Exploiting algebra/coalgebra duality for program fusion extensions

Facundo Domínguez
Intituto de Computación
Universidad de la República
Montevideo, Uruguay
fdomin@fing.edu.uy

Alberto Pardo
Intituto de Computación
Universidad de la República
Montevideo, Uruguay
pardo@fing.edu.uy

ABSTRACT

We reformulate algorithms for optimizing functional programs through a well known fusion technique. The reformulation sheds a new perspective which simplifies significantly the extensions to cope with programs involving mutually recursive definitions and recursion over multiple arguments. The presentation is based on a recursion scheme known as hylomorphism but other related fusion techniques may benefit from the results. Our algorithms are implemented as part of a fusion tool called HFusion.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*; D.3.4 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Program and recursion schemes*

General Terms

Languages, Algorithms, Theory

Keywords

Program Fusion, Program Transformation, Hylomorphism, Functional Programming

1. INTRODUCTION

Most often, programs are written as a composition of modular components. This makes it possible to take advantage of the well-known benefits of modular programming, such as readability and maintainability. Consider, for example, the following program written in Haskell

$$f\ p\ n = \mathit{sum}\ (\mathit{filter}\ p\ [1\,..\,n])$$

This program creates a list with the integers from 1 to n, then creates another list by selecting the integers that satisfy a given predicate p, and finally yields the sum of all of them. This compositional style of programming is suitable from a design perspective, but it is not desirable from the runtime standpoint because the intermediate lists imply extra work for allocating, examining and deallocating their nodes.

Such intermediate data structures can be automatically removed by using program transformation techniques, known as *deforestation* or *fusion*, in which modular programs are replaced by monolithic ones that compute the same without generating intermediate structures (see e.g. [12, 20, 28]).

In this paper we present a reformulation and extension of algorithms for a fusion tool which internally represents programs in terms of a recursive scheme called hylomorphism and applies certain fusion laws known as *acid rain* [26].

Firstly, we offer in Section 4 a concise formulation of the algorithms for deriving algebra/coalgebra transformers presented in [20]. Our formulation makes the fact evident that both algorithms are each other's dual, something not possible to be appreciated in previous formulations. In its original statement, the algorithm for deriving coalgebra transformers was unnecessarily complex as well as technically incorrect as it will be discussed.

Secondly, to showcase the practicality of our formulation beyond the cosmetics, we will extend our algorithms to handle a broader class of functions, namely functions that recurse over multiple arguments (Section 5) and mutually recursive functions (Section 6). Although previous work [14, 16] has already approached these extensions, as far as we are aware the actual algorithms had not been presented before. Moreover, in the case of the extension for recursive functions over multiple arguments addressed in [14], we formulate a different solution which in our opinion solves the same problem while being theoretically simpler.

While our formulation of the algorithms is based on our work using hylomorphisms, we expect the ideas presented herein to benefit the design and implementation of fusion systems based on other approaches, mostly those related to shortcut fusion [12] which could rely on the ability to abstract constructors away from function definitions.

The reformulated algorithms and extensions are part of the development of a fusion tool for Haskell programs called HFusion [13], which started being a partial reimplementation of the HYLO system [20, 24]. An extended presentation of our contributions can be found in [5].

2. THEORETICAL FRAMEWORK

Systems intended to automatically fuse programs often rely on laws expressed over particular representations of the functions of a program. In our approach, the function representation is a recursive scheme known as hylomorphism, which can be automatically converted back and forth to the textual source code representation of the functions. Our implementation HFusion converts recursive functions from a subset of Haskell to hylomorphisms,[1] applies the fusion laws and transforms the result back to Haskell. As all techniques to be described assume that programs are represented in terms of hylomorphisms, we will briefly present them here.

Hylomorphism is a program scheme that represents a function $f :: a \to b$ by splitting its definition into three components, written as $[\![\phi, \psi]\!]_F$, such that ψ describes how the arguments to the recursive calls of f are computed from the input value, ϕ describes how the results of the recursive calls are combined to build the output value, and F (called a functor) captures the essential structure of f. Following, we describe the main characteristics of each of these components and how they play together in the definition of hylomorphism. We also present the three fusion laws associated with hylomorphisms which constitute the core of the approach. Throughout we will assume a cpo semantics in terms of pointed cpos.

Functors.

Both the structure of datatypes and functions can be described using functors. A *functor* F is an operator that applies to types and functions, and satisfies the following properties: $F\ f :: F\ a \to F\ b$, for all $f :: a \to b$, $F\ id = id$, and $F\ (f \circ g) = F\ f \circ F\ g$.

Sometimes it is necessary to consider functors on two variables: A *bifunctor* F is a binary operator that applies to pairs of types and functions, such that: $F\ (f, g) :: F\ (a, c) \to F\ (b, d)$, for all $f :: a \to b$ and $g :: c \to d$, $F\ (id, id) = id$, and $F\ (f \circ g, h \circ k) = F\ (f, h) \circ F\ (g, k)$.

We write (a, b) to denote pairs of types in our meta-language, whereas by $a \times b$ we denote the type of pairs that can be written within programs.

Functors will be specified in compact notation as compositions of a set of elemental functors. The identity functor is given by $I\ a = a$ and $I\ f = f$, whereas for any type c the constant functor $\overline{c}$ is such that $\overline{c}\ a = c$ and $\overline{c}\ f = id$. The product type constructor $a \times b$ can be treated as a bifunctor by defining $f \times g :: a \times b \to c \times d$ as $(f \times g)\ (a, b) = (f\ a, g\ b)$.

In our meta-language we introduce a sum type constructor $a + b$ which builds the disjoint sum of two types. Semantically, this means that $a + b = (\{1\} \times a) \cup (\{2\} \times b) \cup \{\bot\}$. It can be treated as a bifunctor by defining $f + g :: a + b \to c + d$ as the strict function such that $(f + g)\ (1, a) = (1, f\ a)$ and $(f + g)\ (2, b) = (2, g\ b)$. Associated with sums it can be defined a case analysis operator $f \triangledown g :: a + b \to c$ given by the strict function such that $(f \triangledown g)\ (1, a) = f\ a$ and $(f \triangledown g)\ (2, b) = g\ b$.

Products and sums can be generalized to n components in the obvious way. We assume that application has greater precedence than both product and sum, and product has greater precedence than sum. By $\mathbf{1}$ we denote the unit type. We will also use products and sums of functors, that is, $(F * G)\ a = F\ a * G\ a$ and $(F * G)\ f = F\ f * G\ f$, for $* \in \{\times, +\}$.

An example of a functor expressed in compact form is $F = \overline{\mathbf{1}} + \overline{Int} \times I$. We call *recursive positions* of a functor those positions where functor I occurs.

[1]Most notably, HFusion does not handle programs using the *seq* operator yet.

Data types.

Given a functor F and any type a, a function of type $F\ a \to a$ is called an *F-algebra*, whereas a function of type $a \to F\ a$ is called an *F-coalgebra*. The type a is said to be the carrier set of the algebra/coalgebra.

Semantically speaking, recursive datatypes correspond to least fixed points of functors. Given a datatype declaration, it is possible to derive a functor F such that the datatype is the least solution to the equation $x \cong Fx$ and it is usually written μF. The isomorphism is provided by two strict functions, called $in_F :: F\ \mu F \to \mu F$ and $out_F :: \mu F \to F\ \mu F$, inverses of each other. The algebra in_F packs the constructors of the datatype (see [9] for further details).

A functor may be derived from a data type definition by extracting the arity of its constructors. For instance, by placing in a sum the signatures of the constructors of the type of lists **data** $[a] = [\,] \mid a : [a]$ we obtain the following functor $F = \overline{\mathbf{1}} + \overline{a} \times I$,[2] algebra in_F and coalgebra out_F.

```
$in_F :: F\ [a] \to [a]$
$in_F = nil \triangledown uncurry\ (:)$
  **where** $nil\ \_ = [\,]$
        $uncurry\ f\ (x, y) = f\ x\ y$

$out_F :: [a] \to F\ [a]$
$out_F\ [\,]\ \ \ \ \ \ = (1, ())$
$out_F\ (x : xs) = (2, (x, xs))$
```

Hylomorphisms.

Given a functor F, an algebra $\phi :: F\ b \to b$, and a coalgebra $\psi :: a \to F\ a$, the *hylomorphism* $[\![\phi, \psi]\!]_F :: a \to b$ is defined as the least fixed point of the equation $h = \phi \circ F\ h \circ \psi$.

Given an F-algebra of the form

$$\phi_1 \triangledown \cdots \triangledown \phi_n \quad \textbf{where}\ \phi_i\ (v_{i1}, \ldots, v_{in_i}) = t_i$$

we call v_{ij} a *recursive variable* if it corresponds to a recursive position of functor F, attending the fact that the variable holds the result of a recursive call.

The F-coalgebras we will manipulate are of the form:

```
$\lambda v_0 \to$ **case** $t_0$ **of**
        $p_1 \to (1, (t_{11}, \ldots, t_{1n_1}))$
            $\vdots$
        $p_m \to (m, (t_{m1}, \ldots, t_{mn_m}))$
```

We will call t_{ij} a *recursive term* if it corresponds to a recursive position of functor F, attending the fact that a recursive t_{ij} is the argument of a recursive call.

The algebra in_F and the coalgebra out_F allow specializing hylomorphisms to get two well-known recursion schemes called *fold* $(\!|-|\!)_F$ [2] and *unfold* $[\!(-)\!]_F$ [10]: $(\!|\phi|\!)_F = [\![\phi, out_F]\!]_F$ and $[\!(\psi)\!]_F = [\![in_F, \psi]\!]_F$.

Fusion laws.

The following fusion laws are instantiations of the so-called *acid rain laws* [26, 20] for hylomorphisms. Functions $\tau :: \forall a.\ (F\ a \to a) \to (G\ a \to a)$ and $\sigma :: \forall a.\ (a \to G\ a) \to (a \to F\ a)$ are called *transformers* of algebras and coalgebras, respectively [7].

$$(\!|\phi|\!)_F \circ [\!(\psi)\!]_F = [\![\phi, \psi]\!]_F \qquad \text{(FOLD-UNFOLD)}$$

$$(\!|\phi|\!)_F \text{ is strict } \Rightarrow (\!|\phi|\!)_F \circ [\![\tau(in_F), \psi]\!]_G = [\![\tau(\phi), \psi]\!]_G \qquad \text{(FOLD-HYLO)}$$

$$[\![\phi, \sigma(out_G)]\!]_F \circ [\!(\psi)\!]_G = [\![\phi, \sigma(\psi)]\!]_F \qquad \text{(HYLO-UNFOLD)}$$

[2]Formally, when the data type has a type parameter a, the functor is written as F_a. However, we avoid writing the type parameter for the sake of simplicity in notation.

3. ABSTRACTING CONSTRUCTORS AWAY

To illustrate the fusion laws in action let us consider the following program.

$$
\begin{array}{ll}
\mathit{mi}\ f\ e = \mathit{map}\ f \circ \mathit{intersp}\ e & \mathit{intersp} :: a \to [\,a\,] \to [\,a\,] \\
\mathit{map} :: (a \to b) \to [\,a\,] \to [\,b\,] & \mathit{intersp}\ e\ [\,] \qquad\quad = [\,] \\
\mathit{map}\ f\ [\,] \qquad\quad = [\,] & \mathit{intersp}\ e\ (x : [\,])\ = x : [\,] \\
\mathit{map}\ f\ (x : \mathit{xs}) = f\ x : \mathit{map}\ f\ \mathit{xs} & \mathit{intersp}\ e\ (x : \mathit{xs}) = \\
 & \quad x : e : \mathit{intersp}\ e\ \mathit{xs}
\end{array}
$$

If we want to fuse $\mathit{map}\ f \circ \mathit{intersp}\ e$, we first derive hylomorphisms with functors $F = \overline{\mathbf{1}} + \overline{a} \times I$ and $G = \overline{\mathbf{1}} + \overline{a} + \overline{a} \times I$.

$$
\begin{array}{l}
\mathit{map}\ f = (\!|\gamma|\!)_F \\
\quad \textbf{where}\ \gamma = \mathit{nil} \triangledown (\lambda(x, v) \to f\ x : v) \\
\mathit{intersp}\ e = [\![\phi, \psi]\!]_G \\
\quad \textbf{where}\ \phi = \mathit{nil} \triangledown (:[\,]) \triangledown (\lambda(x, v) \to x : e : v) \\
\qquad\qquad \psi = \lambda v \to \textbf{case}\ v\ \textbf{of}\ \begin{array}[t]{ll} [\,] & \to (1, ()) \\ x : [\,] & \to (2, x) \\ x : \mathit{xs} & \to (3, (x, \mathit{xs})) \end{array}
\end{array}
$$

In this case we are able to derive a τ transformer for *intersp*, obtaining:

$$
\begin{array}{l}
\mathit{intersp}\ e = [\![\tau(\mathit{in}_F), \psi]\!]_G \\
\quad \textbf{where}\ \tau :: \forall a.\ (F\ a \to a) \to (G\ a \to a) \\
\qquad\qquad \tau(\alpha) = \tau_1(\alpha) \triangledown \tau_2(\alpha) \triangledown \tau_3(\alpha) \\
\qquad\qquad \tau_1(\alpha_1 \triangledown \alpha_2)\ () \qquad\ = \alpha_1 \\
\qquad\qquad \tau_2(\alpha_1 \triangledown \alpha_2)\ x \qquad\ = \alpha_2\ (x, \alpha_1\ ()) \\
\qquad\qquad \tau_3(\alpha_1 \triangledown \alpha_2)\ (x, v) = \alpha_2\ (x, \alpha_2\ (e, v))
\end{array}
$$

Applying *acid rain* for the FOLD-HYLO case we get $\mathit{mi}\ f\ p = [\![\tau(\gamma), \psi]\!]_G$, which converted back to a Haskell recursive definition is:

$$
\begin{array}{l}
\mathit{mi}\ f\ e\ [\,] \qquad\quad = [\,] \\
\mathit{mi}\ f\ e\ (x : [\,])\ = f\ x : [\,] \\
\mathit{mi}\ f\ e\ (x : \mathit{xs}) = f\ x : f\ e : \mathit{mi}\ f\ e\ \mathit{xs}
\end{array}
$$

As another example we could consider the composition

$$\mathit{im}\ e\ f = \mathit{intersp}\ e \circ \mathit{map}\ f$$

where we would need to write *map f* as an unfold and rewrite the coalgebra of *intersp* using a σ transformer:

$$
\begin{array}{l}
\mathit{map}\ f = [\!(\psi')\!]_F \\
\quad \textbf{where}\ \psi' = \lambda v \to \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad [\,] \to (1, ()) \\
\qquad\qquad\qquad\qquad x : \mathit{xs} \to (2, (f\ x, \mathit{xs})) \\
\mathit{intersp}\ e = [\![\phi, \sigma(\mathit{out}_F)]\!]_G \\
\quad \textbf{where}\ \sigma :: \forall a.\ (a \to F\ a) \to (a \to G\ a) \\
\qquad\qquad \sigma(\beta)\ v = \textbf{case}\ \beta\ v\ \textbf{of} \\
\qquad\qquad\qquad (1, ()) \qquad\quad \to (1, ()) \\
\qquad\qquad\qquad (2, (x, \mathit{xs})) \to \textbf{case}\ \beta\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (1, ()) \to (2, x) \\
\qquad\qquad\qquad\qquad\qquad\qquad \_ \qquad\ \to (3, (x, \mathit{xs}))
\end{array}
$$

The result of applying *acid rain* for the HYLO-UNFOLD case is $[\![\phi, \sigma(\psi')]\!]_G$, which can be written in Haskell as:

$$
\begin{array}{l}
\mathit{im}\ e\ f\ [\,] \qquad\quad = [\,] \\
\mathit{im}\ e\ f\ (x : [\,])\ = f\ x : [\,] \\
\mathit{im}\ e\ f\ (x : \mathit{xs}) = f\ x : e : \mathit{im}\ e\ f\ \mathit{xs}
\end{array}
$$

Both examples involve a step where transformers τ or σ need to be derived. These derivations involve mostly abstracting constructors away, so they can later be replaced by changing the argument of the transformer. In the case of the derivation of τ, constructors are abstracted on the expressions of an algebra. In the case of the derivation of σ, the constructors are abstracted from the patterns of a coalgebra.

Despite the duality that relates transformers of algebras and coalgebras, the treatment in these examples has been quite asymmetrical. When writing a σ for the coalgebra of *intersp* we had to rewrite the coalgebra into a cascade of cases while no dual task was needed to derive τ. This is more a nuance of Haskell rather than one of the theoretical tools we are employing.

In order to improve the expression of coalgebra transformers, we will resort to an idiom known as *view patterns*, an idea first proposed by Wadler [27] which has been added as an extension to the Glasgow Haskell Compiler.[3] A view pattern is a pattern written $t{\cdot}p$ where t is a term and p some other pattern. To match a value v against a pattern $(t{\cdot}p)$, first $(t\ v)$ is evaluated, and then the result is matched against p. Thus, having a function $f\ x = x + 1$, the expression **case** 1 **of** $f{\cdot}2 \to 0$ evaluates to 0 because it is the same as evaluating **case** $f\ 1$ **of** $2 \to 0$.

The convenience of view patterns can be appreciated in a transformer of coalgebras.

$$
\begin{array}{l}
\mathit{intersp}\ e = [\![\phi, \sigma(\mathit{out}_F)]\!]_G \\
\quad \textbf{where}\ \sigma :: \forall a.\ (a \to F\ a) \to (a \to G\ a) \\
\qquad\qquad \sigma(\beta) = \lambda v \to \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad \beta{\cdot}(1, ()) \to (1, ()) \\
\qquad\qquad\qquad\qquad \beta{\cdot}(2, (x, \beta{\cdot}(1, ()))) \to (2, x) \\
\qquad\qquad\qquad\qquad \beta{\cdot}(2, (x, \mathit{xs})) \qquad\quad \to (3, (x, \mathit{xs}))
\end{array}
$$

Note how this definition of σ resembles more closely the definition of the original coalgebra ψ of *intersp*. In this rewriting, the constructors in the patterns of ψ have been literally replaced with a variable, much in the same way we did with the constructors in the algebra of *intersp* when deriving τ. The idiom of view patterns has allowed us to skip the asymmetrical step of writing the cascade of cases.

One may wonder if using view patterns in successive alternatives of a case as we are doing here may not evaluate more than once the coalgebra β over the same value v. However, such redundancy can be automatically avoided when writing hylomorphisms back to Haskell.

4. DERIVATION ALGORITHMS FOR TRANSFORMERS

We focus now on the derivation algorithms for algebra and coalgebra transformers. We show a pair of algorithms for deriving τ and σ, where one is the dual of the other. The simplicity of the formulation will enable us to extend the algorithms with a minimal effort to handle broader classes of functions.

For the sake of this presentation, we will define our algorithms over a simple sub-language comprised of lambda abstractions, application, cases over constructors and view patterns.

Derivation of τ

The derivation algorithm for τ is in essence the same given by Onoue et al. [20]. We present it here only because it is an interesting point of comparison with the algorithm we propose in the next subsection for deriving σ transformers. Given a G-algebra ϕ the goal of the algorithm is to determine whether it can be written as $\tau(\mathit{in}_F)$, for $\tau :: \forall a.(F\ a \to a) \to (G\ a \to a)$. The algorithm requires that ϕ be given by a case analysis $\phi_1 \triangledown \cdots \triangledown \phi_n$ such that each ϕ_i is a function $\lambda(v_1, \ldots, v_{k_i}) \to t_i$ where the term t_i satisfies the following normal form:

1. it is a recursive variable; or

2. it is a constructor application $C_j\ (t'_1, \ldots, t'_m)$ where each t'_k in a recursive position of constructor C_j is in

[3] `http://www.haskell.org/ghc/`.

$$\begin{array}{l}
\mathcal{T}(F, \phi :: G\ \mu F \to \mu F) :: (F\ a \to a) \to (G\ a \to a) \\
\mathcal{T}(F_1 + \cdots + F_m, \phi_1 \triangledown \cdots \triangledown \phi_n) = \\
\quad \lambda(\alpha_1 \triangledown \cdots \triangledown \alpha_m) \to \mathcal{T}'(\phi_1) \triangledown \cdots \triangledown \mathcal{T}'(\phi_n) \\
\quad \textbf{where}\ \mathcal{T}'(\lambda bvs \to t) = \lambda bvs \to \mathcal{A}(t) \\
\qquad\qquad \mathcal{A}(v) = v \quad \text{if } v \text{ is a recursive variable} \\
\qquad\qquad \mathcal{A}(C_j\ (t_1, \ldots, t_k)) = \alpha_j\ ((F_j\ \mathcal{A})\ (t_1, \ldots, t_k)) \\
\qquad\qquad \mathcal{A}(t) = (\!|\alpha_1 \triangledown \cdots \triangledown \alpha_m|\!)_F\ t \quad \text{every other case}
\end{array}$$

Figure 1: Derivation algorithm for τ

$$\begin{array}{l}
\mathcal{S}(F, \psi :: \mu F \to G\ \mu F) :: \forall a.(a \to F\ a) \to (a \to G\ a) \\
\mathcal{S}(F, \lambda v \to \textbf{case}\ v\ \textbf{of}\ \{p_1 \to t_1; \ldots; p_n \to t_n\}) = \\
\quad \lambda\beta \to \lambda v \to \textbf{case}\ v\ \textbf{of}\ \{\mathcal{B}\ (p_1) \to t_1; \ldots; \mathcal{B}(p_n) \to t_n\} \\
\quad \textbf{where}\ F_1 + \cdots + F_m = F \\
\qquad\qquad \mathcal{B}(v) = v \quad \text{if } v \text{ is a recursive variable} \\
\qquad\qquad \mathcal{B}(C_j\ (p_1, \ldots, p_k)) = \beta \cdot (j, (F_j\ \mathcal{B})\ (p_1, \ldots, p_k)) \\
\qquad\qquad \mathcal{B}(p) = [\![\beta]\!]_F \cdot p \quad \text{every other case}
\end{array}$$

Figure 2: Derivation algorithm for σ

normal form. By *recursive positions* of a constructor we mean those positions where the corresponding datatype μF occurs recursively; they correspond to the occurrences of the I functor in the expression of functor F. Those t'_k that are not in recursive positions of C_j can be any term not referencing recursive variables; or

3. it is any term not referencing recursive variables.

When there are **if-then-else** or **case** structures embedded in ϕ, they sometimes can be moved out of the algebra by restructuring the hylomorphism in order to obtain this normal form.

The derivation algorithm for τ is shown in Figure 1. The objective of algorithm $\mathcal{A}$ is to abstract constructors, substituting them for the corresponding operations of the F-algebra $\alpha = \alpha_1 \triangledown \cdots \triangledown \alpha_m$. This algorithm is applied recursively to the recursive arguments only, which are indicated by the functor F required as an input. The output of the algorithm is such that $\mathcal{T}(F, \phi)\ (in_F) = \phi$.

Derivation of σ

The derivation algorithm for σ takes as input a G-coalgebra to be rewritten as $\sigma(out_F)$ where $\sigma :: \forall a.(a \to F\ a) \to (a \to G\ a)$. The input coalgebra must be in the form: $\lambda v \to$ **case** v **of** $\{p_1 \to t_1; \ldots; p_n \to t_n\}$, that is, the **case** must be evaluated over the input variable. There are also the following restrictions:

- Recursive terms must be variables, and non-recursive terms must not contain variables appearing in recursive terms.
- The patterns p_i must satisfy the following normal form:
  1. the pattern is a variable; or
  2. the pattern is of the form $C_i\ (p'_1, \ldots, p'_{k_i})$ and pattern p'_j appearing in a recursive position of C_i is in normal form. A pattern p'_j in a non-recursive position can have any shape as far as it does not reference variables appearing in recursive terms. A pattern p'_j is said to appear in a recursive position if the functor F (not G) tells so, being F the functor characterizing the input datatype μF of the coalgebra.

In Figure 2 we present our derivation algorithm for σ. Algorithm $\mathcal{B}$, which abstracts constructors inside patterns, here plays the same role as algorithm $\mathcal{A}$ in the derivation algorithm for τ. Duality can now be better appreciated from the textual presentation of the algorithms, something that was not evident in previous formulations [20, 24].

Initially, we intended to base our implementation on the algorithm described by Onoue et al. [20] and that is used in the HYLO system. But we realized later that this algorithm, as originally formulated, is not correct since it changes the semantics of the coalgebra. In Haskell, a pattern is a tree-like structure whose nodes are matched in pre-order against a value. We have been careful to preserve that order in our proposal. In contrast, in [20] checks are reorganized in such a way that they are performed in breadth-first order, changing thus the behavior of the functions in the presence of partially defined arguments. Consider for instance a coalgebra over binary trees:

$$\begin{array}{l}
\textbf{data}\ \mathit{Tree} = \mathit{Node}\ \mathit{Tree}\ \mathit{Tree} \mid \mathit{Empty} \\
\psi\ (\mathit{Node}\ (\mathit{Node}\ (\mathit{Node}\ l\ r)\ \_)\ \mathit{Empty}) = (1, (l, r)) \\
\psi\ \_ \qquad\qquad\qquad\qquad\qquad\qquad\quad = (2, ())
\end{array}$$

Calling $\psi\ (\mathit{Node}\ (\mathit{Node}\ \mathit{Empty}\ \_)\ \bot)$ yields $(2, ())$, but would yield $\bot$ if patterns are checked in breadth-first order.

5. RECURSION OVER MULTIPLE ARGUMENTS

The first extension we consider corresponds to compositions that involve functions which recurse over multiple arguments. Classical examples are functions like *zip*, *zipWith* or equality for recursive datatypes [1]. Functions of this kind can be represented as hylomorphisms. For example, let us consider function $\mathit{zip} :: [a] \times [b] \to [a \times b]$:[4]

$$\begin{array}{l}
\mathit{zip}\ (x : xs, y : ys) = (x, y) : \mathit{zip}\ (xs, ys) \\
\mathit{zip}\ (\_, \_) \qquad\quad\ \ = [\,]
\end{array}$$

It can be written as the unfold $[\![\psi]\!]_G$ where:

$$\begin{array}{l}
G = \overline{1} + \overline{a \times b} \times I \\
\psi :: [a] \times [b] \to G\ ([a] \times [b]) \\
\psi\ (x : xs, y : ys) = (2, ((x, y), (xs, ys))) \\
\psi\ (\_ \qquad , \_ \qquad) = (1, ())
\end{array}$$

Whereas the fusion laws expect a single intermediate data structure to eliminate, in a composition like $\mathit{zip} \circ (\mathit{map}\ f \times \mathit{id})$ the intermediate structure comes as a component of the input pair. This problem could be circumvented if we expressed *zip* as a higher-order fold $(\!|\phi|\!) :: [a] \to ([b] \to [a \times b])$ that recurses on the first list and returns a function as result, and we then fuse $(\!|\phi|\!) \circ \mathit{map}\ f$. Similarly, we could fuse $\mathit{zip} \circ (\mathit{id} \times \mathit{map}\ f)$ by considering a higher-order fold on the second list. The problem with this approach is that the representation of a function with multiple arguments may need to be radically different depending on the compositions in which it occurs.

Hu et al. [14] studied the problem of how to deal with functions with multiple arguments. They proposed a special operator to express coalgebras that take pairs of values as input and extended the fusion laws to cope with it.

The solution we developed is different from the two above, and can be considered similar in power to what Svenningsson

[4]For the sake of presentation, functions that recurse over multiple arguments will be written in uncurried form for compatibility with their representation as hylomorphisms.

proposed for the shortcut fusion approach [25]. We maintain the hylomorphism corresponding to the function with multiple arguments mostly unchanged and derive the appropriate transformer σ for its coalgebra. Our approach is based on the following law.

LEMMA 1. $$\frac{\sigma :: \forall a.\ (a \to F\ a) \to (H\ a \to G\ (H\ a))}{[\![\phi, \sigma(out_F)]\!]_G \circ H\ [\!(\psi)\!]_F = [\![\phi, \sigma(\psi)]\!]_G}$$

The main difference with the HYLO-UNFOLD law is that the intermediate structure generated by the producer comes embedded in a structure described by the functor H. A proof of this lemma can be found in [5].

EXAMPLE 1. *To see this law in action, let us consider the composition zipmap f = zip $\circ$ (map $f \times id$). Function map can be written as an unfold $[\!(\psi')\!]_F$ like we showed in Section 3. By rewriting the hylomorphism for zip shown above in terms of a transformer σ we obtain $[\!(\sigma(out_F))\!]_G$, where:*

$$\begin{array}{l} \sigma :: \forall z.(z \to F\ z) \to (z \times [\,b\,] \to G\ (z \times [\,b\,])) \\ \sigma(\beta)\ (x,y) = \textbf{case}\ (x,y)\ \textbf{of} \\ \qquad (\beta\cdot(2,(x,xs)), y : ys) \to (2, ((x,y),(xs,ys))) \\ \qquad (\_ \qquad\qquad\qquad , \_ \quad\ ) \to (1, ()) \end{array}$$

Applying Lemma 1 with $H = I \times \overline{[\,b\,]}$, we obtain zipmap f = $[\!(\sigma(\psi'))\!]_G$. Inlining,

$$\begin{array}{l} zipmap :: (c \to a) \to [\,c\,] \times [\,b\,] \to [\,a \times b\,] \\ zipmap\ f\ (x : xs, y : ys) = (f\ x, y) : zipmap\ f\ (xs, ys) \\ zipmap\ f\ (\_,\_) \qquad\qquad = [\,] \end{array}$$

□

In [14], the focus was on how to apply fusion on all arguments of a function simultaneously. In contrast, our solution is selective in the argument we want to fuse, and this is crucial to enable fusion of functions with accumulating parameters as we will see shortly. Nevertheless, in case the consumer is composed with a product of several producers, like e.g. $zip \circ (map\ f \times map\ g)$, we can simply proceed in multiple steps by splitting the product: $zip \circ (map\ f \times id) \circ (id \times map\ g)$. Then, in our case, we first fuse $zip \circ (map\ f \times id)$ as in Example 1, and then we fuse the result with $(id \times map\ g)$.

Functions that use accumulators or recurse over parameters of non-recursive types can also be represented as hylomorphisms over multiple arguments. Examples of them are *take*, *drop*, and *foldl* [1]. Let us consider the case of *foldl*:

$$\begin{array}{l} foldl :: (b \to a \to b) \to b \times [\,a\,] \to b \\ foldl\ f\ (e, [\,]) = e \\ foldl\ f\ (e, x : xs) = foldl\ f\ (f\ e\ x, xs) \end{array}$$

It can be written as the hylomorphism $foldl\ f = [\![id, \sigma(out_F)]\!]_G$, where:

$$\begin{array}{l} \sigma :: \forall z.\ (z \to F\ z) \to (b \times z \to G\ (b \times z)) \\ \sigma\ \beta\ (e,l) = \textbf{case}\ l\ \textbf{of}\ \beta\cdot(1,[\,]) \qquad \to (1, e) \\ \qquad\qquad\qquad\qquad\quad \beta\cdot(2,(x,xs)) \to (2, (f\ e\ x, xs)) \end{array}$$

for $G = \overline{b} + I$ and $F = \overline{\mathbf{1}} + \overline{a} \times I$, such that it can be fused with another hylomorphism on the list argument. For instance, if we fuse the composition $fm\ f\ g = foldl\ f \circ (id \times map\ g)$, we obtain $fm\ f\ g = [\![id, \sigma(\psi')]\!]_G$, being ψ' the same coalgebra for *map* described in Section 3. Inlining,

$$\begin{array}{l} fm\ f\ g\ (e, [\,]) \qquad = e \\ fm\ f\ g\ (e, x : xs) = fm\ f\ g\ (f\ e\ (g\ x), xs) \end{array}$$

Note that a transformer $\sigma :: \forall z.\ (z \to F\ z) \to (z \times a \to G\ (z \times a))$, for some functor F, cannot be derived from the definition of *foldl*, which would be needed to fuse compositions on the argument e. This means that in this case we cannot perform fusion on the accumulator position, so the fact reinforces the importance of having a law that is selective in the arguments considered for fusion.

Derivation of σ

For the sake of clarity we only discuss how to derive a transformer σ which enables fusion on the first argument of a definition having two arguments. Generalizing the algorithm to any amount of arguments and to fusion over any of them does not pose any substantial challenge.

Now, the input coalgebra is expected to be in the form:

$$\begin{array}{l} \psi :: a \times b \to G\ (a \times b) \\ \psi = \lambda v \to \textbf{case}\ v\ \textbf{of} \\ \qquad\qquad (p_{11}, p_{12}) \to (1, (t_{11}, ..., t_{1k_1})) \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad (p_{m1}, p_{m2}) \to (m, (t_{m1}, ..., t_{mk_m})) \end{array}$$

Every term t_{ij} in a recursive position of G must be of the form (v, t), where v is a variable being bound in a recursive position of a constructor in pattern p_{i1}, and t must not reference any such variable. The patterns p_{i1} must conform to the same normal form that we required for patterns in our original derivation algorithm for σ. The coalgebras for *zip* and *foldl* satisfy these restrictions.

The extended algorithm is the following:

$$\begin{array}{l} \mathcal{S}'(F, \lambda v \to \textbf{case}\ v\ \textbf{of} \\ \qquad\qquad \{(p_{11}, p_{12}) \to t_1; \ldots; (p_{m1}, p_{m2}) \to t_m)\}) = \\ \quad \lambda\beta \to \lambda v \to \textbf{case}\ v\ \textbf{of} \\ \qquad\qquad \{(\mathcal{B}(p_{11}), p_{12}) \to t_1; \ldots; (\mathcal{B}(p_{m1}), p_{m2}) \to t_m\} \end{array}$$

being $\mathcal{B}$ the same algorithm presented in Section 4. Algorithm $\mathcal{S}'$ now returns a transformer $\sigma :: \forall a.(a \to F\ a) \to (a \times b \to G\ (a \times b))$, such that $\psi = \sigma(out_F)$.

6. MUTUALLY RECURSIVE FUNCTIONS

The next extension we consider is the one that makes it possible to deal with mutually recursive functions. As an example, let us consider the function *rmostR* which extracts the rightmost leave of a finitary tree.

$$\begin{array}{l} \textbf{data}\ Rose\ a = Rose\ a\ [Rose\ a] \\ rmostR :: Rose\ a \to a \\ rmostR\ (Rose\ a\ [\,])\ = a \\ rmostR\ (Rose\ a\ xs) = rmostL\ xs \\ rmostL :: [Rose\ a] \to a \\ rmostL\ (x : [\,])\ = rmostR\ x \\ rmostL\ (x : xs) = rmostL\ xs \end{array}$$

Theoretically, the situation is handled by considering pairs of functions [16]. We now need to consider functors that take and return pairs. Those functors will be constructed as the split $\langle F, G\rangle$ of two bifunctors F and G such that $\langle F, G\rangle\ (a,b) = (F\ (a,b), G\ (a,b))$. It is also useful to consider the projection functors: $\Pi_1\ (a,b) = a$, $\Pi_2\ (a,b) = b$. Product, sum and constant functors will be overloaded to work over pairs of types and functions. For instance, $(a,b) \times (c,d) = (a \times c, b \times d)$. Also, it is necessary to generalize hylomorphisms to express mutually recursive definitions.

Let H be a functor from pairs to pairs. Let $\phi :: H\ (c,d) \to (c,d)$ be an H-algebra and $\psi :: (a,b) \to H\ (a,b)$ be an H-coalgebra. A *mutual hylomorphism* $[\![\phi, \psi]\!]_H$ is a pair of functions $(f :: a \to c, g :: b \to d)$ which is the least fix-point of the equation $(f,g) = \phi \circ H\ (f,g) \circ \psi$.

Now we can write *rmostR* and *rmostL* as a mutual hylomorphism:

$$\begin{array}{l} (rmostR, rmostL) = [\![(id \triangledown id, id \triangledown id), (\psi_1, \psi_2)]\!]_{\langle G_1, G_2\rangle} \\ \quad \textbf{where}\ G_1 = \overline{a} + \Pi_2 \\ \qquad\qquad G_2 = \Pi_1 + \Pi_2 \\ \qquad\qquad \psi_1 :: Rose\ a \to G_1\ (Rose\ a, [Rose\ a]) \\ \qquad\qquad \psi_2 :: Rose\ a \to G_2\ (Rose\ a, [Rose\ a]) \end{array}$$

$$\begin{array}{ll} \psi_1\ (Rose\ a\ [\,]) = (1,a) & \psi_2\ (x:[\,]) = (1,x) \\ \psi_1\ (Rose\ a\ xs) = (2,xs) & \psi_2\ (x:xs) = (2,xs) \end{array}$$

A nice characteristic of this generalization is that the properties of hylomorphism remain true, in particular the *acid rain* laws, with the characteristic that now the ingredients work on pairs.

In practice, one would have written *rmostR* without mutual recursion:

$$\begin{array}{l} rmostR\ (Rose\ a\ [\,]) = a \\ rmostR\ (Rose\ a\ xs) = last\ (map\ rmostR\ xs) \end{array}$$

This particular form of recursion cannot be treated with the algorithms we present here, but provisions can be taken to automatically derive the mutually recursive form from it: fuse the composition of $last \circ map\ rmostR$ and then specialize the result *last_map rmostR* for the non-recursive parameter *rmostR*. Such manipulations are effective in general for the functions that can be represented using hylomorphisms which have regular functors and HFusion can perform the needed steps [5]. Another function that could be subject to the same treatment is, for example:

$$\begin{array}{l} sumR :: Rose\ Int \to Int \\ sumR\ (Rose\ a\ xs) = a + sum\ (map\ sumR\ xs) \\[4pt] sum :: [Int] \to Int \\ sum\ [\,] = 0 \\ sum\ (x:xs) = x + sum\ xs \end{array}$$

We could express this function with mutual recursion by first fusing $sum \circ map\ sumR$,

$$\begin{array}{l} sumR\ (Rose\ a\ xs) = a + sm\ sumR\ xs \\[4pt] sm\ f\ [\,] = 0 \\ sm\ f\ (x:xs) = f\ x + sm\ f\ xs \end{array}$$

and then specializing *sm mapR* for the argument *mapR* obtaining

$$\begin{array}{l} sumR\ (Rose\ a\ xs) = a + sm\_sumR\ xs \\[4pt] sm\_sumR\ [\,] = 0 \\ sm\_sumR\ (x:xs) = sumR\ x + sm\_sumR\ xs \end{array}$$

One other motivation for fusing mutually recursive functions is that the normal recursive functions which contain nested constructor applications in either patterns or terms can be usually expressed as mutual folds and unfolds. For example, consider function *intersp*, shown in Section 3. The function *intersp* could be written either as a mutual fold

$$\begin{array}{l} intersp\ e = (\!|(\phi_1,\phi_2)|\!)_F \\ \quad \textbf{where}\ F = \langle \underline{\mathbf{1}} + \overline{a} \times \Pi_2, \underline{\mathbf{1}} + \overline{a} \times \Pi_2 \rangle \\ \qquad\quad \phi_1\ (1,()) \qquad\ \ = [\,] \\ \qquad\quad \phi_1\ (2,(x,vs)) = x : vs \\ \qquad\quad \phi_2\ (1,()) \qquad\ \ = [\,] \\ \qquad\quad \phi_2\ (2,(x,vs)) = e : x : vs \end{array}$$

which mimics the following definition of *intersp*:

$$\begin{array}{l} intersp :: a \to [a] \to [a] \\ intersp\ e\ [\,] \qquad\ \ = [\,] \\ intersp\ e\ (x:xs) = x : intersp'\ e\ xs \\[4pt] intersp'\ e\ [\,] \qquad\ \ = [\,] \\ intersp'\ e\ (x:xs) = e : x : intersp'\ e\ xs \end{array}$$

or it could be written as a mutual unfold

$$\begin{array}{l} intersp\ e = [\!(\psi_1,\psi_2)\!]_F \\ \quad \textbf{where}\ F = \langle \underline{\mathbf{1}} + \overline{a} \times \Pi_2, \underline{\mathbf{1}} + \overline{a} \times \Pi_1 \rangle \\ \qquad\quad \psi_1\ [\,] \qquad\quad = (1,()) \\ \qquad\quad \psi_1\ (x:xs) = (2,(x,xs)) \\ \qquad\quad \psi_2\ [\,] \qquad\quad = (1,()) \\ \qquad\quad \psi_2\ vs \qquad\ \ = (2,(e,vs)) \end{array}$$

which mimics this other definition:

$$\begin{array}{l} intersp :: a \to [a] \to [a] \\ intersp\ e\ [\,] \qquad\ \ = [\,] \\ intersp\ e\ (x:xs) = x : intersp'\ e\ xs \\[4pt] intersp'\ e\ [\,] = [\,] \\ intersp'\ e\ xs = e : intersp\ e\ xs \end{array}$$

The algorithms for rewriting a normal hylomorphism in terms of a mutually recursive form is something we have not addressed yet, though.

$$\begin{array}{l} \mathcal{T}(h,F,\phi :: G_h\ \mu F \to \mu F) \\ \qquad :: (F\ (a_1,a_2) \to (a_1,a_2)) \to G_h\ a_h \to a_h \\ \mathcal{T}(h,\langle F_1,F_2\rangle,\phi_1 \triangledown \cdots \triangledown \phi_n) = \\ \quad \lambda(\alpha_{11} \triangledown \cdots \triangledown \alpha_{1m_1}, \alpha_{21} \triangledown \cdots \triangledown \alpha_{2m_2}) \to \mathcal{T}'(\phi_1) \triangledown \cdots \triangledown \mathcal{T}'(\phi_n) \\ \quad \textbf{where}\ F_{i1} + \cdots + F_{im_i} = F_i \\ \qquad\qquad \mathcal{T}'(\lambda bvs \to t) = \lambda bvs \to \mathcal{A}_h\ t \\ \qquad\qquad \mathcal{A}_h(v) = v\ \text{(if } v \text{ is a recursive var. according to } G\text{)} \\ \qquad\qquad \mathcal{A}_1(C_j\ (t_1,\ldots,t_k)) = \alpha_{1j}\ ((F_{1j}\ (\mathcal{A}_1,\mathcal{A}_2))\ (t_1,\ldots,t_k)) \\ \qquad\qquad \mathcal{A}_2\ (C_j\ (t_1,\ldots,t_k)) = \alpha_{2j}\ ((F_{2j}\ (\mathcal{A}_1,\mathcal{A}_2))\ (t_1,\ldots,t_k)) \\ \qquad\qquad \mathcal{A}_h\ (t) = \Pi_h\ (\!|(\alpha_1,\alpha_2)|\!)_{\langle F_1,F_2\rangle}\ t\ \text{(all other cases)} \end{array}$$

Figure 3: Algorithm for deriving τ

$$\begin{array}{l} \mathcal{S}(h,F,\psi :: \mu F \to G_h\ \mu F) \\ \qquad :: ((a_1,a_2) \to F\ (a_1,a_2)) \to a_h \to G_h\ a_h \\ \mathcal{S}(h,\langle F_1,F_2\rangle,\lambda v \to \textbf{case}\ v\ \textbf{of}\ p_1 \to t_1;\ldots;p_n \to t_n) = \\ \quad \lambda(\beta_1,\beta_2) \to \lambda v \to \textbf{case}\ v\ \textbf{of} \\ \qquad\qquad\qquad\qquad \{\mathcal{B}_h\ (p_1) \to t_1;\ldots;\mathcal{B}_h(p_n) \to t_n\} \\ \quad \textbf{where}\ F_{i1} + \cdots + F_{im_i} = F_i \\ \qquad\qquad \mathcal{B}_h(v) = v\ \ \text{if } v \text{ is a recursive variable} \\ \qquad\qquad \mathcal{B}_h(C_j\ (p_1,\ldots,p_k)) = \\ \qquad\qquad\qquad \beta_h \cdot (j,(F_{hj}\ (\mathcal{B}_1,\mathcal{B}_2))\ (p_1,\ldots,p_k)) \\ \qquad\qquad \mathcal{B}_h(p) = \Pi_h\ [\!(\beta_1,\beta_2)\!]_{\langle F_1,F_2\rangle} \cdot p\ \ \text{all other cases} \end{array}$$

Figure 4: Algorithm for deriving σ

Derivation of τ

Having a mutual hylomorphism $[\![(\phi_1,\phi_2),\psi]\!]_{\langle G_1,G_2\rangle}$, we might want to derive an equivalent one of the form $[\![\tau(in_F),\psi]\!]_{\langle G_1,G_2\rangle}$ when composed with a mutual fold $(\!|\phi'|\!)_F$. Indeed, we will be deriving a transformer τ that is also given as a pair: $\tau(\alpha) = (\tau_1\ (\alpha), \tau_2\ (\alpha))$. This is how we obtain τ:

$$\begin{array}{l} \tau : \forall a.\ \forall b.\ (F\ (a,b) \to (a,b)) \to \langle G_1,G_2\rangle\ (a,b) \to (a,b) \\ \tau(\alpha) = (\mathcal{T}(1,F,\phi_1)(\alpha), \mathcal{T}(2,F,\phi_2)(\alpha)) \end{array}$$

where $\mathcal{T}$ is the algorithm presented in Figure 3. The auxiliary algorithm $\mathcal{A}$ must keep track of the datatype owning the constructor whose arguments are being traversed. This is necessary because the algebra of a mutual hylomorphism may contain nested applications of constructors of mixed types. For example, inside a value of type *Rose* we may find the application of constructors of type [*Rose a*].

Derivation of σ

Having a mutual hylomorphism $[\![\phi,(\psi_1,\psi_2)]\!]_{\langle G_1,G_2\rangle}$ we might want to derive an equivalent one of the form $[\![\phi,\sigma(out_F)]\!]_{\langle G_1,G_2\rangle}$ when composed with a mutual unfold $[\!(\psi'\!)]_G$. The transformer σ is calculated as follows:

$$\begin{array}{l} \sigma : \forall a.\ \forall b.\ ((a,b) \to F\ (a,b)) \to ((a,b) \to \langle G_1,G_2\rangle\ (a,b)) \\ \sigma(\beta) = (\mathcal{S}(1,F,\psi_1)(\beta), \mathcal{S}(2,F,\psi_2)(\beta)) \end{array}$$

where $\mathcal{S}$ is the algorithm presented in Figure 4. Once more, algorithm $\mathcal{S}$ is the dual of the derivation algorithm for τ. Through its sub-index, $\mathcal{B}_h$ keeps track of which type the constructors being abstracted in the pattern belong to.

Example 2. *Let's suppose we want to fuse $rm\ f = rmostR \circ mapR\ f$ where function mapR applies a function f to all the elements in a tree of type Rose a:*

$$\begin{array}{l} mapR\ f = [\!(\psi'_1,\psi'_2)\!]_F \\ \quad \textbf{where}\ F = \langle \overline{a} \times \Pi_2, \underline{\mathbf{1}} + \Pi_1 \times \Pi_2 \rangle \\ \qquad\quad \psi'_1\ (Rose\ a\ xs) = (f\ a, xs) \\ \qquad\quad \psi'_2\ [\,] \qquad\quad = (1,()) \\ \qquad\quad \psi'_2\ (x:xs) = (2,(x,xs)) \end{array}$$

If we derive derive σ from the coalgebra of rmostR, we obtain:

$$
\begin{array}{l}
(\mathit{rmostR}, \mathit{rmostL}) = [\![(id \triangledown id, id \triangledown id), \sigma(out_F)]\!]_{\langle G_1, G_2 \rangle} \\
\quad \textbf{where} \\
\qquad \sigma(\beta) = (\sigma_1(\beta), \sigma_2(\beta)) \\
\qquad \sigma_1 :: \forall a.\ \forall b.\ ((a,b) \to F\ (a,b)) \to ((a,b) \to G_1\ (a,b)) \\
\qquad \sigma_1(\beta_1, \beta_2) = \lambda v \to \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad \beta_1 \cdot (a, \beta_2 \cdot (1, ())) \to (1, a) \\
\qquad\qquad \beta_1 \cdot (a, xs) \quad\quad\;\; \to (2, xs) \\
\qquad \sigma_2 :: \forall a.\ \forall b.\ ((a,b) \to F\ (a,b)) \to ((a,b) \to G_2\ (a,b)) \\
\qquad \sigma_2(\beta_1, \beta_2)\ v = \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad \beta_2 \cdot (2, (x, \beta_2 \cdot (1, ()))) \to (1, x) \\
\qquad\qquad \beta_2 \cdot (2, (x, xs)) \quad\quad\;\; \to (2, xs)
\end{array}
$$

Applying the HYLO-UNFOLD *law for mutual hylomorphisms and inlining yields:*

$$
\mathit{rm}\ f = [\![(id \triangledown id, id \triangledown id), (\sigma_1(\psi_1', \psi_2'), \sigma_2(\psi_1', \psi_2'))]\!]_{\langle G_1, G_2 \rangle}
$$

$$
\begin{array}{ll}
\mathit{rm}\ f\ (\mathit{Rose}\ a\ []) = f\ a & \mathit{rmL}\ (x : []) \ = \mathit{rm}\ f\ x \\
\mathit{rm}\ f\ (\mathit{Rose}\ a\ xs) = \mathit{rmL}\ f\ xs & \mathit{rmL}\ (x : xs) = \mathit{rmL}\ f\ xs
\end{array}
$$

□

7. RELATED WORK

In this paper we have worked with fusion of hylomorphisms. There are other approaches to program fusion of which we discuss the most related here.

Firstly, we have the work of Onoue et al. [20], who implemented the HYLO system, and Schwartz [24], who reimplemented part of it in the pH [23] compiler. We are deeply indebted to these developments, which helped sorting out the workings of fusion systems based on hylomorphisms.

Shortcut fusion [11, 12, 26] is another approach that is based on a more general statement of the *acid rain* laws. As it is not constrained to work with hylomorphisms, it has the potential to fuse a broader class of functions. The most widespread used implementation is in the GHC compiler, which is capable of fusing compositions if the programmer takes the effort to write transformation rules for the involved functions. Rules for definitions in standard libraries are predefined. The ability to handle *automatically* programmer-supplied definitions has been one of the key motivations of the hylomorphism approach. Nonetheless, there exists work on shortcut fusion to overcome this limitation [3, 17, 19, 29]. In fact, the potential of these approaches should be greater than ours since the hylomorphism fusion laws we employ are a special case of the *acid rain* laws on which these systems are based:[5]

$$
\begin{array}{l}
(\!|\phi|\!)_F \text{ is strict} \\
f :: \forall b.(F\ b \to b) \to a \to b
\end{array}
\Rightarrow (\!|\phi|\!)_F \circ f\ in_F = f\ \phi
$$

$$
g :: \forall b.(b \to F\ b) \to b \to a \quad \Rightarrow \quad g\ out_F \circ [\!(\psi)\!]_F = g\ \psi
$$

Hylomorphisms, however, have offered a useful ground in which to solve a problem common to all approaches, which is automating derivation of folds and unfolds suitable for the different kinds of recursion (mutual, primitive, over multiple arguments, with regular functors), and there is a chance that the algorithms for deriving transformers could be generalized to derive functions f and g in the laws above.

Another relevant approach is stream fusion [4]. Stream fusion represents recursive functions as stream processors which can be composed and merged into a single recursive function thus achieving deforestation. In comparison to the approach discussed in this article, stream fusion requires functions to be written explicitly as stream processors, and it is not clear yet how such a task should be automated. Moreover, stream fusion has so far been proposed to deforest lists, deforesting other data structures would require further study. Regarding fusion of list functions, stream fusion is similar to what the hylomorphism approach could achieve with all of its extensions. For instance, stream fusion can fuse rather directly $\mathit{zip} \circ \mathit{filter}\ p$ or $\mathit{last} \circ \mathit{filter}\ p$. HFusion is also capable of fusing such compositions, but an additional trick is required in order to treat *filter p* as an unfold [5].

[5]The laws as expressed here are only valid for programs which do not use the *seq* operator[18].

8. FUTURE WORK AND CONCLUSIONS

We have shown a reformulation of the algorithm for abstracting constructors in coalgebras, being view patterns the key for it. The new presentation simplifies the description of the algorithms and stands out the duality of abstracting constructors in algebras and coalgebras. By presenting our simple extensions in Sections 5 and 6, we expect to have made a strong case that our reformulation is not merely aesthetic but one of practical benefits.

All the algorithms we have presented are implemented in our experimental tool HFusion, which also implements some other extensions like fusion of primitive recursive functions, hylomorphisms with non-polynomial functors, and many manipulations of recursive definitions to squeeze the most from the fusion laws [5, 6]. At the moment, HFusion does not find compositions automatically, which prevents us from making large scale performance tests to evaluate our extensions; we are working on it. Some preliminary benchmark results for HFusion can be found in the web page:
`http://www.fing.edu.uy/inco/proyectos/fusion/benchm.html`

There are many directions in which one may attempt to extend a fusion system. Possible extensions may include handling functions returning multiple results (like *unzip*) by dualizing our law and algorithms for fusing recursive functions over multiple arguments. An extension for fusing monadic programs [8, 21, 22] could allow our approach to optimize, for example, monadic parsers in conjunction with the extensions for mutual recursion. Also, it could be possible to integrate other transformations amenable to the hylomorphism representation, like the case of tupling [15].

As our contributions offer a new perspective on how constructors are abstracted out of patterns, another line of research would be porting this as well as other insights we have gathered while working with hylomorphisms to the context of shortcut fusion.

9. ACKNOWLEDGMENTS

We wish to thank the anonymous reviewers for their valuable and helpful comments on an earlier version of this paper.

10. REFERENCES

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

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

[3] O. Chitil. *Type-Inference Based Deforestation of Functional Programs.* PhD thesis, RWTH Aachen, October 2000.

[4] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *Int. Conf. on Func. Programming*, pages 315–326. ACM, 2007.

[5] F. Domínguez. HFusion: a fusion tool based on Acid Rain plus extensions. Master's thesis, PEDECIBA Informática, Universidad de la República, Uruguay, 2009.

[6] F. Domínguez and A. Pardo. Program fusion with paramorphisms. In *Mathematically Structured Func. Programming, Electronic Workshops in Computing.* BCS, 2006.

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

[8] N. Ghani and P. Johann. Short Cut Fusion of Recursive Programs with Computational Effects. In *Trends in Func. Programming*, volume 9, pages 113–128, 2009.

[9] J. Gibbons. Calculating Functional Programs. In *Alg. and Coalgebraic Methods in the Mathematics of Program Construction*, LNCS vol. 2297. Springer, 2002.

[10] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *3rd. ACM Int. Conf. on Func. Programming*, pages 273–279. ACM Press, 1998.

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

[12] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Funct. Prog. Lang. and Comp. Arch.*, pages 223–232. ACM Press, 1993.

[13] HFusion. `http://www.fing.edu.uy/inco/proyectos/fusion`.

[14] Z. Hu, H. Iwasaki, and M. Takeichi. An Extension of the Acid Rain Theorem. In *Fuji Int. Workshop on Func. and Logic Programming*, pages 91–105. World Scientific, 1996.

[15] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *Int. Conf. on Func. Programming*, pages 164–175. ACM Press, 1997.

[16] H. Iwasaki, Z. Hu, and M. Takeichi. Towards manipulation of mutually recursive functions. In *Fuji Int. Symp. on Func. and Logic Programming*, pages 61–79, 1998.

[17] P. Johann and E. Visser. Warm fusion in Stratego: A case study in generation of program transformation systems. *Annals of Mathematics and A.I.*, 29(1-4):1–34, 2000.

[18] P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *31st Symp. on Principles of Prog. Lang*, pages 99–110. ACM, 2004.

[19] L. Németh. *Catamorphism-Based Program Transformations for Non-Strict Functional Languages.* PhD thesis, Glasgow University, 2000.

[20] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *IFIP TC 2 WG 2.1 Int. Workshop on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, Ltd., 1997.

[21] A. Pardo. Fusion of recursive programs with computational effects. *Theoretical Compupter Science*, 260(1-2):165–207, 2001.

[22] A. Pardo. Combining datatypes and effects. In *Advanced Functional Programming*, volume 3622 (2005) of *LNCS*, pages 171–209. Springer, 2005.

[23] pH compiler. `http://csg.csail.mit.edu/projects/languages/ph.shtml`.

[24] J. Schwartz. Eliminating Intermediate Lists in pH. Master's thesis, MIT, 2000.

[25] J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *7th ACM Int. Conf. on Func. Programming*, pages 124–132. ACM, October 2002.

[26] A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Funct. Prog. Lang. and Comp. Arch.*, pages 306–313. ACM Press, 1995.

[27] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14th Conf. on Principles of Prog. Lang*, pages 307–313. ACM Press, 1987.

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

[29] T. Yokoyama, Z. Hu, and M. Takeichi. Calculation rules for warming-up in fusion transformation. In *Trends in Func. Programming*, pages 399–41, 2005.