Program Fusion with Paramorphisms

Facundo Domínguez   Alberto Pardo
Instituto de Computación, Universidad de la República
Julio Herrera y Reissig 565, 11300 Montevideo, Uruguay
{*fdomin,pardo*}*@fing.edu.uy*

Abstract

The design of programs as the composition of smaller ones is a wide spread approach to programming. In functional programming, this approach raises the necessity of creating a good amount of intermediate data structures with the only aim of passing data from one function to another. Using program fusion techniques, it is possible to eliminate many of those intermediate data structures by an appropriate combination of the codes of the involved functions. In the standard case, no mention to the eliminated data structure remains in the code obtained from fusion. However, there are situations in which parts of that data structure becomes an internal value manipulated by the fused program. This happens, for example, when primitive recursive functions (so-called paramorphisms) are involved. We show, for example, that the result of fusing a primitive recursive function p with another function f may give as result a function that contains calls to f. Moreover, we show that in some cases the result of fusion may be less efficient than the original composition. We also investigate a general recursive version of paramorphism. This study is strongly motivated by the development of a fusion tool for Haskell programs called HFUSION.

Keywords: program fusion, deforestation, paramorphism, primitive recursion, functional programming

1. INTRODUCTION

In functional programming, function composition is a fundamental tool for combining smaller programs to build new ones. Between two composed functions there is always an intermediate data structure which carries data from one function to the other. The overhead of handling such intermediate data structures can be avoided in many cases by replacing the composition by an equivalent function which does not construct the data structure.

Intermediate data structures can be eliminated by a program transformation technique known as *deforestation* [16, 9]. In this paper, we follow an approach to deforestation based on recursion program schemes associated with recursive types [14, 13]. Program schemes have associated algebraic laws, which are useful for reasoning about programs as well as for program transformation purposes. In connection with deforestation, there is a particularly relevant set of algebraic laws, so-called *fusion laws*, which involve the elimination of intermediate data structures.

In the standard case, no mention of the eliminated data structure remains in the code obtained from fusion. However, there are situations in which parts of that data structure becomes an internal value manipulated by the fused program. This may happen, for example, when *paramorphisms* [11] are involved. Paramorphism is a program scheme that captures functions defined by primitive recursion. These are functions that use the arguments and values of the recursive calls to compute their result. A classical example of paramorphism is the factorial function.

The aim of this paper is to analyze fusion laws for paramorphisms. We show, for example, that the fusion of the composition of a paramorphism p with another function f may give as result a function p' that contains calls to f inside, and therefore includes the generation of values produced by f. Moreover, we show the existence of cases where the fusion with a paramorphism may lead to a function that is less efficient than the original composition. To see a simple example of this situation, consider the composition of `tails` and `map`:

```
  tm f = tails . map f
```

```
tails        :: [a] -> [[a]]
tails []      = []
tails (a:as) = as : tails as

map          :: (a -> b) -> [a] -> [b]
map f []      = []
map f (a:as) = f a : map f as
```

Function `tails` is a paramorphism which uses the successive tails of the list as argument to the recursive calls and as values to construct the output list. The fusion of these two functions gives as result the following recursive definition of `tm`:

```
tm f []      = []
tm f (a:as) = map f as : tm f as
```

which retains a call to `map`, now applied to the successive tails of the list separately. This definition of `tm` has $O(n^2)$ time complexity, while the original one was the composition of two $O(n)$ functions. The origin of the problem resides in `tails` and it is caused by the successive sharing of the tails between the recursive calls and the cons operation that builds the output list.

In addition to studying fusion laws for paramorphism, we introduce a new program scheme, called *generalized paramorphism*, which is a general recursion version of paramorphism. The new scheme generalizes paramorphism in the same way hylomorphism (another general recursion scheme) generalizes fold: it is obtained by replacing in the definition of paramorphism the coalgebra of the destructors (of the input datatype) by an arbitrary coalgebra. The expressive power of generalized paramorphism is similar to that of hylomorphism, but in addition it incorporates fusion cases that are not achievable with the fusion laws of hylomorphism.

Our interest in studying this generalization of paramorphism has been motivated by the development of a fusion tool for Haskell programs, called HFUSION.[1] This tool was originally based on hylomorphisms, but now it uses generalized paramorphisms as the internal representation of recursive functions. We are particularly interested in presenting so-called *acid rain laws* [14] for the different program schemes, as they correspond to a mechanizable subset of fusion laws and are the kind of laws being used in the implementation of HFUSION. It is worth mentioning that all the examples to be shown in the paper have been tested in the tool.

The rest of the paper is organized as follows. Section 2 presents background material about recursion program schemes and sets up the notation to be used during the paper. Section 3 reviews the definition of paramorphism and its standard laws, and introduces acid rain laws for this program scheme. Section 4 presents generalized paramorphism and its laws. In these two sections we will show both positive and negative examples of fusion with (generalized) paramorphisms. Section 5 explains our practical motivations for introducing generalized paramorphism. We also describe the fusion cases we gain by the introduction of this program scheme. Section 6 presents final remarks and conclusions.

2. RECURSIVE TYPES AND PROGRAM SCHEMES

Recursive program schemes encapsulate common patterns of computation of recursive functions and have a strong connection with datatypes. A generic definition of them can be formulated using on the categorical approach to recursive types, in which types constitute objects of a category $\mathcal{C}$ and programs are modelled by arrows of the category. This section summarizes the relevant concepts of this approach to recursive types and the generic definition and standard laws of three well-known schemes: fold, unfold and hylomorphism (see e.g. [12, 4, 5, 14, 6]).

Throughout the paper the working category will be $\mathbf{Cpo}$, the category of pointed cpos (complete partial orders with a least element $\perp$) and continuous functions. The choice of this category

[1]`http://www.fing.edu.uy/inco/proyectos/fusion/tool`

facilitates us to work with arbitrary recursive functions. As usual, a function f is said to be strict if it preserves the least element, i.e. $f \perp = \perp$. The final object of $\mathbf{Cpo}$ is given by the singleton set $\{\perp\}$ and will be written as 1. This object will correspond to our interpretation of the unit type `()`, whose unique element is also written as `()`.[2] Product is defined as cartesian product, with projections $\pi_1 :: a \times b \to a$ and $\pi_2 :: a \times b \to b$. The pairing (or split) of two functions $f :: c \to a$ and $g :: c \to b$ is written $\langle f, g \rangle :: c \to a \times b$. Sum $a + b$ is defined as separated sum, with sum inclusions $inl :: a \to a + b$ and $inr :: b \to a + b$. Given two continuous functions $f :: a \to c$ and $g :: b \to c$, case analysis is defined as the strict function $f \triangledown g :: a + b \to c$ such that $(f \triangledown g) \circ inl = f$ and $(f \triangledown g) \circ inr = g$. Product and sum can be generalized to n components. In the generalization of the sum we will write in_i to denote the i-th injection.

In the categorical modelling of types, a datatype τ is understood as a solution to an equation $x \cong Fx$, for an appropriate endofunctor F that captures the shape (or signature) of the type. Given a locally continuous and strictness preserving functor F on $\mathbf{Cpo}$, a recursive domain equation $x \cong Fx$ has a unique solution specified by a pointed cpo μF together with an isomorphism provided by a pair of strict functions $in_F :: F\mu F \to \mu F$ and $out_F :: \mu F \to F\mu F$ each others inverse. The cpo μF contains partial, finite, as well as infinite values. Function in_F encodes the constructors of the datatype, while out_F the destructors.

2.1. Fold

Given an endofunctor $F : \mathbf{Cpo} \to \mathbf{Cpo}$, a function $\phi :: Fa \to a$ is called a F-*algebra*. In particular, observe that in_F is an algebra. A *homomorphism* between two algebras $\phi :: Fa \to a$ and $\phi' :: Fb \to b$ is a function $f :: a \to b$ such that $f \circ \phi = \phi' \circ Ff$.

The least homomorphism between in_F and any other algebra $\phi :: F\ a \to a$ gives rise to a recursion scheme, denoted by $(\!|\phi|\!)_F :: \mu F \to a$ and usually called *fold* [2] or *catamorphism* [12], which captures definitions by structural recursion. That is, fold is defined as the least function that satisfies the equation $f \circ in_F = \phi \circ Ff$. Therefore,

$$(\!|\phi|\!)_F \circ in_F = \phi \circ F(\!|\phi|\!)_F \tag{1}$$

For lists, fold corresponds to the standard `foldr` operator used in functional programming. A fold $(\!|\phi|\!)_F$ is strict iff its algebra ϕ is strict. Fold satisfies the following laws:

Fold identity

$$(\!|in_F|\!)_F = id_{\mu F} \tag{2}$$

Fold fusion

$$f \text{ strict } \wedge\ f \circ \phi = \phi' \circ Ff \ \Rightarrow\ f \circ (\!|\phi|\!)_F = (\!|\phi'|\!)_F \tag{3}$$

Fold-fold fusion

$$\phi \text{ strict } \wedge\ \tau :: \forall a.\, (F\ a \to a) \to (G\ a \to a) \quad \Rightarrow \quad (\!|\phi|\!)_F \circ (\!|\tau(in_F)|\!)_G = (\!|\tau(\phi)|\!)_G \tag{4}$$

The last law is usually referred to as *acid rain* [14]. The goal of acid rain is to combine a function that produces a data structure with another that consumes it. A polymorphic function $\tau :: \forall a.\, (F\ a \to a) \to (G\ a \to a)$ that converts F-algebras into G-algebras is said to be a *transformer* [5]. Every function τ of this type satisfies the following property, which can be inferred as a free theorem [15]: for every $f :: a \to b$, $\phi :: F\ a \to a$ and $\phi' :: F\ b \to b$, if $f \circ \phi = \phi' \circ Ff$ then $f \circ \tau(\phi) = \tau(\phi') \circ Gf$. That is, every homomorphism between two F-algebras is also a homomorphism between the respective G-algebras.

[2]Our semantics differs slightly from that of Haskell in that we do not consider lifted domains as the interpretation of types.

2.2. Unfold

Given a functor F, a function $\psi :: a \to F a$ is called a F*-coalgebra*. In particular, $out_F :: \mu F \to F \mu F$ is a coalgebra. A *homomorphism* between two coalgebras $\psi :: a \to F a$ and $\psi' :: b \to F b$ is a function $f :: a \to b$ such that $\psi' \circ f = F f \circ \psi$. F-coalgebras and their homomorphisms form a category. The coalgebra $out_F :: \mu F \to F \mu F$ turns out to be *final* in this categoty. This means that there exists a unique homomorphism from any coalgebra $\psi :: a \to F a$ to out_F, which is denoted by $[\!(\psi)\!]_F :: a \to \mu F$. It gives rise to a recursion scheme, called *unfold* [8] or *anamorphism* [12], which satisfies the equation:

$$out_F \circ [\!(\psi)\!]_F = F\,[\!(\psi)\!]_F \circ \psi \tag{5}$$

Unfold captures definitions by structural corecursion. It satisfies the following laws:

Unfold identity

$$[\!(out_F)\!]_F = id_{\mu F} \tag{6}$$

Unfold fusion

$$\psi \circ f = F f \circ \psi' \;\Rightarrow\; [\!(\psi)\!]_F \circ f = [\!(\psi')\!]_F \tag{7}$$

Unfold-unfold fusion

$$\sigma :: \forall a.\,(a \to F\,a) \to (a \to G\,a) \quad\Rightarrow\quad [\!(\sigma(out_F))\!]_G \circ [\!(\psi)\!]_F = [\!(\sigma(\psi))\!]_G \tag{8}$$

Unfold-unfold fusion is another case of *acid rain* [14]. A polymorphic function $\sigma :: \forall a.\,(a \to F\,a) \to (a \to G\,a)$ is now a transformer from F-coalgebras to G-coalgebras. In this case the free theorem states that, for every $f :: a \to b$, and coalgebras $\psi :: a \to F\,a$ and $\psi' :: b \to F\,b$, if $\psi' \circ f = F f \circ \psi$ then $\sigma(\psi') \circ f = G f \circ \sigma(\psi)$.

2.3. Hylomorphism

Fold and unfold express standard ways of consuming and generating data structures, respectively. Now we look at functions given by the composition of a fold with an unfold. They correspond to general recursive functions whose structure is dictated by a virtual intermediate data structure.

Given an algebra $\phi :: F\,b \to b$ and a coalgebra $\psi :: a \to F a$, the *hylomorphism* [12, 4, 5] $[\![\phi, \psi]\!]_F :: a \to b$ is the function defined as:

$$[\![\phi, \psi]\!]_F = (\!|\phi|\!)_F \circ [\!(\psi)\!]_F \tag{9}$$

Alternatively, hylomorphism can be defined as the least function that satisfies the equation $f = \phi \circ F\,f \circ \psi$. This shows that we can always transform the composition of a fold with an unfold into a single function that avoids the construction of the intermediate data structure. From this definition, we obtain the equation

$$[\![\phi, \psi]\!]_F = \phi \circ F\,[\![\phi, \psi]\!]_F \circ \psi \tag{10}$$

which expresses the shape of recursion that comes with each datatype. Two well-known examples of hylomorphisms are quicksort and merge sort (see e.g [1, 6]). The expressiveness of hylomorphisms is very rich. In practice, almost every interesting recursive function can be expressed as a hylomorphism.

Applying the identity laws corresponding to fold and unfold, it is immediate to see that fold and unfold are themselves a hylomorphism:

$$(\!|\phi|\!)_F = [\![\phi, out_F]\!]_F \qquad\qquad [\!(\psi)\!]_F = [\![in_F, \psi]\!]_F$$

The following fusion laws are a direct consequence of (9) and the fusion laws for fold and unfold.

Hylo fusion

$$f \text{ strict } \wedge\ f \circ \phi = \phi' \circ F f \quad \Rightarrow \quad f \circ \llbracket \phi, \psi \rrbracket_F = \llbracket \phi', \psi \rrbracket_F \tag{11}$$

$$\psi \circ f = F f \circ \psi' \quad \Rightarrow \quad \llbracket \phi, \psi \rrbracket_F \circ f = \llbracket \phi, \psi' \rrbracket_F \tag{12}$$

Fold-hylo fusion

$$\phi \text{ strict } \wedge\ \tau :: \forall a.\, (F\ a \to a) \to (G\ a \to a) \quad \Rightarrow \quad (\!|\phi|\!)_F \circ \llbracket \tau(in_F), \psi \rrbracket_G = \llbracket \tau(\phi), \psi \rrbracket_G \tag{13}$$

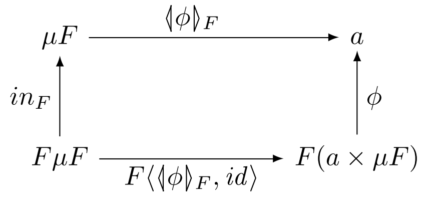
Hylo-unfold fusion

$$\sigma :: \forall a.\, (a \to F\ a) \to (a \to G\ a) \quad \Rightarrow \quad \llbracket \phi, \sigma(out_F) \rrbracket_G \circ [\!(\psi)\!]_F = \llbracket \phi, \sigma(\psi) \rrbracket_G \tag{14}$$

3. PARAMORPHISM

Paramorphisms [11] correspond to primitive recursive functions. Therefore, like folds, they capture functions that are defined by structural recursion. In this section we review the definition of paramorphism (presenting it in the context of $\mathbf{Cpo}$) and some of its standard laws. We also introduce new acid rain laws that relate paramorphisms with folds.

Given a function $\phi :: F(a \times \mu F) \to a$, the *paramorphism* $\langle\!|\phi|\!\rangle_F :: \mu F \to a$ is the least function that satisfies the equation $f \circ in_F = \phi \circ F\langle f, id\rangle$. The following diagram makes the types explicit:



$$\begin{tikzcd}
\mu F \arrow[r, "\langle\!|\phi|\!\rangle_F"] & a \\
F\mu F \arrow[u, "in_F"] \arrow[r, "F\langle \langle\!|\phi|\!\rangle_F, id\rangle"'] & F(a \times \mu F) \arrow[u, "\phi"']
\end{tikzcd}$$

The difference between paramorphisms and folds is in the amount of information available in each recursive step. In addition to the values returned by the recursive calls (as in fold), function ϕ has also available their arguments. As we will see later on in this section, this subtle difference with folds makes paramorphisms inappropriate for fusion in some cases.

Example 3.1 Consider the list datatype. Its base functor is given by $L_a b = 1 + a \times b$. For $\phi_1 :: 1 \to b$ and $\phi_2 :: a \times (b \times [a]) \to b$, the paramorphism $\langle\!|\phi_1 \triangledown \phi_2|\!\rangle_{L_a} :: [a] \to b$ is the least function such that

$$f\ [\,] = \phi_1() \qquad\qquad f\ (a : as) = \phi_2(a, (f\ as, as))$$

□

The following equations express the well-known relationship between paramorphisms and folds.

$$\langle\!|\phi|\!\rangle_F = \pi_1 \circ (\!|\langle \phi, in_F \circ F\pi_2\rangle|\!)_F \tag{15}$$

$$(\!|\phi|\!)_F = \langle\!|\phi \circ F\ \pi_1|\!\rangle_F \tag{16}$$

Equation (15) is usually taken as the definition of paramorphism. It states that a paramorphism can be implemented as a fold that produces a pair, whose second component contains a (recursively generated) copy of the input. Equation (16) shows that a fold is a paramorphism that ignores the copy of the arguments to the recursive calls.

The following is the fusion law for paramorphism [11].

$$f \text{ strict } \wedge\ f \circ \phi = \phi' \circ F(f \times id) \quad \Rightarrow \quad f \circ \langle\!|\phi|\!\rangle_F = \langle\!|\phi'|\!\rangle_F \tag{17}$$

The rest of this section is devoted to the analysis of acid rain laws for paramorphisms. We are not aware that they have been presented before. These laws will serve us as basis for designing the acid rain laws for generalized paramorphisms in section 4.

The first law we consider refers to the composition of a fold with a paramorphism.

Proposition 3.2 (fold-para fusion) *For strict ϕ,*

$$\tau :: \forall a.\, (F\, a \to a) \to (G\, (a \times \mu G) \to a) \quad \Rightarrow \quad (\!|\phi|\!)_F \circ \{\!|\tau(in_F)|\!\}_G = \{\!|\tau(\phi)|\!\}_G$$

Proof Since $(\!|\phi|\!)_F$ is a homomorphism between the algebras in_F and ϕ, by the free theorem associated with the polymorphic type of τ it follows that

$$(\!|\phi|\!)_F \circ \tau(in_F) = \tau(\phi) \circ G\, ((\!|\phi|\!)_F \times id)$$

Therefore, by applying (17) we obtain the desired result. The strictness condition required to $(\!|\phi|\!)_F$ in (17) follows from the assumption that ϕ is strict. □

The next fusion law refers to the composition between a paramorphism and a fold. It is particularly interesting and important as it exhibits a case in which the paramorphism internalizes the generation of values of the intemediate data structure that wants to be eliminated. The following lemma will be used in the proof of the law.

Lemma 3.3 *For F-algebras $\phi :: Fa \to a$ and $\psi :: F(b \times a) \to (b \times a)$, and strict $f :: a \to b$,*

$$\langle f, id\rangle \circ \phi = \psi \circ F\langle f, id\rangle \quad \Rightarrow \quad f \circ (\!|\phi|\!)_F = \{\!|\pi_1 \circ \psi \circ F(id \times (\!|\phi|\!)_F)|\!\}_F$$

Proof Let us call p the paramorphism and c the fold. By definition of paramorphism and fold we have that

$$p \circ in_F = \pi_1 \circ \psi \circ F\langle p, c\rangle \qquad \text{and} \qquad c \circ in_F = \phi \circ F\, c$$

The two functions are defined simultaneously in an asymmetric way. That is, p depends on c while c does not depend on p. Definitions following this pattern are called a *zygomorphism* [10]. From the definition of p and c, it can be derived that [5]:

$$\langle p, c\rangle = (\!|\langle \pi_1 \circ \psi, \phi \circ F\pi_2\rangle|\!)_F$$

In the context of $\mathbf{Cpo}$ this equation is proved by fixed point induction. If we call pc the split $\langle p, c\rangle$, then the statement of the lemma can be rewritten as:

$$\langle f, id\rangle \circ \phi = \psi \circ F\langle f, id\rangle \quad \Rightarrow \quad f \circ \pi_2 \circ pc = \pi_1 \circ pc$$

which can then be proved by fixed point induction. □

Proposition 3.4 (para-fold fusion) *For strict ϕ,*

$$\begin{array}{ll} & \tau :: \forall a.\, (F\, a \to a) \to (G\, a \to a) \\ \Rightarrow & \\ & \{\!|\phi|\!\}_F \circ (\!|\tau(in_F)|\!)_G = \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times (\!|\tau(in_F)|\!)_G)|\!\}_G \end{array}$$

Proof From the definition of paramorphism we can derive that $\langle \{\!|\phi|\!\}_F, id\rangle$ is a homomorphism between the F-algebras in_F and $\langle \phi, in_F \circ F\pi_2\rangle$:

$$\langle \{\!|\phi|\!\}_F, id\rangle \circ in_F = \langle \phi, in_F \circ F\pi_2\rangle \circ F\langle \{\!|\phi|\!\}_F, id\rangle$$

Then, by the free theorem associated with the polymorphic type of τ it follows that $\langle \{\!|\phi|\!\}_F, id\rangle$ is also a homomorphism between the G-algebras $\tau(in_F)$ and $\tau(\langle \phi, in_F \circ F\pi_2\rangle)$:

$$\langle (\!|\phi|\!)_F, id\rangle \circ \tau(in_F) = \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G\langle (\!|\phi|\!)_F, id\rangle$$

Finally, by Lemma 3.3 the desired result follows. Strictness of $(\!|\phi|\!)_F$, necessary for the application of Lemma 3.3, is a consequence of the assumption that ϕ is strict. □

Example 3.5 This example shows a simple case in which the fold is copied into the body of the resulting paramorphism, producing multiple generations of data structures.

```
tf p = tails . filter p

filter          :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (a:as) = if p a then a : filter p as else filter p as
```

Function `tails` is a paramorphism while `filter` is a fold:

```
tails = ⟨$\phi_1 \nabla \phi_2$⟩
  where $\phi_1$ = $\lambda$(). []
        $\phi_2$ = $\lambda$(a,(ys,as)). as : ys

filter p = (|$\phi'_1 \nabla \phi'_2$|)
    where $\phi'_1$ = $\lambda$(). []
          $\phi'_2$ = $\lambda$(a,ys). if p a then a : ys else ys
```

The algebra of `filter` can be expressed as $\tau(in)$, where τ is a polymorphic function given by:

```
$\tau$                 :: $(1 + a \times b \to b) \to (1 + a \times b \to b)$
$\tau(\alpha)$              = $\tau_1(\alpha) \nabla \tau_2(\alpha)$
$\tau_1(\alpha_1 \nabla \alpha_2)$  = $\alpha_1$
$\tau_2(\alpha_1 \nabla \alpha_2)$  = $\lambda$(a,b). if p a then $\alpha_2$(a,b) else b
```

Therefore, if we apply para-fold fusion we obtain the following:

$$\texttt{tf p} = \langle\!| \pi_1 \circ \tau(\langle \phi, in \circ (id + id \times \pi_2)\rangle) \circ (id + id \times (id \times \texttt{filter p}) |\!\rangle$$

Inlining,

```
tf p [] = []
tf p (a:as) = if p a then filter p as : tf p as else tf p as
```

We applied fusion with the aim to eliminate the intermediate list that was generated by `filter`, but as result we obtained a function that filters the successive tails of the input list separately. This means that fusion transformed the composition of two functions with linear time behaviour to a function which is quadratic! In other words, in this case the effect of the medicine was worse than the illness itself. □

Example 3.6 This example shows another case of the situation presented in the previous example (we simply show the result of applying fusion and skip the details). Consider the function that counts the number of words of a text after having filtered it with a predicate `p`.

```
wcf p = wc . filter p

wc        :: String -> Int
wc []      = 0
wc (c:cs) = case cs of
              []   -> if isSpace c then 0 else 1
              d:ys -> if (not (isSpace c)) && (isSpace d)
                         then 1 + wc cs
                         else wc cs
```

Function `wc` is a paramorphism. It is inspired in one of the word counting algorithms described in [7]. This function adds one each time the end of a word is detected, and for this it uses the current character `c` and the next one `d` (except at the end). By para-fold fusion we obtain as result a paramorphism with the following recursive definition:

```
wcf p []     = 0
wcf p (c:cs) = if p c
                 then case filter p cs of
                        []   -> if isSpace c then 0 else 1
                        d:ys -> if (not (isSpace c)) && (isSpace d)
                                   then 1 + wcf p cs
                                   else wcf p cs
                 else wcf p cs
```

In the original definition of `wcf`, the inspection of the tail was performed on a text that was already filtered. Now, on the contrary, an on-line filtering of the tail is necessary each time before inspection. In this case the time behaviour of the resulting program is linear as the original ones. However, it may happen that the predicate `p` is applied twice to some of the elements of the input string: once in the context of `filter` and another one in the condition of the `if-then-else`. Also, note that the list nodes originally produced by `filter` are still produced when evaluating the `case` on `filter p cs`. So, in spite of our efforts, we could not eliminate the intermediate list. □

There exist of course applications of para-fold fusion that yield satisfactory results. This is illustrated by the following example.

Example 3.7 Consider the function that replaces the first occurrence of a value in a list by a given value.

```
replace           :: Eq a => a -> a -> [a] -> [a]
replace x y []     = []
replace x y (a:as) = if (a==x) then y : as else a : replace x y as
```

This function is a paramorphism because it returns the tail of the input list as part of the result when the sought value is met.

```
 replace x y = ⟨φ₁∇φ₂⟩
   where φ₁ = λ(). []
         φ₂ = λ(a,zs,as). if a==x then y : as else a : zs
```

Suppose that we want to replace an element in a list after filtering.

```
repf x y p = replace x y . filter p
```

We are again in a situation where we can apply para-fold fusion, obtaining a paramorphism with the following recursive definition:

```
repf x y p []     = []
repf x y p (a:as) = if p a then if a==x then y : filter p as
                                        else a : repf x y p as
                      else repf x y p as
```

In this case `filter` needs to be applied to the sublist that remains after the replaced element (in case that element was found), as that sublist is returned as part of the result. □

Example 3.8 Consider the composition of the function that inserts a value in a binary search tree with the map function for binary trees.

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

```
insmap x f = insert x . mapT f

insert x Empty          = Node x Empty Empty
insert x (Node a t1 t2) = if x < a then Node a (insert x t1) t2
                                   else Node a t1 (insert x t2)

mapT f Empty          = Empty
mapT f (Node a t1 t2) = Node (f a) (mapT f t1) (mapT f t2)
```

The application of para-fold fusion yields a satisfactory result in this case:

```
insmap x f Empty          = Node x Empty Empty
insmap x f (Node a t1 t2) = if x < f a then Node (f a) (insmap x f t1) (mapT f t2)
                                       else Node (f a) (mapT f t1) (insmap x f t2)
```

□

Note 3.9 The previous examples have shown the existence of some cases where para-fold fusion may worsen performance. These are fusions of the form $\langle\!| \phi |\!\rangle_F \circ f$ in which occurrences of f in the result produce the generation of duplicated computations. This means that, in the presence of paramorphisms, fusion cannot be applied without restrictions. It is necessary thus to include some code analysis that helps us to avoid the application of fusion in those cases we know performance will decrease. At the moment HFUSION does not perform this kind of analysis, but we plan to do so in the near future.

We give an intuitive characterization of the different cases of $\langle\!| \phi |\!\rangle_F \circ f$ in terms of the notion of "computation". The analysis focuses on function ϕ of the paramorphism:

- If during the computation of ϕ both the values returned by the recursive calls and their arguments are necessary, then fusion should be avoided. This is the case of `tails . filter p` and `wc . filter p`.
- If the values returned by the recursive calls or their arguments (but not both) appear during the computation of ϕ, then fusion can be safely performed. This is the case of `replace x y . filter p` and `insert x . mapT f`.
  For instance, in the case of `insert x` $= \langle\!| \phi |\!\rangle$, ϕ is given by:

  ϕ_1 `()                    = Node x Empty Empty`
  ϕ_2 `(a,(t1,r1),(t2,r2)) = if x < a then Node a r1 t2 else Node a t1 r2`

  If a computation uses `t1`, then it does not use `r1`, and vice-versa. The same holds for `t2` and `r2`. This is the reason that makes fusion in Example 3.8 adequate. □

4. GENERALIZED PARAMORPHISMS

This section presents a new program scheme that generalizes paramorphisms in the same sense hylomorphisms generalize folds. This generalization of paramorphism will permit us to capture a wider class of recursive functions that use the arguments of the recursive calls to compute the final result. We will state fusion laws associated with generalized paramorphisms, but now in combination with folds, unfolds and hylomorphisms.

To see how this generalization is obtained, let us recall the diagram that a paramorphism satisfies, writing out_F instead of in_F:

$$\begin{tikzcd}
\mu F \arrow[r, "f"] \arrow[d, "out_F"'] & a \\
F\mu F \arrow[r, "F\langle f, id\rangle"'] & F(a \times \mu F) \arrow[u, "\phi"']
\end{tikzcd}$$

The arguments to the recursive calls are obtained by applying the coalgebra corresponding to the destructors of the data type. The generalization we introduce is obtained by considering an arbitrary coalgebra instead.

Given $\phi :: F(b \times a) \to b$ and a coalgebra $\psi :: a \to F\ a$, the *generalized paramorphism* $\{\!|\phi, \psi|\!\}_F :: a \to b$ is the least function that makes the following diagram commute:

$$\begin{tikzcd}
a \arrow[r, "f"] \arrow[d, "\psi"'] & b \\
F\ a \arrow[r, "F\langle f, id\rangle"'] & F(b \times a) \arrow[u, "\phi"']
\end{tikzcd}$$

The notion of generalized paramorphism is in some sense related with that of parametrically recursive coalgebra [3].

Example 4.1 Consider the functor L_a that captures the signature of lists. For $\phi_1 :: () \to b$ and $\phi_2 :: a \times (c \times b) \to c$, the paramorphism $f = \{\!|\phi_1 \triangledown \phi_2, \psi|\!\}_{L_a} :: b \to c$ is the least function such that

$$\begin{array}{l} f\ b = \texttt{case}\ \psi\ b\ \texttt{of} \\ \qquad inl() \qquad \to \phi_1 \\ \qquad inr(a, b') \to \phi_2(a, (f\ b', b')) \end{array}$$

□

The following equation expresses the fact that paramorphisms are a particular instance of generalized paramorphisms:

$$\langle\!|\phi|\!\rangle_F = \{\!|\phi, out_F|\!\}_F \tag{18}$$

Generalized paramorphisms are so expressive as hylomorphisms. The following equation shows that every hylomorphism can be written as a generalized paramorphism. It states a relationship similar to the one between folds and paramorphisms (equation 16).

$$[\![\phi, \psi]\!]_F = \{\!|\phi \circ F\ \pi_1, \psi|\!\}_F \tag{19}$$

The relationship in the other direction is the following. For each $\psi :: a \to F\ a$, let us define the functor $G\,x = F(x \times a)$. Then,

$$\{\!|\phi, \psi|\!\}_F = [\![\phi, F\ \Delta \circ \psi]\!]_G \tag{20}$$

where $\Delta = \langle id, id\rangle$. Thus, for $g = \{\!|\phi, \psi|\!\}_F$,

$$\begin{tikzcd}
a \arrow[r, "g"] \arrow[d, "F\Delta \circ \psi"'] & b \\
F\ (a \times a) \arrow[r, "F(g \times id)"'] & F(b \times a) \arrow[u, "\phi"']
\end{tikzcd}$$

The following two fusion laws resemble laws for hylomorphism. Observe that in (22) the colagebra homomorphism is internalized as part of the code of the resulting generalized paramorphism.

Proposition 4.2 (gpara fusion)

$$f\ \textit{strict} \ \wedge\ f \circ \phi = \phi' \circ F(f \times id) \quad \Rightarrow \quad f \circ \{\!|\phi, \psi|\!\}_F = \{\!|\phi', \psi|\!\}_F \tag{21}$$

$$\psi \circ f = F\ f \circ \psi' \quad \Rightarrow \quad \{\!|\phi, \psi|\!\}_F \circ f = \{\!|\phi \circ F(id \times f), \psi'|\!\}_F \tag{22}$$

Proof Both laws can be proved by fixed point induction. We show the proof of (22) as it illustrates how f becomes part of the result. Let us define $\gamma(g) = \phi \circ F\langle g, id\rangle \circ \psi$ and $\gamma'(g) =$

$\phi \circ F(id \times f) \circ F\langle g, id\rangle \circ \psi'$. The proof proceeds with predicate $g \circ f = g'$. The base case $\bot \circ f = \bot$ is immediate. Now, assume that $g \circ f = g'$. Then, $\gamma(g) \circ f = \phi \circ F\langle g, id\rangle \circ \psi \circ f = \phi \circ F\langle g, id\rangle \circ F f \circ \psi' = \phi \circ F\langle g \circ f, f\rangle \circ \psi' = \phi \circ F(id \times f) \circ \langle g', id\rangle \circ \psi' = \gamma'(g')$. Therefore, by fixed point induction it follows that $\mathsf{fix}(\gamma) \circ f = \mathsf{fix}(\gamma')$. □

Taking into account the close similarity between generalized paramorphisms and hylomorphisms, one may think of the existence of a factorization property similar to that of hylomorphism, which states that every generalized paramorphism can be split up into the composition of a paramorphism with an unfold, i.e. $\{\!|\phi, \psi|\!\}_F = \langle\!|\phi|\!\rangle_F \circ [\!(\psi)\!]_F$. However, this law does not hold. The reason for the failure is originated in the fact that paramorphisms, in contrast to folds, use the arguments to the recursive calls to compute their results. The following law shows that the result of fusing the composition of a paramorphism with an unfold is a generalized paramorphism which internalizes the computation of the unfold as part of its code.

Proposition 4.3 (para-unfold fusion)

$$\langle\!|\phi|\!\rangle_F \circ [\!(\psi)\!]_F = \{\!|\phi \circ F(id \times [\!(\psi)\!]_F), \psi|\!\}_F$$

Proof $\langle\!|\phi|\!\rangle_F \circ [\!(\psi)\!]_F \overset{(18)}{=} \{\!|\phi, out_F|\!\}_F \circ [\!(\psi)\!]_F \overset{(22)}{=} \{\!|\phi \circ F(id \times [\!(\psi)\!]_F), \psi|\!\}_F$ □

Example 4.4 Consider the following composition:

```
tdown = tails . down

down  :: Int -> [Int]
down 0 = []
down n = n : down (n-1)
```

Function `tails` is a paramorphism while `down` is an unfold. By applying para-unfold fusion we obtain:

```
tdown 0 = []
tdown n = down (n-1) : tdown (n-1)
```

This is again a situation in which the composition of two linear time functions gives a quadratic function as result. This is due to `tails`. □

The following law is a direct consequence of para-fold fusion (Proposition 3.4).

Proposition 4.5 (para-hylo fusion) *For strict ϕ,*

$$\begin{array}{l} \quad \tau :: \forall a.\, (F\ a \to a) \to (G\ a \to a) \\ \Rightarrow \\ \quad \langle\!|\phi|\!\rangle_F \circ [\![\tau(in_F), \psi]\!]_G = \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times [\![\tau(in_F), \psi]\!]_G), \psi|\!\}_G \end{array}$$

Proof

$$\begin{array}{ll} & \langle\!|\phi|\!\rangle_F \circ [\![\tau(in_F), \psi]\!]_G \\ = & \qquad \{ \text{ hylo factorization (9) } \} \\ & \langle\!|\phi|\!\rangle_F \circ (\!|\tau(in_F)|\!)_G \circ [\!(\psi)\!]_G \\ = & \qquad \{ \text{ para-fold fusion (Prop. 3.4) } \} \\ & \langle\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times (\!|\tau(in_F)|\!)_G)|\!\rangle_G \circ [\!(\psi)\!]_G \\ = & \qquad \{ \text{ para-unfold fusion (Prop. 4.3) } \} \\ & \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times (\!|\tau(in_F)|\!)_G) \circ G(id \times [\!(\psi)\!]_G), \psi|\!\}_G \\ = & \qquad \{ \text{ functor } G \text{ and hylo factorization (9) } \} \\ & \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times [\![\tau(in_F), \psi]\!]_G), \psi|\!\}_G \end{array}$$

□

The two previous fusion laws showed compositions that yield generalized paramorphisms as result. The laws that follow are acid rain laws with generalized paramorphism as argument.

Proposition 4.6 (fold-gpara fusion) *Let $\psi : b \to G\, b$. For strict ϕ,*

$$\tau :: \forall a.\, (F\ a \to a) \to (G\ (a \times b) \to a)\ \Rightarrow\ (\!|\phi|\!)_F \circ \{\!|\tau(in_F), \psi|\!\}_G = \{\!|\tau(\phi), \psi|\!\}_G$$

Proof Similar proof to fold-para fusion (Prop. 3.2), but using (21) instead. □

The generalization of paramorphism opens the possibility of an acid rain law with unfold.

Proposition 4.7 (gpara-unfold fusion)

$$\sigma :: (a \to F\ a) \to (a \to G\ a)\ \Rightarrow\ \{\!|\phi, \sigma(out_F)|\!\}_G \circ [\!(\psi)\!]_F = \{\!|\phi \circ G(id \times [\!(\psi)\!]_F), \sigma(\psi)|\!\}_G$$

Proof Same proof to fold-para fusion (Prop. 3.2), but using (22) and the free theorem for σ. □

Example 4.8 Consider the following composition:

```
dm p f = drop2While p . map f

drop2While            :: (a -> Bool) -> [a] -> [a]
drop2While p []        = []
drop2While p [a]       = if p a then [] else [a]
drop2While p (a:a':as) = if p a then drop2While p as else a:a':as
```

Function `drop2While` can be defined as a generalized paramorphism with functor $H_a\ b = 1 + a + a \times a \times b$.

```
drop2While p = {|$\phi_1 \triangledown \phi_2 \triangledown \phi_3, \psi$|}$_{H_a}$
    where $\phi_1$ = $\lambda$(). []
          $\phi_2$ = $\lambda$a.if p a then [] else [a]
          $\phi_3$ = $\lambda$(a,a',(ys,as)).if p a then ys else a:a':as
           $\psi$ = $\lambda$as. case as of
                    []        -> $in_1$ ()
                    [a]       -> $in_2$ a
                    (a:a':as)-> $in_3$ (a,a',as)
```

The coalgebra ψ does not correspond to out_{L_a}, for L_a the base functor of lists, because it contains nested patterns. It can, however, be written as $\psi = \sigma(out_{L_a})$, where σ is given by:

```
$\sigma$      :: $(b \to L_a\ b) \to (b \to H_a\ b)$
$\sigma(\beta)$ = $\lambda$b. case $\beta$ b of
            in$_1$ ()     -> in$_1$ ()
            in$_2$ (a,b')->  case $\beta$ b' of
                            in$_1$ ()       -> in$_2$ a
                            in$_2$ (a',b'')-> in$_3$ (a,a',b'')
```

On the other hand, `map`, which is usually presentd as a fold over lists, can be expressed as an unfold as well: `map f` = $[\!(\psi)\!]$, where

```
$\psi$ = $\lambda$xs. case xs of
         []     -> $in_1$ ()
         (x:xs) -> $in_2$ (f x,xs)
```

Therefore, we can apply gpara-unfold fusion, obtaining

$$\texttt{dm p f} = \{\!|\phi_1 \nabla \phi_2 \nabla (\phi_3 \circ (id \times id \times (id \times \texttt{map f})), \sigma(\psi)|\!\}$$

Inlining,

```
dm p f []     = []
dm p f (a:as) = let fa = f a
                in case as of
                     []      -> if p fa then [] else [fa]
                     (a',xs) -> if p fa then dm p f xs else fa:f a':map f xs
```

Fusion in this case is completely satisfactory. □

And now we show a law that relates paramorphisms with generalized paramorphisms.

Proposition 4.9 (para-gpara fusion) *Let $\psi : b \to G\, b$. For strict ϕ,*

$$\tau :: \forall a.\, (F\; a \to a) \to (G\; (a \times b) \to a)$$
$$\Rightarrow$$
$$\langle\!|\phi|\!\rangle_F \circ \{\!|\tau(in_F), \psi|\!\}_G = \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G\langle id \times \{\!|\tau(in_F), \psi|\!\}_G, \pi_2\rangle, \psi|\!\}_G$$

Proof

$$
\begin{aligned}
& \langle\!|\phi|\!\rangle_F \circ \{\!|\tau(in_F), \psi|\!\}_G \\
= & \quad \{ \text{ (20), } H\; x = G(x \times b) \; \} \\
& \langle\!|\phi|\!\rangle_F \circ [\![\tau(in_F), G\Delta \circ \psi]\!]_H \\
= & \quad \{ \text{ Prop. 4.5 and def. of } H \; \} \\
& \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G((id \times \{\!|\tau(in_F), \psi|\!\}_G) \times id), G\Delta \circ \psi|\!\}_H \\
= & \quad \{ \text{ product manipulation } \} \\
& \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G\langle id \times \{\!|\tau(in_F), \psi|\!\}_G, \pi_2\rangle, \psi|\!\}_G
\end{aligned}
$$

□

Corollary 4.10 (para-para fusion) *For strict ϕ,*

$$\tau :: \forall a.\, (F\; a \to a) \to (G\; (a \times \mu G) \to a)$$
$$\Rightarrow$$
$$\langle\!|\phi|\!\rangle_F \circ \langle\!|\tau(in_F)|\!\rangle_G = \langle\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G\langle id \times \langle\!|\tau(in_F)|\!\rangle_G, \pi_2\rangle|\!\rangle_G$$

Example 4.11 Using para-para fusion we can transform

```
dWt p = dropWhile p . tails

dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile p []      = []
dropWhile p (a:as) = if p a then dropWhile p as else a : as
```

into

```
dWt p []     = []
dWt p (a:as) = if p as then dWt p as else as : tails as
```

□

Note 4.12 The characterization of good and bad cases of fusion that we can add with the introduction of generalized paramorphism is very the same as the one presented in Note 3.9. Now we must analyze function ϕ in compositions of the form $\{\!|\phi, \psi|\!\}_F \circ f$ and $\langle\!|\phi|\!\rangle_F \circ f$ in order to conclude whether fusion is desirable or not. Performing such an analysis we can conclude, for instance, that `tails . down` is a bad case while `drop2While p . map f` and `dropWhile p . tails` are good ones. □

5. FUSION IN PRACTICE

Our interest in studying generalized paramorphisms has arisen in the context of the development of HFUSION, a fusion tool for Haskell programs that is a reimplementation and an extension of the HYLO system [13]. The tool essentially translates recursive function definitions written in Haskell into hylomorphisms and then applies acid rain laws of hylomorphism to function compositions indicated by the user. This explains our special attention in the acid rains laws for the different recursion schemes.

The reasons for translating all recursive functions into hylomorphisms is twofold. One is due to the expressive power of hylomorphism, in the sense that all other recursive program schemes can be written in terms of it. The other reason is simplicity: translating all recursive functions into hylomorphisms, the internal engine needs manipulate only one form of recursion and thus implement only a few fusion laws and restructuring algorithms.

During the implementation of the kernel of the tool we started experimenting with some examples that were fusable by our implementation (modulo some simple modifications to the internal representation of hylomorphisms), but were impossible to be fused with the original representation and laws. We wanted then to give an explanation of these modifications at the abstract level, and it was during that process that the notion of generalized paramorphism came up as the appropriate abstraction that reflects the class of special cases we were playing with. With these modifications, the tool essentially interprets every recursive function as a generalized paramorphism. This explains our definition of generalized paramorphism.

The equivalence in the expressive power between hylomorphism and generalized paramorphism (witnessed by equations (19) and (20)) permits us to assure that we are not loosing fusion cases with the introduction of generalized paramorphism. On the contrary, we gain new cases captured by para-hylo fusion. We illustrate this by means of an example. Consider again the function compositon presented in Example 3.7: `repf x y p = replace x y . filter p`. This composition corresponds to a successful case of fusion. The main reason for the success is the fact of having viewed `replace` as a paramorphism. If, on the contrary, we view this function as a hylomorphism, then fusion fails. Let us see the reason. The definition of `replace` as a hylomorphism is:

$$\texttt{replace x y} = [\![\phi_1 \triangledown \phi_2, \psi]\!]_{G_a}$$

where $\psi = L_a\, \Delta \circ out_{L_a} :: [a] \to G_a[a]$ and $G_a\, x = L_a(x \times [a]) = 1 + a \times (x \times [a])$. Expanding the definition of ψ we obtain:

```
ψ = λas. case as of
          []      -> inl ()
          (a:as') -> inr (a,(as',as'))
```

The coalgebra ψ is not exactly out_{L_a}, but it contains it. Therefore, `replace` is a hylomorphism of the form $[\![\phi, \psi]\!]_{G_a}$, for $\psi \neq out_{L_a}$. On the other hand, `filter` is a fold $(\!|\tau(in)|\!)$ whose algebra can be expressed in terms of a polymorphic function τ. If we try to fuse these two functions as hylomorphisms, then the only law we could apply is fold-hylo fusion, but this is impossible because `replace` is not a fold.

6. CONCLUSIONS AND FINAL REMARKS

In this paper we introduced a generalized version of paramorphism which has an expressive power equivalent to hylomorphism. We showed acid rain laws for both the generalized and the standard version of paramorphism. With the introduction of generalized paramorphism we gained new fusion cases that cannot be captured with the laws of hylomorphism.

However, there are also some negative aspects. In particular, we saw the existence of compositions involving paramorphisms that may lead to programs with worse performance by the application of fusion. Therefore, in the presence of paramorphisms one should first perform some analysis on the code in order to determine whether to apply fusion or not.

There are some other cases, like `insert x . mapT f`, where fusion deforests just a single path from the root to the leaves. This is due to the fact that a paramorphism not only traverses its input, but also keeps it for computing the outcome. So in this case only a small amount of the intermediate data structure was eliminated. Nonetheless, fusion with paramorphisms may be good for bringing other functions together. For example, after fusing `map g . replace x y . filter q` we will have `map g . filter q` in the body of the resulting function.

Concerning HFUSION, we still owe a benchmark where to test on real programs the effectiveness of our approach based on generalized paramorphisms. We also need to implement a code analysis to avoid bad fusion cases.

Acknowledgements We thank the anonymous referees for their comments and suggestions regarding contents and presentation.

REFERENCES

[1] L. Augusteijn. Sorting Morphisms. In *Advanced Functional Programming,* LNCS 1608. Springer-Verlag, 1999.

[2] R. Bird. *Introduction to Functional Programming using Haskell (*2nd edition*)*. Prentice-Hall, UK, 1998.

[3] V. Capretta, T. Uustalu, and V. Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, 2006.

[4] M. Fokkinga and E. Meijer. Program Calculation Properties of Continuous Algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

[5] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, The Netherlands, 1992.

[6] J. Gibbons. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction,* LNCS 2297, pages 148–203. Springer-Verlag, January 2002.

[7] J. Gibbons. Fission for Program Comprehension. In *Intl. Conf. on Mathematics of Program Construction (MPC 2006),* LNCS ??? Springer-Verlag, 2006.

[8] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *Proc. 3rd. ACM SIGPLAN International Conference on Functional Programming*. acm, September 1998.

[9] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[10] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Dept. of Computer Science, University of Groningen, The Netherlands, 1990.

[11] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[12] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of Functional Programming Languages and Computer Architecture'91,* LNCS 523. Springer-Verlag, August 1991.

[13] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *Algorithmic Languages and Calculi*, pages 76–106, 1997.

[14] A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Proceedings of Functional Programming Languages and Computer Architecture'95*, 1995.

[15] P. Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.

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