zero \to h_1\ ()\\
&\qquad\qquad\qquad\qquad Succ\ n' \to h_2\ (fold_N\ (h_1 \triangledown h_2)\ n')
\end{aligned}$$

For example, the function that computes the product of two numbers can be defined by:

$$\begin{aligned}
&prod : Nat \to Nat \to Nat\\
&prod\ m = fold_N\ (const\ Zero \triangledown (+m))
\end{aligned}$$

Example 2.2 The fold operator associated to the datatype of lists is defined by:

$$\begin{aligned}
&fold_L\ : (L_a\ b \to b) \to \quad List\ a \to b\\
&fold_L\ (h_1 \triangledown h_2)\ as = \textbf{case}\ as\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Nil \to h_1\ ()\\
&\qquad\qquad\qquad\qquad Cons\ a\ as' \to h_2\ a\ (fold_L\ (h_1 \triangledown h_2)\ as')
\end{aligned}$$

As example we consider the function *length*, which computes the length of a list.

$$\begin{aligned}
&length : List\ a \to Nat\\
&length = fold_L\ (const\ Zero \triangledown Succ \circ inr)
\end{aligned}$$

Example 2.3 The fold operator associated to the datatype of leaf-labelled binary trees is given by:

$$\begin{aligned}
&fold_B\ : (B_a\ b \to b) \to \quad Btree\ a \to b\\
&fold_B\ (h_1 \triangledown h_2)\ t = \ \textbf{case}\ t\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Leaf\ a \to h_1\ a\\
&\qquad\qquad\qquad\qquad Join\ t'\ u \to h_2\ (fold_B\ (h_1 \triangledown h_2)\ t')\ (fold_B\ (h_1 \triangledown h_2)\ u')
\end{aligned}$$

The function that calculates the height of a tree can be defined by:

$$\begin{aligned}
&height \;: Btree\ a \to Nat \\
&height = fold_B\ (const\ Zero \triangledown Succ \circ max) \\
&\qquad\qquad \mathbf{where}\ max\ (x, y) = \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y
\end{aligned}$$

2.5 Unfold

The dual of fold is the *unfold* operator. This operator is used to represent functions defined by structural corecursion[2]. It is denoted by $unfold_F\ (g)$, where g is a F-coalgebra.

The definition of unfold is obtained by dualizing that of fold. As a result we get the following definition:

$$unfold_F(g) = in_F \circ F\ (unfold_F(g)) \circ g$$

This equation states that unfold constructs a data structure by decomposing its arguments using the coalgebra g.

Example 2.1 The unfold operator for the datatype of natural numbers is defined as:

$$\begin{aligned}
&unfold_N \;: (a \to\ N\ a) \to a \to Nat \\
&unfold_N\ g\ a = \mathbf{case}\ (g\ a)\ \mathbf{of} \\
&\qquad\qquad Left\ () \to Zero \\
&\qquad\qquad Right\ a' \to Succ\ (unfold_N\ g\ a')
\end{aligned}$$

Example 2.2 In the case of the datatype of lists, unfold is defined as:

$$\begin{aligned}
&unfold_L \;: (b \to\ L_a\ b) \to b \to List\ a \\
&unfold_L\ g\ b = \mathbf{case}\ (g\ b)\ \mathbf{of} \\
&\qquad\qquad Left\ () \to Nil \\
&\qquad\qquad Right\ (a, b') \to Cons\ a\ (unfold_L\ g\ b')
\end{aligned}$$

[2]A function is defined by structural corecursion when its structure depends on the values produced as result.

For example, the function $enumFromTo$, which takes two numbers a and b and produces the list from a to b, can be defined in terms of $unfold_L$ as:

$$\begin{aligned}
&enumFromTo \ : \ Nat \to Nat \to List\ Nat \\
&enumFromTo\ a\ b = unfold_L\ (\lambda\, i\,.\, \textbf{if}\ i > b\ \textbf{then}\ Left\ () \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ Right\ (i, Succ\ i))\ a
\end{aligned}$$

Example 2.3 For leaf-labelled binary trees the unfold operator is defined as:

$$\begin{aligned}
&unfold_B\ : (b \to\ B_a\ b) \to b \to Btree\ a \\
&unfold_B\ g\ t = \textbf{case}\ (g\ t)\ \textbf{of} \\
&\qquad\qquad Left\ a \to Leaf\ a \\
&\qquad\qquad Right\ (t_1, t_2) \to Join\ (unfold_B\ g\ t_1)\ (unfold_B\ g\ t_2)
\end{aligned}$$

As example we consider the function that maps a function over a tree.

$$\begin{aligned}
&mapt : (a \to b) \to Btree\ a \to Btree\ b \\
&mapt\ f = unfold_B\ (\lambda\, t\,.\, \textbf{case}\ t\ \textbf{of} \\
&\qquad\qquad\qquad Leaf\ a \to Left\ (f\ a) \\
&\qquad\qquad\qquad Join\ (t_1, t_2) \to\ Right\ (t_1, t_2))
\end{aligned}$$

2.6 Hylomorphism

The composition of a fold with an unfold is called a hylomorphism.

Definition 2.12 Given an algebra $h : F\ a \to a$ and a coalgebra $g : b \to F\ b$, an *hylomorphism* is a function $hylo_F\ h\ g\ :\ b \to a$ defined as:

$$hylo_F\ h\ g = fold_F\ h\ \circ\ unfold_F\ g$$

Note that in this definition an intermediate data structure is created. There exists an alternative definition of hylomorphism which avoids the construction of this data structure.

Definition 2.13

$$\begin{aligned}
&hylo_F\ :\ (F\ b \to b) \to (a \to F\ a) \to a \to b \\
&hylo_F\ h\ g = h \circ F\ (hylo_F\ h\ g) \circ g
\end{aligned}$$

The operators fold and unfold are two special cases of hylomorphisms:

$$fold_F\ h = hylo_F\ h\ out_F$$
$$unfold_F\ g = hylo_F\ in_F\ g$$

We can instantiate this operator for any regular datatype. The following examples consider the definition of hylo for some of them.

Example 2.1 For the datatype of lists, hylo is defined as follows:

$$\begin{aligned}
&hylo_L\ : (L_a\ c \to c) \to \quad (b \to L_a\ b) \to b \to c\\
&hylo_L\ (h_1 \triangledown h_2)\ g\ b = \textbf{case}\ (g\ b)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad Left\ () \to h_1\\
&\qquad\qquad\qquad\qquad\qquad Right\ (a, b') \to h_2\ a\ (hylo_L\ (h_1 \triangledown h_2)\ g\ b')
\end{aligned}$$

For example, the function that computes the factorial of a number is given by:

$$\begin{aligned}
&fact\ : Int \to Int\\
&fact = hylo_L\ (const\ 1 \triangledown (*))\ g\\
&\qquad \textbf{where}\ g\ n = \ \textbf{if}\ n < 1\ \textbf{then}\ Left\ ()\\
&\qquad\qquad\qquad\quad \textbf{else}\ Right\ (n, n-1)
\end{aligned}$$

Example 2.2 The hylo operator corresponding to binary trees is defined as follows.

$$\textbf{data}\ Tree\ a = Empty \mid Node\ (Tree\ a)\ a\ (Tree\ a)$$

$$\textbf{type}\ T_a\ b = () + b \times a \times b$$

$$\begin{aligned}
&T_a : (b \to c) \to (T_a\ b \to T_a\ c)\\
&T_a\ f = id + f \times id \times f
\end{aligned}$$

$$\begin{aligned}
&hylo_T\ : (T_a\ b \to c) \to \quad (T_a\ b\ \to T_a\ c)\\
&hylo_T\ (h_1 \triangledown h_2)\ g\ b = \textbf{case}\ (g\ b)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad Left\ () \to h_1\\
&\qquad\qquad\qquad\qquad\qquad Right\ (t_1, a, t_2) \to h_2\ (hylo_T\ (h_1 \triangledown h_2)\ g\ t_1)\ a\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (hylo_T\ (h_1 \triangledown h_2)\ g\ t_2)
\end{aligned}$$

As an example we consider an implementation of the algorithm for sorting lists known as Quicksort. It can be written as the composition of a fold and an unfold, in which we first construct a binary search tree with the elements of the input list, and then flatten it into an ordered list by an in-order traversal.

$$
\begin{aligned}
&\mathit{quicksort} \;:\; (\mathit{Ord}\ a) \Rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \\
&\mathit{quicksort} = \mathit{flatten} \circ \mathit{mktree}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{flatten} \;:\; \mathit{Tree}\ a \rightarrow \mathit{List}\ a \\
&\mathit{flatten}\ \mathit{Empty} = \mathit{Nil} \\
&\mathit{flatten}\ (\mathit{Node}\ t_1\ x\ t_2) = \mathit{flatten}\ t_1 \;+\!\!+\; (\mathit{Cons}\ x\ \mathit{Nil}) \;+\!\!+\; \mathit{flatten}\ t_2
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mktree} \;:\; \mathit{List}\ a \rightarrow \mathit{Tree}\ a \\
&\mathit{mktree}\ \mathit{Nil} = \mathit{Empty} \\
&\mathit{mktree}\ (\mathit{Cons}\ a\ \mathit{as}) = \;\; \mathit{Node}\ (\mathit{mktree}\ \mathit{xs})\ a\ (\mathit{mktree}\ \mathit{ys}) \\
&\qquad\qquad\qquad\qquad\quad \mathbf{where}\ (xs, ys) = \mathit{split}\ (\leq a)\ \mathit{as}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{split} \;:\; (a \rightarrow \mathit{Bool}) \rightarrow \mathit{List}\ a \rightarrow \mathit{List}\ a \times \mathit{List}\ a \\
&\mathit{split}\ p\ \mathit{Nil} = (\mathit{Nil}, \mathit{Nil}) \\
&\mathit{split}\ p\ (\mathit{Cons}\ a\ \mathit{as}) = \;\; \mathbf{if}\ p\ a\ \mathbf{then}\ \ (\mathit{Cons}\ a\ xs, ys) \\
&\qquad\qquad\qquad\qquad\quad \mathbf{else}\ (xs, \mathit{Cons}\ a\ ys) \\
&\qquad\qquad\qquad\qquad\quad \mathbf{where}\ (xs, ys) = \mathit{split}\ p\ \mathit{as}
\end{aligned}
$$

Given these definitions we can get the usual definition of quicksort:

$$
\begin{aligned}
\mathit{quicksort} =&\, \mathit{hylo}_T\ (\mathit{const}\ \mathit{Nil} \triangledown h)\ g \\
&\mathbf{where} \qquad h\ xs\ a\ ys = xs \;+\!\!+\; (\mathit{Cons}\ a\ \mathit{Nil}) \;+\!\!+\; ys \\
&\qquad\qquad\quad\; g\ \mathit{Nil} = \mathit{Left}\ () \\
&\qquad\qquad\quad\; g\ (\mathit{Cons}\ a\ \mathit{as}) \;=\; \mathit{Right}\ (zs, a, ws) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{where}\ (zs, ws) = \mathit{split}\ (\leq a)\ \mathit{as}
\end{aligned}
$$

2.7 Type functors

A parameterized functor F_a is obtained by fixing the first argument of a bifunctor F with a type A. We say that the datatype defined as least fixed point of the functor F_a is parameterized by type A.

For each parameterized datatype, we can define a type functor, denoted by D in the following way.

$$\begin{aligned} D\ a &= \mu_{F_a} \\ D\ f &= \mathit{fold}_F\ (\mathit{in}_F \circ F\ f\ \mathit{id}) \end{aligned}$$

Example 2.1 As an example we consider the list type functor defined by:

$$\begin{aligned} D\ a &= \mathit{List}\ a \\ D\ f &= \mathit{fold}_L\ (\mathit{const}\ \mathit{Nil} \bigtriangledown (\lambda\, a\, b\, .\, \mathit{Cons}\ (f\ a)\ \mathit{bs})) \end{aligned}$$

This corresponds to the well-known map function for lists that is present in most functional programming languages.

3 Shortcut Deforestation

The deforestation algorithm proposed in [GLJ, Gil96] consists basically in standardizing the way in which lists are consumed and produced by two functions `foldr` and `build`. The function `foldr` is well known in functional programming. It corresponds to the fold operator for lists. That is,

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f n [] = n
foldr f n (x:xs) = f x (foldr f n xs)
```

The function `build` is defined as:

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

The main thing about this function is that it produces a list using the constructors `(:)` and `[]` only.

If we build and consume lists using these functions the `foldr/build` rule can be applied to eliminate the intermediate list that is created in their composition.

Law 3.1 (foldr/build rule)
If for some fixed type `a` we have

```
g: forall b . (a -> b -> b ) -> b -> b
```

then

```
foldr k z ( build g ) = g k z
```

The polymorphic restriction on `g` guarantees that `g` constructs its result only using its two arguments. That way, we ensure that `build` does not use other functions than the list constructors to produce a list as result.

The proof of this law is given by the "free theorem" associated with `g`'s type ([Wad89]).

Proof

The "free theorem" for `g's` type is that: For all types `b` and `b'`, and functions `f: a -> b -> b`, `f': a -> b' -> b'`, and (a strict) `h: b -> b'` the following rule holds:

```
(forall x:a . forall y:b . h (f x y) = f' x (h y))
=> (forall y:b . h (g f y) = g f'(h y))
```

We instantiate this rule by taking `h = foldr k z`, `f = (:)`, and `f' = k`, obtaining:

```
(forall x:a . forall y:b . foldr k z (x:y) = k x (foldr k z y))
=> (forall y:b . foldr k z (g (:) y) = g k (foldr k z y))
```

The left hand side holds vacuously since it is a consequence of the definition of `foldr`. Therefore, the rule reduces to:

```
forall y:b . foldr k z (g (:) y) = g k (foldr k z y)
```

If we let `y = []` and apply `foldr`'s definition, we finally get:

```
foldr k z (g (:) []) = g k z
```

which is the result we wanted to prove. □

As an example of the action of this rule we consider the following function taken from [Gil96].

```
sumFromTo :: Int -> Int -> Int
sumFromTo = sum . enumFromTo

  where sum [] = 0
        sum (x:xs) = x + sum (xs)

enumFromTo n m = if (n>m) then []
                 else  n: enumFromTo (n+1) m
```

First, we rewrite the list consumption function `sum` in terms of `foldr`, and the list production function `enumFromTo` in terms of `build`.

```
sum = foldr (+) 0


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

Then, we apply the `foldr/build` rule and β-reductions to get:

```
sumFromTo a b =
  let
    enumFromTo' a b = if a>b then 0
                      else a + (enumFromTo' (a+1) b)
  in enumFromTo' a b
```

As we can see, this definition of `sumFromTo` does not use any intermediate list.

In order to apply this rule automatically, some functions of the standard libraries of Haskell have been defined using `foldr` and `build`. (see [GLJ])

3.1 The destroy/unfoldr transformation

The dual rule of `foldr/build` was defined in [Sve02]. The rule, called `destroy/unfoldr`, is based on the functions `unfoldr` and `destroy` which produce and consume a list respectively.

The function `unfoldr` corresponds to the unfold operator for lists. That is:

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f b = case f b of
```

```
                   Nothing -> []
                   Just (a, b') -> a: unfoldr f b'
```

Here the function `f` is applied to `b` to determine whether we shall produce one more element of the output list or return the empty list.
The definition of `destroy` is:

```
destroy :: (forall a. (a -> Maybe (b, a)) -> a -> c) -> [b] -> c
destroy g xs = g listpsi xs
    where listpsi :: [a] -> Maybe (a, [a])
          listpsi [] = Nothing
          listpsi (x:xs) = Just (x, xs)
```

The main thing about this function is the first argument of g, the function `listpsi`, which is used to inspect the list. We ensure that `g` consumes a list by doing pattern matching on it using the function `listpsi`, through which `g` can create a kind of view of the list.

The `destroy/unfoldr` rule is stated as:

Law 3.2 (destroy/unfoldr rule)
If for some fixed type `b` and `c` we have:

```
g :: forall a. (a -> Maybe(b, a)) -> a -> c
```

then

```
destroy g (unfoldr f e) = g f e
```

As with the `foldr/build` law, this law also requires a polimorphic restriction on `g`. This restriction guarantees that `g` only uses its first argument to inspect a list.

We prove this law by proving its generalization to any algebraic datatype in the next section.

3.2 Acid Rain

The acid rain law was proposed by Takano and Meijer [TM95] with the aim to generalize the `foldr/build` rule to any algebraic datatype. They obtained this theorem by replacing in the `foldr/build` rule the function `foldr` by an arbitrary fold, the constructors `(:)` and `[]` by the initial algebra in_F, and the arguments `k` and `z` by an arbitrary F-algebra.

Law 3.3 (Acid Rain: Catamorphism)

$$\begin{array}{c} g : \forall b\ .\ (F\ b \to b) \to a \to b \\ \Rightarrow \mathit{fold}_F\ (h)\ \circ\ (g\ in_F)\ =\ g\ h \end{array}$$

where a is some fixed type.

Proof

The "free theorem" for `g's` type is that: For all types a and a', and functions $h : F\ a \to a$, $h' : F\ a' \to a'$, and (a strict) $f' : a \to a'$ the following rule holds:

$$f'\ \circ\ h\ =\ h'\ \circ\ (F\ f') \Rightarrow f'\ \circ\ (g\ h)\ =\ g\ h'$$

We instantiate this result by taking $f' = \mathit{fold}_F\ (f)$, $h = in_F$, and $h' = f$, obtaining:

$$\mathit{fold}_F\ (f)\ \circ\ in_F\ =\ f\ \circ\ (F\ \mathit{fold}_F\ (f)) \Rightarrow \mathit{fold}_F\ (f)\ \circ\ (g\ in_F)\ =\ g\ f$$

The left hand side of this rule holds because fold satisfies its defining fixed point equation. Therefore, we obtain the desired result. $\Box$

This law corresponds to the most general version of Acid Rain, in the sense that function g has an extra parameter. The addition of this parameter permits to express the law on the function level, where we can take the dual of it.

The dual of this rule is the following law:

Law 3.4 (Acid Rain: Anamorphism)

$$
\begin{gathered}
h : \forall b \,.\, (b \to F\ b) \to b \to a \\
\Rightarrow (h\ out_F) \ \circ\ unfold_F(g) \ = \ h\ g
\end{gathered}
$$

where a is some fixed type.

We prove this law as a free theorem in the same way as the previous one.

Proof

The "free theorem" for `h's` type is that: For all types a and a', and functions $f : a \to F\ a$, $f' : a' \to F\ a'$, and $g' : a \to a'$ the following rule holds:

$$F\ g' \ \circ\ f \ = \ f' \ \circ\ g' \ \Rightarrow \ h\ f \ = \ (h\ f') \ \circ\ g'$$

We instantiate this result by taking $g' = unfold_F\ (h)$, $f = h$ and $f' = out_F$, obtaining:

$$F\ unfold_F\ (h) \ \circ\ g \ = \ out_F \ \circ\ unfold_F\ (h) \ \Rightarrow \ h\ g \ = \ (h\ out_F) \ \circ\ unfold_F\ (h)$$

The premise of this rule is the fixed point definition of unfold. Therefore, we obtain the desired result. □

3.3 The foldr/augment transformation

The function `build` can not always be used to represent list producers functions. For example, consider the following definition of append:

```
xs ++ ys = foldr (:) ys xs
```

If we try to express append in terms of build, by abstracting `(:)` and `[]`, we obtain this:

```
xs ++ ys = build g
   where g c n = foldr c ys xs
```

This definition is ill-typed, since the type of `g` is not general enough. The problem is that the cons cells in the list `ys` are not abstracted, so `ys` has type `list` instead of `b` (see 3.1).

To solve this problem, Gill [Gil96] introduces the `augment` function, which generalizes `build` by abstracting the constructor `[]` in a list:

```
augment :: (forall b. (a -> b -> b) -> b -> b) -> [a] -> [a]
augment g ys = g (:) ys
```

Now, the append function can be defined in terms of `augment` by,

```
xs ++ ys = augment g ys
  where g c n = foldr c n xs
```

The `build` function can be expressed in terms of augment as:

```
build g = augment g []
```

The `foldr/build` rule generalizes to `foldr/augment`:

Law 3.5 (foldr/augment rule)
If for some fixed type `a` we have

```
g:: forall b . (a -> b -> b ) -> b -> b
```

then

```
foldr k z ( augment g ys) = g k (foldr k z ys)
```

The generalization of `foldr/augment` to arbitrary inductive datatypes is given in the paper [GUV04]. This defines this law using an alternative semantic of inductive types.

The augment operator can be defined for any inductive datatype by abstracting all non-recursive constructor in build's definition. As an example, we condider the augment operator for leaf-labelled binary trees:

```
augmentB :: (forall b. (b -> b -> b) -> (a -> b) -> b) -> Btree a
            -> Btree a
augmentB g f = g Join f
```

The following function corresponds to the fold operator for leaf-labelled binary trees:

```
foldB :: (b -> b -> b) -> (a -> b) -> Btree a -> b
foldB j l (Leaf a) = l a
foldB j l (Join u t) = j (foldB j l u) (foldB j l t)
```

The foldB/augmentB rule is defined as:

Law 3.6 (foldB/augmentB rule)
If for some fixed type `a` we have

```
g:: forall b . (b -> b -> b) -> (a -> b) -> b
```

then

```
foldB j l (augmentB g f) = g j (foldB j l . f)
```

4 Monads

Monads are used in functional programming for describing a wide range of programming language features such as global state, exceptions, I/O and non-determinism. The concept of monad comes originally from category theory. It was proposed by Moggi [Mog91] as a devise to structure denotational semantics descriptions of programming languages, and then applied by Wadler [Wad95] to structure functional programs.

In this section, we briefly introduce monads. We will give its definition in the context of functional programming.

4.1 Definition

A monad is defined as a *kleisli triple* `(m, return, >>=)` where:

- `m` is a functor
- `return : a -> m a` is a polimorphic function, and
- `(>>=): m a -> (a -> m b) -> m b` is a polimorphic operator, called *bind*.

Monads permit to make a distinction between values and computations. An object of type `m a` is usually understood as a computation that yields a value of type `a`. In functional programming a computation may raise exceptions, act on states, generate outputs, or produce any other effect.

A function of type `a -> m b` is called a *monadic* function. This can be read as a function that takes a value of type `a` and returns a value of type `b`, with a possible additional effect captured by `m`.

The function `return` can be interpreted as the inclusion of values into computations that yield these values without producing any effect. The bind operator specifies the way in which computations can be sequenced. The expression `m >>= \x -> m'` can be read as: evaluate computation `m`, bind the resulting value to the variable `x`, and then evaluate `m'` passing the effect around. The way in which effects are propagated throughout computations depends on the specific monad.

The components of a kleisli triple `(m, return, >>=)` must satisty the following three laws:

1. *Right identity* `m >>= \x -> return  =  m`
2. *Left identity* `return a >>= \x -> m  =  m[x:=a]`
3. *Associativity* `(m >>= \x -> m') >>= \x ->  m'' =`
   `m >>= \x -> (m' >>= \x -> m'')`

4.2 Monads in Haskell

Monads can be defined in Haskell using classes.

```
class Monad m where
   return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```

When instantiating this class with particular definitions of `return` and `(>>=)` one must check that they satisfy the mondad laws given above.

Another useful operator for sequencing two computations is the following:

```
(>>) :: Monad m => m a -> m b -> m b
m >> m' = m >>= \_ -> m'
```

In the expression `m >> m'` the result yielded by `m` is not used by the computation `m'`, only the effect is passed around.

Example 4.1 (Identity) The trivial monad is the *identity* monad, defined as:

```
data Id a = I a
```

```
instance Monad Id where
   return a = I a
   (I a) >>= f = f a
```

Example 4.2 (Exception) The *exception* monad is defined to model programs with exceptions.

```
data Exc a = Ok a | Fail Exception
type Exception = ...

instance Monad Exc where
   return a = Ok a
   (Ok a) >>= f = f a
   (Fail e) >>= f = Fail e
```

The function `return` simply takes a value and returns it. The bind operator propagates an exception if the first argument is an exception, or applies the second argument to the value returned by the first one, in other case.

When there is a unique exception value, the Maybe monad is used instead of the exception monad.

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
   return a = Just a
   (Just a) >>= f = f a
   (Nothing) >>= f = Nothing
```

Example 4.3 (State) The *state* monad models programs that handle a state. In this monad a computation is a state transformer that accepts an initial state and returns a value and a final state.

```
data State s a = State (s -> (a, s))
```

```
instance Monad State where
   return a = State (\s -> (x, s))
   (State m) >>= f = State (\s -> let (s', a) = m s
                                       State m' = f a
                                  in m' s' )
```

The function `return` takes a value and returns a computation that yields this value, leaving the state unchanged. The bind operator applies the first computation to an initial state, yielding a value `a` and a state `s'`, and then applies the second computation to `a` and `s'`.

Example 4.4 (Output) The *Output* monad was defined to model programs which generate outputs. In this monad, a computation consists of the output generated paired with the value returned. We define the Output monad for outputs of type string.

```
data Output a = C (String, a)

instance Monad Output where
   return a = C ("", a)
   C (x,a) >>= f let C (y, b)= f a in
             (x ++ y, b)
```

The function `return` takes a value and returns no output paired with this value. The bind operator applies the two computations and returns as output the concatenation of the outputs produced by each of the computations.

4.2.1 The do notation

Haskell provides an alternative notation for writing monadic programs, called the do notation.

In general, programs written with this notation are easier to read than those written using the operators `(>>=)` and `(>>)`. The syntax of a do expression is defined by the following rules:

```
do {x <- m; m'} = m >>= \x -> do {m'}
do {m; m'} = m >>= do {m'}
do {m} = m
```

5 Monadic Shortcut Deforestation

In this section we present a monadic generalization of the acid rain law and its dual which will permit us to apply deforestation to programs with effects.

The programs we want to fuse are written as the composition of a monadic function, that produces or consumes a data structure, with a recursive function. We will consider the cases in which this recursive function is a fold or an unfold.

5.1 Monadic Acid Rain

To obtain a definition of this law for the monadic case we think of the function g in the acid rain law as an effect-producing function and then we derive the "free theorem" associated with its type. That is, we consider a function g with type:

$$g : \forall\, a\, .\, (F\ a \rightarrow a) \rightarrow c \rightarrow m\ a$$

where c is some fixed type.

The "free theorem" associated with g's type states that: for all types b and b', and functions $f : F\ b \rightarrow b$, $f' : F\ b' \rightarrow b'$ and $h' : b \rightarrow b'$, the following implication holds:

$$h' \circ f = f' \circ F\ h' \ \Rightarrow\ (m\ h') \circ (g\ f) = g\ f'$$

We instantiate this theorem by taking $h' := \mathit{fold}_F\ (h)$, $f := \mathit{in}_F$ and $f' := h$, obtaining:

$$\mathit{fold}_F\ (h) \circ \mathit{in}_F = h \circ F\ \mathit{fold}_F\ (h) \ \Rightarrow\ (m\ \mathit{fold}_F\ (h)) \circ (g\ \mathit{in}_F) = g\ h$$

The premise of this rule vacuously holds because *fold* satisfies its defining fixed point equation.

Finally, we define the monadic acid rain law as the conclusion of this rule:

Law 5.1 *(Monadic acid rain: fold)* Let c be some fixed type,

$$\begin{aligned} & g : \forall a \,.\, (F\ a \to a) \to c \to m\ a \\ \Rightarrow\ & (m\ fold_F\ (h)) \circ (g\ in_F) = g\ h \end{aligned}$$

5.2 Monadic Acid Rain (dual)

To obtain a definition of the dual of a monadic acid rain law we proceed in a similar way as before. First, we define the type of the polymorphic function g as:

$$g : \forall\ a \,.\, (a \to F\ a) \to a \to m\ c$$

The next step is to consider the "free theorem" associated with g's type. That is: for all types b and b', and functions $f : b \to F\ b$, $f' : b' \to F\ b'$ and $h' : b \to b'$, the following rule holds:

$$F\ h' \circ f \ = \ f' \circ h' \ \Rightarrow\ g\ f \ = \ (g\ f') \circ h'$$

We instantiate this theorem by taking $h' := unfold_F\ (h)$, $f := h$ and $f' := out_F$, obtaining the following:

$$F\ unfold_F\ (h) \circ h \ = \ out_F \circ unfold_F\ (h) \ \Rightarrow\ g\, h \ = \ (g\, out_F) \circ unfold_F\ (h)$$

By definition of unfold the premise of this rule is vacuously true. Therefore, we get the following law.

Law 5.2 *(Monadic acid rain: unfold)* Let c be some fixed type,

$$\begin{aligned} & g : \forall a \,.\, (a \to F\ a) \to a \to m\ c \\ \Rightarrow\ & g\ f \ = \ (g\ out_F) \circ unfold_F\ (f) \end{aligned}$$

5.3 Relationship with other fusion laws

We note that function $m\ fold_F\ (f)$ in law 5.1 does not produce any effect. This may be seen as a drawback since it means that we won't be able to apply deforestation to the composition of two effectful functions. An example of a transformation for such a situation was given by Meijer and Jeuring [MJ]. They defined a rule for eliminating intermediate lists in the composition of monadic programs. Their rule can be generalized to other algebraic datatypes, but important restrictions need to be added in order to make the rule applicable.

Law 5.1 can be specialized to obtain the fusion laws *mhylo-fold* and *mfold-fold* presented in [Par01, Par05], by instantiating the polimorphic function g with particular functions called monadic hylomorphisms. In a similar way, law 5.2 can be instantiated to obtain fusion laws relating monadic hylomorphisms with unfolds, for instance the laws *unfold-mhylo* and *unfold-unfold*.

Finally, comparing law 5.2 with law 3.4, the difference between them is only the type of the function g. In law 5.2, g must return a computation of type $m\ c$ while in law 3.4, it returns a value of type c. Since there is no restriction in how we choose c in law 3.4, we can conclude that law 5.2 is a particular case of the other. As a consequence of this fact, the laws *unfold-mhylo* and *unfold-unfold* are also instances of law 3.4.

5.4 Case study: A monadic Interpreter

In this section we present an example of a simple interpreter for arithmetic expressions that illustrates the application of the monadic acid rain law. We construct an interpreter in two phases. First we build a monadic parser of simple expressions and then we define a function which evaluates the result of this parser.

5.4.1 Monadic Parser

We start by defining a type for parsers:

$$\textbf{newtype}\ Parser\ a = Parser\ (String \to list\ (a \times String))$$

That is, a parser of type a is a function that takes a state representing the string to be parsed, and returns a list containning a value of type a parsed from the string, and a state representing the remaining string to be parsed. The convention is that the empty list denotes failure of a parser, and a non-empty list denotes success. We say that a parser is deterministic if it allways returns an empty list or a singleton list.

The type constructor $Parser$ can be made into an instance of the class $Monad$ as follows:

$$\begin{aligned}
&apply : Parser\ a \to (String \to list\ (a \times String))\\
&apply\ (Parser\ p) = p
\end{aligned}$$

$$\begin{aligned}
&\textbf{Instance}\ Monad\ Parser\ \textbf{where}\\
&\quad return\ a\ \ = Parser\ (\lambda\ xs\ .\ Cons\ (a, xs)\ Nil)\\
&\quad p >>= f\ \ = Parser\ (\lambda\ xs\ .\ concat\ (map\ g\ (apply\ p\ xs)))\\
&\qquad\qquad\qquad \textbf{where}\ g\ (a, b) = apply\ (f\ a)\ b
\end{aligned}$$

This means that $return$ takes a value and returns this without consuming any input. The bind operator is a sequencing operator for parsers. The parser $p >>= f$ takes an input string xs and first applies p to xs to give a list of pairs of the form (a_i, as_i), then apply the parser $f\ a_i$ to each string as_i. The result is a list of lists that is concatenated to give the final list of results.

Let us start by defining some primitive parsers.

One of them is the parser $item$, which returns the first character of the input string and fails if the input is the empty list.

$$
\begin{aligned}
&item : Parser\ Char\\
&item = Parser\ (\lambda\ xs\ .\ \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad Nil \to Nil\\
&\qquad\qquad\qquad\qquad Cons\ a\ xs' \to Cons\ (a, xs')\ Nil)
\end{aligned}
$$

The parser $zero$ takes an input string and always fail.

$$
\begin{aligned}
&zero : Parser\ a\\
&zero = Parser\ (\lambda\ xs\ .\ Nil)
\end{aligned}
$$

The following parser is used for consuming certain specific characters. Its takes a predicate and yields a parser that returns the first character of the input string if it satisfies the predicate, and fails otherwise.

$$
\begin{aligned}
&sat : (Char \to Bool) \to Parser\ Char\\
&sat\ p = item\ \ >>=\ \ (\lambda\ x\ .\ \ \mathbf{if}\ p\ x\ \mathbf{then}\ return\ x\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ zero)
\end{aligned}
$$

Using sat, we define two parsers for specific characters and single digits.

$$
\begin{aligned}
&char : Char \to Parser\ Char\\
&char\ x = sat\ (\lambda\ y\ .\ x == y)\\
\\
&digit : Parser\ Char\\
&digit = sat\ (\lambda\ x\ .\ x \geq' 0'\ \wedge\ x \leq' 9')
\end{aligned}
$$

To form more useful parsers we will use the following parser combinator. We define the operator $<|>$, which takes two parsers p_1 and p_2 and computes as follows: applies the parser p_1, if it fails then applies p_2.

$$
\begin{aligned}
&(<|>) : Parser\ a \to Parser\ a\ \to Parser\ a\\
&p_1\ <|>\ p_2 = Parser\ (\lambda\ s\ .\ \mathbf{let}\ rs = (apply\ p_1)\ s\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ null\ rs\ \mathbf{then}\ (apply\ p_2)\ s\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ rs)
\end{aligned}
$$

5.4.2 Syntax Analizer

We will consider a parser for simple arithmetic expressions given by the following datatype declaration:

$$\textbf{data}\ Exp ::= Add\ Exp\ Exp \mid Num\ Int$$

The structure of this datatype is capture by the functor E defined by:

$$\textbf{type}\ E\ a = Int + Exp \times Exp$$

$$\begin{array}{l}
E : (a \to b) \to (E\ a \to E\ b) \\
F\ f = id + f \times f
\end{array}$$

A sintax analizer for expressions of type Exp is given by the following function:

$$\begin{array}{lll}
syntax_E : Parse\ r\ Exp & & \\
syntax_E = do\ \{ & n \leftarrow & digit; \\
 & do\ \{ & char\ '+'; \\
 & & e \leftarrow syntax_E; \\
 & & return\ (Add\ (Num\ (ord\ n))\ e)\ \} \\
 & <|> & return\ (Num\ n)\ \}
\end{array}$$

This function constructs a value of type Exp by examining the string containing in the state. Since this string is not an argument of the function, we can not represent this directly as a monadic build. First, we need to regard this function as a function of type $\underline{1} \to Parser\ Exp$. Doing so, and by abstracting the constructors Num and Add from the definition of $syntax_E$, we obtain this:

$$buildm_E : (\forall a\ .\ (E\ a \to a) \to \underline{1} \to m\ a) \to \underline{1} \to Exp$$
$$buildm_E\ g = g\ (Num \triangledown Add)$$

$$syntax_E : \underline{1} \to Parser\ Exp$$
$$syntax_E = \ buildm_E\ gAdd$$

$$\begin{array}{llll} gAdd\ (h_1 \triangledown h_2)\ i = do\ \{ & n \leftarrow & digit; & \\ & do\ \{ & char\ '+'; & \\ & & e \leftarrow syntax_E\ i; & \\ & & return\ (h_2\ (h_1\ (ord\ n))\ e)\ \} & \\ & <|> & return\ (h_1\ n)\ \} & \end{array}$$

5.4.3 Evaluation

The last phase of the construction of this monadic interpreter is the evaluation of an expression. In this case evaluating an expression will yield a number.

We consider the following function which evaluates an arithmetic expression.

$$\begin{array}{l} eval : Exp \to Int \\ eval\ (Num\ n) = n \\ eval\ (Add\ e_1\ e_2) = (eval\ e_1)\ +\ (eval\ e_2) \end{array}$$

We can define this function in terms of $fold_E$ as:

$$eval = fold_E\ (id \triangledown (+))$$

$$\begin{array}{l} fold_E\ : (E\ a \to a) \to\ Exp \to a \\ fold_E\ (h_1 \triangledown h_2)\ e = \textbf{case}\ e\ \textbf{of} \\ \qquad\qquad Num\ n \to h_1\ n \\ \qquad\qquad Add\ e_1\ e_2 \to h_2\ (fold_E\ (h_1 \triangledown h_2)\ e_1)\ (fold_E\ (h_1 \triangledown h_2)\ e_2) \end{array}$$

Now, we can build an interpreter of arithmetic expressions by composing the function $syntax_E$ with $eval$ as follows:

$$\begin{aligned} &interpreter : \underline{1} \rightarrow Parser\ Int \\ &interpreter = m\ eval \circ syntax_E \end{aligned}$$

Inlining the definitions of $eval$ and $syntax_E$ we obtain the following definition:

$$interpreter = m\ fold_E\ (id \triangledown (+)) \circ (buildm_E\ gAdd)$$

We can now apply the monadic acid rain law to transform this definition into a more efficient one, which does not generate any intermediate data structure:

$$interpreter = gAdd\ (id \triangledown (+))$$

6 Implementation

In this section we present an implementation of the monadic acid rain laws in the Glasgow Haskell Compiler (GHC), obtained by using the *rule pragma* of GHC [JTH01].

6.1 Rewrite Rules in GHC

We start with a brief overview of the rewrite rule mechanism implemented in GHC. The basic idea of this mechanism is to allow the programmer to express rewrite rules that will be taken into account by the compiler. Rewrite rules are expressed using Haskell syntax and must be enclosed in a pragma, which is an expression between brackets of the form `{-# ... #-}` . This convention allows a non-optimizing compiler to ignore the rules.

As an example, consider the following rule which is used to replace the expression `map f (map g xs)` by the equivalent one `map (f.g) xs`.

```
{-# RULES
    "map/map" forall f g xs .
map f (map g xs) = map (f . g) xs
#-}
```

Throughout a compilation, GHC tries to spot instances of the left hand side of a rule, and rewrites them into the right hand side.

The form of a rule has a restriction. The left hand side of it must be a function application.

6.2 The mmap/foldr/buildm rule

We want to add rewrite rules that implement instances of the law 5.1 on particular datatypes.

For example, the instance of law 5.1 for lists gives rise to the rule `mmap/foldr/buildm`.

```
mmap :: Monad m => ( a -> b) -> (m a -> m b)
mmap f m = m >>= \a -> return (f a)

buildm :: Monad m => (forall b. (a -> b ->b) -> b -> m b) -> m [a]
buildm g = g (:) []

{-# RULES "mmap/foldr/buildm"
    forall k z (g :: Monad m => (forall b. (a -> b -> b) -> b -> m b)) .
        mmap (foldr k z) (buildm g) = g k z
#-}
```

The function `buildm` is the monadic version of `build` (see section 3.1). This function constructs a list by applying a monadic function `g` to the list constructors `(:)` and `[]`. The rule states that, when `(foldr k z)` is applied to the value returned by `buildm`, we can eliminate the intermediate list by applying `g` to `k` and `z`, provided that `g` has type `forall b.(a -> b -> b) -> b -> m b`. The presence of `mmap` means that the effect produced by `buildm` is simply propagated.

To use the rule `mmap/foldr/buildm` for removing intermediate lists we must express monadic list generating functions in terms of `buildm` and list consuming functions in terms of `foldr`. In figures 1 and 2 we show some functions taken from standard libraries defined in terms of `buildm` and `foldr`.

6.2.1 Restriction

One can find a restriction with the application of the rule `mmap/foldr/buildm`. A programmer who don't know the existence of this rule could write his programs using a do notation instead of the function `mmap`.

We would like to express the rule `mmap/foldr/buildm` using a much more common way to write monadic programs:

```
{-# RULES "mmap/foldr/buildm"
    forall k z (g :: Monad m => (forall b. (a -> b -> b) -> b -> m b)) .
        do { xs <- buildm g ; return (foldr k z xs)} = g k z
```

```
zipWithM :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys  = buildm (\c n ->
                      let zip' (x:xs') (y:ys') = do p <- f x y
                                                    zs<- zip' xs' ys'
                                                    return (c p zs)
                          zip' _ _ = return n
                      in zip' xs ys )

getLine :: IO String
getLine = buildm (\c n ->  do x <- getChar
                              if x == '\n' then return n
                              else do xs <- getL' c n
                                      return (c x xs) )

accumulate :: Monad m => [m a] -> m [a]
accumulate xs = buildm (\c n -> let f xs = case xs of
                                        []-> return n
                                        (x:xs')-> do y <- x
                                                     ys<- f xs'
                                                     return (c y ys)
                                in f xs )

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f xs = buildm (\c n -> let g xs = case xs of
                                        []-> return n
                                        (x:xs')-> do y <-  f x
                                                     ys <- g xs'
                                                     return (c y ys)
                            in g xs )

hGetContents :: Handle -> IO String
hGetContents h = buildm (\c n -> do eof <- hIsEOF h
                                    if eof then hClose h >> return n
                                    else do x <- hGetChar h
                                            xs <- hgetC c n
                                            return (c x xs) )
```

Figure 1: Definitions of some standard functions using buildm

```
accumulate :: Monad m => [m a] -> m [a]
accumulate = foldr (\ma mb -> do x <- ma
                                 xs<- mb
                                 return (x:xs))
                   (return [])

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = foldr (\x xs -> (f x) ++ xs) []

length :: [a] -> Int
length = foldr (\x n -> n+1) 0

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f = foldr (\a mb -> do y <- f a
                            ys<- mb
                            return (y:ys))
               (return [])

product :: Num a => [a] -> a
product = foldr (*) 1

sum :: Num a => [a] -> a
sum = foldr (+) 0

and :: [Bool] -> Bool
and = foldr (&&) True

unlines :: [String] -> String
unlines = foldr (\ x y -> x++"\n"++y) []

map :: (a -> b) -> [a] -> [b]
map f = foldr (\ x y -> f x : y) []
```

Figure 2: Definitions of some standard functions using foldr

```
#-}
```

However, this new form of writing the rule can not be added to GHC. The problem is that the left hand side of the rule is not a function application. One alternative could be the addition of a preprocessing that applies a syntactic transformation to programs in order to replace some uses of the do-notation by `mmap`.

6.3 The destroy/unfoldr rule

Instead of defining a rule for the dual of monadic acid rain law, we use the rule `destroy/unfoldr`, which is the implementation of law 3.4 for lists. The rewrite rule associated with this transformation is written as:

```
{-# RULES "destroy/unfoldr"
    forall k e (g :: forall a. (a -> Maybe(b, a)) -> a -> c) .
        destroy g (unfoldr f e) = g f e
#-}
```

This rule states that when `destroy` consumes the result of applying `unfoldr` to `f` and `e` we can eliminate the intermediate list by applying `g` to `f` and `e`, provided that g has type `forall a.(a -> Maybe(b, a)) -> a -> m c`.

Since we are interested in the application of the rule `destroy/unfoldr` to monadic programs, we only write list consuming functions that produces effects in terms of destroy.

Figure 3 and 4 show some functions of standard libraries defined in terms of `destroy` and `unfoldr`.

6.4 Case Studies

We present two small examples that illustrate how deforestation is carried out. In each of them we measure efficiency improvements obtained by the application of deforestation.

```
putStr :: String ->IO ()
putStr xs = destroy putS xs
 where putS f xs = case f xs of
                     Nothing -> return []
                     Just (c,cs) -> putChar c >> putS f cs

foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
foldM f z xs = destroy foldM' xs
 where foldM' f xs = case f xs of
                       Nothing -> return z
                       Just (x, ys) -> f z x >> foldM' f ys

accumulate :: Monad m => [m a] -> m [a]
accumulate xs = destroy acc xs
 where acc f xs    = case f xs of
                       Nothing -> return []
                       Just (c, cs)-> do y <- c
                                         ys<- acc f cs
                                         return (y:ys)

mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f xs = destroy mapM' xs
 where mapM' f xs = case f xs of
                       Nothing -> return []
                       Just (c, cs)-> do y <- f c
                                         ys<- mapM' f cs
                                         return (y:ys)

hPutStr :: Handle -> String -> IO ()
hPutStr h xs  = destroy hPutS xs
 where hPutS f xs = case f xs of
                      Nothing -> return ()
                      Just (c,cs) -> hPutChar h c >> hPutS f cs
```

Figure 3: Definitions of some standard functions using destroy

```
init :: [a] -> [a]
init = unfoldr (\xs -> case xs of
                         [x] -> Nothing
                         (x:xs') -> Just(x,xs'))

iterate :: (a -> a) -> a -> [a]
iterate f = unfoldr (\x -> Just(x, f x))

replicate :: Int -> a -> [a]
replicate n x = unfoldr (\y -> if y==0 then Nothing
                               else Just (x, y-1)) n

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p    = unfoldr (\xs ->
                   case xs of
                     [] -> Nothing
                     (x:xs') -> if p x then Just(x, xs')
                                else Nothing)

zip :: [a] -> [b] -> [(a,b)]
zip xs ys = unfoldr (\(ms, ns) ->
              case ms of
                [] -> Nothing
                (x:ms') -> case ns of
                             [] -> Nothing
                             (y:ns') -> Just ((x,y), (ms', ns')))
                    (xs,ys)

zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith  f xs ys = unfoldr (\(ms, ns) ->
                     case ms of
                       []-> Nothing
                       (x:ms')-> case ns of
                                   []-> Nothing
                                   (y:ns')-> Just (f x y, (ms', ns')))
                           (xs,ys)

enumFromTo x y    = unfoldr (\i -> if i>y then Nothing
                                   else Just (i, i+1)) x
```


Figure 4: Definitions of some standard functions using unfoldr

6.4.1 Example 1

We consider first a program that eliminates the character `^M` from a file.

```
main = do xs <- hGetContents h
          ys <- return (filter (/='\^M') xs)
          hPutStr h ys
```

To apply the rule `mmap/foldr/builm` we need to rewrite this function in terms of `mmap`.

```
main = do xs <- mmap (filter (/='\^M')) (hGetContents h)
          hPutStr h xs
```

If we unfold these functions we get:

```
main = do xs <- mmap (foldr (\x xs-> if (x /='\^M') then (x:xs)
                                     else xs) [])
                     (buildm hgetC')
          hPutStr h xs

  where hgetC' c n = do eof <- hIsEOF h
                        if eof then hClose h >> return n
                        else do x <- hGetChar h
                                xs <- hgetC' c n
                                return (c x xs)
```

Now we can apply the `mmap/foldr/buildm` rule and β-reductions, obtaining the following definition, which reads and filter the file simultaneously.

```
main = do xs <- hgetC
          hPutStr h xs
  where hgetC =  do eof <- hIsEOF h
                    if eof then hClose h >> return []
                    else do x <- hGetChar h
                            xs <- hgetC
                            return (if (x /='\^M') then (x:xs) else xs)
```

Observe that the intermediate list between `hGetContents` and `filter` was eliminated. The list between `hgetC` and `hPutStr` can not be eliminated using the rule `mmap/foldr/buildm`, since both functions produce effects.

We run this program with a file of 630 Kbytes. Compiled without adding the rule `mmap/foldr/buildm` to GHC, it took 3.38 seconds and allocated a total of 196 Megabytes of memory. Compiled with the rule, it run in 2.76 seconds and allocated 178 Megabytes of memory. So it had a speed up of 18,34 % and allocated 9.18 % less memory.

6.4.2 Example 2

We consider now a function that compares two lists and prints as result a list of characters '0' and '1', where '1' represents coincidence and '0' the contrary. This function can be defined as:

```
main = putStr (zipWith g xs ys)
 where g x y = if (x==y) then '1' else '0'
```

Unfolding `putStr` and `zipWith`, we obtain:

```
main = destroy putS (unfoldr (\(ms, ns) ->
                   case ms of
                     [] -> Nothing
                     (x:ms')-> case ns of
                                 [] -> Nothing
                                 (y:ns')-> Just (g x y, (ms', ns')))
                             (xs,ys))


 where putS f xs   = case f xs of
                       Nothing -> return ()
                       Just (c,cs) -> putChar c >> putS f cs


       g x y = if (x==y) then '1' else '0'
```

After applying the rule `destroy/unfoldr` we get the more efficient definition:

```
main = putS (\(ms, ns) ->
                 case ms of
                   []-> Nothing
                   (x:ms')-> case ns of
                               []-> Nothing
                               (y:ns')-> Just (g x y, (ms', ns')))
            (xs,ys)

where putS f xs   = case f xs of
                      Nothing -> return ()
                      Just (c,cs) -> putChar c >> putS f cs

      g x y = if (x==y) then '1' else '0'
```

We gave two lists of size 3.8 Megabytes as argument to this function. Compiled without the rule `destroy/unfoldr`, it takes 9.76 seconds and allocates a total of 1120 Megabytes of memory. Compiled with the rule, it takes 8.28 seconds and consumes 944 Megabytes of memory. This means a speed up of 15.16 % and a reduction of 15.71 % in memory allocation.

6.5 Measuring results

To get an idea of what effects the rules produce on the efficiency of programs, we measure run time and memory allocation for a set of programs.

Table 5 gives the results of applying the rule `mmap/foldr/buildm` to 10 programs written using some functions defined in Figures 1 and 2. As we can see, the application of this rule produces significant changes in the efficiency of some programs and does not produce mayor changes in the efficiency of others. None of the programs is getting worse with the application of the rule. The best one improves 45% its run time and 97% its memory allocation, whereas the worst case improves only 1.4% in run time and 4.8% in memory allocation. In average it produces an improvement of 19.6 % in run time and 19.9 % in memory allocation.

Table 6 gives the results of applying the rule `destroy/unfoldr` to 10 programs written using the functions defined in Figures 3 and 4. The improvement is similar in all these programs. Run time decreases 20.9 % in average and memory allocation decrease 25.5 % in average.

Program	Decrease in run time (%)	Decrease in allocation (%)
1	31.5	50.3
2	22.1	20.0
3	11.2	20.0
4	36.0	32.5
5	1.4	4.8
6	13.5	4.8
7	20.6	4.8
8	6.7	20.0
9	7.9	0.8
10	45.0	97.2

Figure 5: Run time and memory allocation improvements for rule mmap/foldr/buildm

6.6 The mmap/foldR/buildRm rule

Let us now instantiate the law 5.1 for rose trees.

```
data Rose a = Empty | Fork a [Rose a]

buildRm :: Monad m => (forall b. (a -> [b] ->b) -> b -> m b) -> m (Rose a)
buildRm g = g Fork Empty

foldR :: (a -> [b] -> b) -> b -> Rose a -> b
foldR k z Empty = z
```

Program	Decrease in run time (%)	Decrease in allocation (%)
1	16.3	21.6
2	24.6	16.7
3	9.7	15.7
4	20.4	17.7
5	28.3	16.9
6	10.8	15.1
7	16.4	20.0
8	25.5	17.7
9	28.5	38.1
10	28.8	20.0

Figure 6: Run time and memory allocation improvements for rule destroy/unfoldr

```
foldR k z (Fork a xs) = k a (map (foldR k z) xs)


{-# RULES "mmap/foldR/buildRm"
    forall k z (g :: Monad m => (forall b. (a -> [b] -> b) -> b -> m b)).
        mmap (foldR k z) (buildRm g) = g k z
#-}
```

We present an example of a program that can be optimized using this rule. The program takes a rose tree and produces as result the sequence of distinct elements found by traversing the tree in depth-first order. To define it we introduce first the following definitions:

```
type MS a b = State [a] b


isDup :: a -> MS a Bool
isDup a = ST (\s -> (any (==a) s, s))
```

```
addS :: a -> MS a ()
addS a = ST (\s -> ((), (a:s)))

elimDups :: Rose a -> MS a (Rose a)
elimDups t = buildRm g
 where g = (\f e -> case t of
                    Empty -> return e
                    Fork a xs -> do b <- isDup a
                                    if b then return e
                                    else do addS a
                                            ts <- mapM g xs
                                            return (f a ts))


flat :: Rose a -> [a]
flat = foldR (:) []
```

The function `elimDups` traverses a rose tree in depth-first order maintaining in the state the nodes visited along the way. In each iteration, it calls the function `isDup` to check if a node of the tree has been visited before. In case a node is marked as visited, the function discards the subtree that has this node as root. In other case, it calls the function `addS` to compute a new state by inserting the node in the list of visited nodes.

The function `flat` takes a rose tree and produces the sequence of nodes that are found by traversing the tree in depth-first order.

The program we want to compute is the composition of `elimDups` and `flat`:

```
main t = do  xs <- mmap flat (elimDups t)
             return ()
```

After applying the rule mmap/foldR/buildRm and some β-reductions, the intermediate tree is removed, leading to the following version:

```
main t = do (case t of
               Empty -> return []
               Fork a xs -> do b <- isDup a
                               if b then return []
                               else do addS a
                                       ts <- mapM g xs
                                       return (a:ts))
            return ()
```

6.7 The mmap/foldr/augmentm rule

In this section we will present a monadic generalization of the foldr/augment rule.

For the definition of this rule we need first to introduce the monadic version of augment:

```
augmentm :: Monad m => (forall b. (a -> b ->b) -> b -> m b)
                        -> [a] -> m [a]
augmentm g ys = g (:) ys
```

This function generalizes buildm by abstracting the constructor `[]` in a list.

The mmap/foldr/augmentm rule is stated as:

```
{-# RULES "mmap/foldr/augmentm"
    forall k z (g :: Monad m =>(forall b.(a -> b -> b) -> b -> m b)).
        mmap (foldr k z) (augmentm g ys) = g k (foldr k z ys)
#-}
```

6.8 The mmap/foldB/augmentBm rule

In the same way as in the previous section, we can write a monadic generalization of the foldB/augmentB rule:

```
augmentBm :: Monad m => (forall b. (b -> b -> b) -> (a -> b) -> m b)
                          -> Btree a -> m (Btree a)
augmentBm g f = g Join f

{-# RULES "mmap/foldB/augmentBm"
    forall j l (g :: Monad m => (forall b. (b -> b -> b)
                                  -> (a -> b) -> m b)).
        mmap (foldB j l) (augmentBm g f) = g j (foldB j l . f)
#-}
```

7 Conclusion

In this thesis we have presented generalizations of monadic short cut deforestation. To define them we needed some notions of category theory. The acid rain theorem and free theorems were also useful to give these definitions. We show that the fusion laws obtained here cover other fusion laws defined in [Par01] as special cases.

We have made an implementation in the Glasgow Haskell compiler, using rewrites rules, of monadic short cut deforestation for lists and rose trees. We describe some practical applications of these, and conclude with a report of performance improvements on a range of monadic programs.

The following are possible future works:

- The construction of an algorithm that derives `foldr`, `builm`, `destroy` and `unfoldr` from recursive definitions, so that monadic short cut deforestation becomes applicable. A work in this direction can be found in [LS95, ZHT96].
- The construction of an algorithm which applies a syntactic transformation to programs to introduce the function `mmap`.
- An implementation in GHC of an algorithm that choose among different definitions of a function in terms of `foldr`, `destroy`, `buildm` and `unfoldr`, so that both rules `mmap/foldr/buildm` and `destroy/unfoldr` can be applied.

References

[BdOM97] R.S. Bird and de O. Moor. *Algebra of Programming.* Prentice Hall, UK, 1997.

[Fok92] M.M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Twente University, 1992.

[Gil96] A.J. Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, University of Glasgow, 1996.

[GLJ] A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation.

[GUV04] Neil Ghani, Tarmo Uustalu, and Varmo Vene. Build, Augment, Destroy. Universally. In *Proc. of Programming Languages and Systems: Second Asian Symposium, APLAS, 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 327–341. Springer Verlag, 2004.

[Jeu93] J.T. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, Utrecht University, 1993.

[JTH01] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. In *International Conference on Functional Programming*, 2001.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Languages and Computer Architecture*, 1995.

[MJ] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming.

[Mog91] E. Moggi. Notions of computation and monads. In *Information and Computation*, pages 93:55–92, 1991.

[Par01] A. Pardo. Fusion of recursive programs with computational effects. In *Theoretical Computer Science*, pages 260:165–207, 2001.

[Par05] A. Pardo. Combining datatypes and effects. In *Advanced Functional Programming*, Lectures Notes in Computer Science, Vol. 3622, 2005.

[Sve02] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming*. Pittsburgh, October 2002.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Functional programming Languages and computer architecture*, 1995.

[Wad89] P. Wadler. Theorems for free. In *4th International Conference on Functional Programming Languages and Computer Architecture*. London, June 1989.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Theoretical Computer Science*, pages 231–248, 1990.

[Wad95] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Berlin, 1995.

[ZHT96] H. Iwasaki Z. Hu and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *International Conference on Functional Programming*. ACM/SIGPLAN, 1996.