Exploiting algebra/coalgebra duality for program fusion extensions

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

Alberto Pardo
Instituto 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

\[ f \ p \ n = \text{sum} (\text{filter} \ p \ [1 \ldots n]) \]

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 imple-
mentation H Fusion 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 func-
tion \( f :: a \to b \) by splitting its definition into three compo-
nents, written as \([\phi, \psi]_F\), such that \( \psi \) describes how
the arguments to the recursive calls of \( f \) are computed from
the input value, \( \phi \) describes how the results of the recur-
sive 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 associ-
ated with hylomorphisms which constitute the core of the
approach. Throughout we will assume a cpo semantics in
terms of pointed cps.

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

Sometimes it is necessary to consider functors on two vari-
able: A bifunctor \( F \) is a binary operator that applies to
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 \( F (f \circ (g, h \circ k) = F \circ F \).

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

Functors will be specified in compact notation as com-
positions of a set of elemental functors. The identity functor
is given by \( I \colon a = a \) and \( f = f \), whereas for any type \( c \)
the constant functor \( i \) is such that \( a = c \) and \( i \) is \( id \).
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) = \((f, g)(a, b)\). In
our meta-language we introduce a sum type constructor
\( a + b \) which builds the disjoint sum of two types. Semi-
technically, this means that \( a + b = \{(1) \times a \} \cup \{(2) \times b \} \cup \{(1, 2) \} \). It
can be treated as a bifunctor by defining \( f + g :: a + b \to c + d \)
with 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 as a case analysis operator \( f \lor g :: a + b \to c \) given
by the strict function such that \( (f \lor g)(1, a) = f a \) and
\( (f \lor g)(2, b) = g b \).

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 \( ! \) we denote the unit type.
We will also use products and sums of functors, that is,
\( F \times G \) \( a = F \times F \times G \times a \) and \( (F \times G) f \) \( = F \times f \times G \times f \), for
\( \times \in \{ +, \} \).

An example of a functor expressed in compact form is
\( F = \mathbb{T} + \mathbb{M} \times I \). We call recursive positions of a functor
those positions where functor \( I \) occurs.

---

1 Most notably, H Fusion does not handle programs using the
\textit{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 usu-
ally written \( \mu F \). The isomorphism is provided by two strict
functions, called \( in_F :: \mu F \to \mu F \) and \( out_F :: \mu F \to \mu F \),
that are 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
parametric for the sake of simplicity in notation.

---

Functors.

The following fusion laws are instantiations of the so-
called fusion 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 coalge-
bras, respectively [7].

\[
\begin{align*}
\phi F \circ \psi F &= \mu \phi, \psi F \\
\phi F &\text{ is strict } \Rightarrow \phi F \circ [\tau(\psi F), \psi G] = \tau(\phi F, \psi G) \\
[\phi, \sigma(\psi G)] F \circ [\psi F] G &= \phi F, \psi G
\end{align*}
\]

2Formally, 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.

\[
\begin{align*}
\text{map } f &\colon (a \to b) \to [a] \to [b] \\
\text{map } f &\colon [] \to [] \\
\text{map } f (x : xs) &= f x : \text{map } f xs \\
\text{intersp } e &\colon [a] \to [a] \\
\text{intersp } e (x : []) &= x : [] \\
\text{intersp } e (x : xs) &= x : e : \text{intersp } e xs
\end{align*}
\]

If we want to fuse \(\text{map } f \circ \text{intersp } e\), we first derive hylomorphisms with functors \(F = 1 + \pi \times I\) and \(G = 1 + \pi \times I\).

\[
\text{map } f = \{\gamma \}_{F} \, \phi, \sigma\] for the coalgebra of \(\tau\). This is

\[
\text{intersp } e = \lbrack \tau (\text{in}_F) \rbrack, \psi \rbrack_G
\]

where \(\tau \colon \forall a. (F a \to a) \to (G a \to a)\)

\[
\begin{align*}
\tau (a) &= (a) \forall \tau_2 (a) \forall \tau_3 (a) \\
\tau_1 (a_1 \forall a_2) (a) &= a_1 \\
\tau_2 (a_1 \forall a_2) x &= a_2 (x, a_1) \\
\tau_3 (a_1 \forall a_2) (a, v) &= a_2 (x, a_2 (e, v))
\end{align*}
\]

Applying Acid Rain for the Fold-Hylo case we get \(\text{mi } f p = \lbrack \tau (\gamma) \rbrack, \psi \rbrack_G\), which converted back to a Haskell recursive definition is:

\[
\begin{align*}
\text{mi } f e &\colon [a] \to [a] \\
\text{mi } f e (x : []) &= f x : [] \\
\text{mi } f e (x : xs) &= f x : e : \text{mi } f e xs
\end{align*}
\]

As another example we could consider the composition

\[
\text{im } e f = \text{intersp } e \circ \text{map } f
\]

where we would need to write \(\text{map } f\) as an unfold and rewrite the coalgebra of \(\text{intersp } e\) using a \(\tau\) transformer:

\[
\begin{align*}
\text{map } f &= \phi, \sigma\] for the algebra of \(\phi\).
\]

\[
\begin{align*}
\text{intersp } e &= \lbrack \phi, \sigma (\text{out}_F) \rbrack_G \\
\text{where } &\sigma : \forall a. (a \to F a) \to (a \to G a) \\
\sigma (\beta) v &= \text{case } \beta \text{ of } \begin{cases} 
(1, ()) &\to (1, ()) \\
(2, (x, xs)) &\to \text{case } \beta (x, xs) \text{ of } 
(1, ()) &\to (2, x) \\
- &\to (3, (x, xs)) \end{cases}
\end{align*}
\]

The result of applying Acid Rain for the Hylo-Unfold case is \(\lbrack \phi, \sigma (\psi') \rbrack_G\), which can be written in Haskell as:

\[
\begin{align*}
\text{im } e f [] &= [] \\
\text{im } e f (x : []) &= f x : [] \\
\text{im } e f (x : xs) &= f x : e : \text{im } e f xs
\end{align*}
\]

Both examples involve a step where transformers \(\tau\) or \(\sigma\) 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 \(\tau\), constructors are abstracted on the expressions of an algebra. In the case of the derivation of \(\sigma\), 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 \(\sigma\) for the coalgebra of \(\text{intersp } e\) we had to rewrite the coalgebra into a cascade of cases while no dual task was needed to derive \(\tau\).

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 \(\tau\) and \(\sigma\), 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 \(\tau\)

The derivation algorithm for \(\tau\) is in essence the same given by Ong 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 \(\sigma\) transformers.

Given a \(G\)-algebra \(\phi\) the goal of the algorithm is to determine whether it can be written as \(\tau (\text{in}_F)\), for \(\tau : \forall a. (F a \to a) \to (G a \to a)\). The algorithm requires that \(\phi\) be given by a case analysis \(\phi_1 \cdots \phi_n\) such that each \(\phi_i\) is a function \(\lambda(v_1, \ldots, v_n) \to t_i\) where the term \(t_i\) satisfies the following normal form:

1. it is a recursive variable; or
2. it is a constructor application \(C_j (t'_1, \ldots, t'_n)\) where each \(t'_k\) in a recursive position of constructor \(C_j\) is in

\[\text{http://www.haskell.org/ghc/}.\]
normal form. By recursive positions of a constructor we mean those positions where the corresponding datatype \( \mu F \) occurs recursively; they correspond to the occurrences of the \( F \) functor in the expression of function \( F \).

Those \( t'_{\lambda} \) that are not in recursive positions of \( C \) 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 \( \phi \), they sometimes can be moved out of the algebra by restructuring the hylomorphism in order to obtain this normal form.

The derivation algorithm for \( \tau \) is shown in Figure 1. The objective of algorithm \( A \) is to abstract constructors, substituting them for the corresponding operations of the \( F \)-algebra \( \alpha = \alpha_1 \ldots \ldots \ldots \alpha_n \). 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 \( T(F, \phi) (in_F) = \phi \).

**Derivation of \( \sigma \)**

The derivation algorithm for \( \sigma \) takes as input a \( G \)-coalgebra to be rewritten as \( \sigma(out_F) \) where \( \sigma : \forall a.(a \rightarrow F \ a) \rightarrow (a \rightarrow G \ a) \). The input coalgebra must be in the form: \( \lambda v \rightarrow \text{case } v \text{ of } \{p_1 \rightarrow t_1; \ldots; p_m \rightarrow t_m\} \), 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 \( C_i \ (p'_1, \ldots, p'_n) \) and pattern \( p'_i \) appearing in a recursive position of \( C \) is in normal form. A pattern \( p'_i \) in a non-recursive position can have any shape as far as it does not reference variables appearing in recursive terms.

  A pattern \( p'_i \) is said to appear in a recursive position if the functor \( F \) (not \( G \)) tells so, being \( F \) the functor characterizing the input datatype \( \mu F \) of the coalgebra.

In Figure 2 we present our derivation algorithm for \( \sigma \). Algorithm \( B \), which abstracts constructors inside patterns, here plays the same role as algorithm \( A \) in the derivation algorithm for \( \tau \). 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:

**Figure 1: Derivation algorithm for \( \tau \)**

**Figure 2: Derivation algorithm for \( \sigma \)**

\[
T(F, \phi :: \mu F \rightarrow \mu F) :: (F \ a \rightarrow a) \rightarrow (G \ a \rightarrow a) = \\
T(F_1 + \ldots + F_m, \phi_1 \vdots \cdot \cdot \cdot \vdots \phi_n) = \\
\lambda(\alpha_1 \vdots \cdot \cdot \cdot \vdots \alpha_n) \rightarrow T'(\phi_1) \vdots \cdot \cdot \cdot \vdots T'(\phi_n)
\]

where \( T'(\lambda vs \rightarrow t) = \lambda vs \rightarrow A(t) \)

- \( A(v) = v \) if \( v \) is a variable recursive
- \( A(C_1 \ (t_1, \ldots, t_k)) = \alpha \ ((F_1 \ A) \ (t_1, \ldots, t_k)) \)
- \( A(t) = \{\alpha_1 \vdots \cdot \cdot \cdot \vdots \alpha_n\} \rightarrow t \) every other case

\[
S(F, \psi :: \mu F \rightarrow G \ \mu F) :: \forall a.(a \rightarrow F \ a) \rightarrow (a \rightarrow G \ a) = \\
S(F, A \rightarrow \text{case } v \text{ of } \{p_1 \rightarrow t_1; \ldots; p_m \rightarrow t_m\}) = \\
\lambda \beta : \lambda v \rightarrow \text{case } v \text{ of } \{B(p_1) \rightarrow t_1; \ldots; B(p_m) \rightarrow t_m\}
\]

where \( F_1 + \ldots + F_m = F \)

- \( B(v) = v \) if \( v \) is a variable recursive
- \( B(C_j \ (p_1, \ldots, p_k)) = \beta \ ((j \ (F_j \ B) \ (p_1, \ldots, p_k)) \)
- \( B(p) = \{\beta \}_{F \rightarrow p} \) every other case

\[
data \ Tree = Node \ Tree \ Tree \ | \ Empty
\]

\[
\psi (Node \ (Node \ I \ r) \ _ \ Empty) = (1, (I, r))
\]

Calling \( \psi (Node \ (Node \ Empty \ _ \ _) \ Empty) \) 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 \( zip :: [a] \times [b] \rightarrow [a \times b] \) :

\[
\begin{align*}
zip & : (x : xs, y : ys) \rightarrow (x, y) : zip (xs, ys) \\
zip & : (\ _ \ _) \rightarrow []
\end{align*}
\]

It can be written as the unfold \( [\psi]_G \) where:

\[
G = I + \frac{a \times b \times I}{a \times b} \rightarrow G \ ([a] \times [b])
\]

\[
\psi : (x : xs, y : ys) \rightarrow (2, ((x, y), (xs, ys)))
\]

\[
\psi : (\ _ \ _ \ _ \ _ ) \rightarrow (1, (\ _ ))
\]

Whereas the fusion laws expect a single intermediate data structure to eliminate, in a composition like \( zip \circ (map f \times id) \) 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 \( [\phi] :: [a] \rightarrow ([b] \rightarrow [a \times b]) \) that recurses on the first list and returns a function as result, and we then fuse \( [\phi] \circ map f \). Similarly, we could fuse \( zip \circ (id \times map f) \) 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

\[\text{For the sake of presentation, functions that recurse over multiple arguments will be written in uncurried form for compatibility with their representation as hylomorphisms.}\]
proposed for the shortcut fusion approach [25]. We maintain the
hylo morphism 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.** \( \sigma : \forall a. (a \to F a) \to (H a \to G (H a)) \)
\[ [\sigma, \sigma(out_F)]_G \circ H \psi \] \[ [\sigma, \sigma(out_F)]_G \]

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 \( \text{zipmap } f = \text{zip } \circ (\text{map } f \times \text{id} ) \). Function
map can be written as an unfold \( [\psi']_F \) like we showed in
Section 3. By rewriting the hylo morphism for zip shown above in
terms of a transformer \( \sigma \) we obtain \( [\sigma(out_F)]_G \),
where:

\[ \sigma : \forall z. (z \to F z) \to (z \times [b] \to G (z \times [b])) \]
\[ \sigma(\beta) (x, y) = \text{case } (x, y) \text{ of } \]
\[ \beta((z, x, y), y : y's) \to (2, (x, y), (x, y))) \]

Applying Lemma 1 with \( H = I \times \overline{b} \), we obtain \( \text{zipmap } f = \)
\[ [\sigma(out_F)]_G \). Inlining,
\[ \text{zipmap} :: (c \to a) \to [c] \times [b] \to [a \times b] \]
\[ \text{zipmap } f (x : x, y : y's) = (f \ y : \ y : \text{zipmap } f (x, y)) \]
\[ \text{zipmap } f (\_ \_ ) = [\_ ] \]

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
particularly 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 \( \text{zip } \circ (\text{map } f \times \text{id} ) \), we can simply proceed in multiple
steps by splitting the product: \( \text{zip } \circ (\text{map } f \times \text{id} ) \circ (\text{id} \times \text{map } g) \). Then, in our case, we first fuse \( \text{zip } \circ (\text{map } f \times \text{id} ) \) as
in Example 1, and then we fuse the result with \( (\text{id} \times \text{map } g) \).

Functions that use accumulators or recurse over parameters
of non-recursive types can also be represented as hylo-morphisms
over multiple arguments. Examples of them are \( \text{take, drop, and foldl} [1] \). Let us consider the case of foldl:

\[ \text{foldl} :: (k : a \to b) \to b \times [a] \to b \]
\[ \text{foldl } f (\_ \_ ) = e \]
\[ \text{foldl } f (x, x : xs) = \text{foldl } f (f x, xs) \]

It can be written as the hylo morphism \( \text{foldl } f = \big[ \text{id}, \sigma(out_F) \big]_G \),
where:

\[ \sigma : \forall z. (z \to F z) \to (b \times z \to G (b \times z)) \]
\[ \sigma(\beta) (e, l) = \text{case } l \text{ of } \beta(1, [\_ ] \_ ) \to (1, e) \]
\[ \beta(2, (x, z, xs)) \to (2, (f x, xs)) \]

for \( G = \overline{b} + I \) and \( F = \overline{F} + \overline{a} \times I \), such that it can be fused with
another hylo morphism on the list argument. For instance, if we
fuse the composition \( f m \_ g = \text{foldl } f \circ (\text{id} \times \text{map } g) \), we obtain
\( f m \_ g = \big[ \text{id}, \sigma(out_F) \big]_G \), being \( \psi \) the same coalgebra
for map described in Section 3. Inlining,
\[ f m \_ g (f x, x : xs) = \text{foldl } f (g f (e \_ g x), xs) \]

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 \( \text{foldl} \), 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 \( \sigma \)

For the sake of clarity we only discuss how to derive a trans-
former \( \sigma \) which enables fusion on the first argument of a de-
finition 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:
\[ \psi : a \times b \to G (a \times b) \]
\[ \psi = \lambda \cdot \text{case } \psi \text{ of } \]
\[ (p_{11}, p_{12}) \to (1, (t_{11} \ldots, t_{1k_1})) \]
\[ \ldots \]
\[ (p_{n1}, p_{n2}) \to (m, (t_{n1} \ldots, t_{nk_n})) \]

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 \( p_{11} \), and \( t \) must not reference
any such variable. The patterns \( p_{11} \) must conform to the
same normal form that we required for patterns in our
original derivation algorithm for \( \sigma \). The coalgebras for
\( \text{zip} \) and \( \text{foldl} \) satisfy these restrictions.

The extended algorithm is the following:

\[ S' (F, \lambda \cdot \text{case } \psi \text{ of } \]
\[ \{(p_{11}, p_{12}) \to t_1 \ldots (p_{n1}, p_{n2}) \to t_n \}) = \lambda \cdot \text{case } \psi \text{ of } \]
\[ \{(B(p_{11}, p_{12}) \to t_1 \ldots (B(p_{n1}, p_{n2}) \to t_n \}) \]

being \( B \) the same algorithm presented in Section 4. Alg.
\( S' \) 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 \( \text{rmostR} \) which extracts
the rightmost leave of a finitary tree.

\[ \text{data} \ Rose a = \text{Rose a} \ [\text{Rose a}] \]
\[ \text{rmostR} :: \text{Rose a} \to a \]
\[ \text{rmostR} (\text{Rose a}[\_ ]\_ ) = a \]
\[ \text{rmostR} (\text{Rose a} \_ x) = \text{rmostL} \_ x \]
\[ \text{rmostL} :: \text{Rose a} \to a \]
\[ \text{rmostL} (\_ x) = \text{rmostR} \_ x \]
\[ \text{rmostL} (\_ \_ x) = \text{rmostL} \_ x \]

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 \( (F, G) \) of two bifunctors \( F \) and \( G \) such that
\( (F, G) (a, b) = (F (a, b), G (a, b)) \). 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
hylo morphisms 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 \( \psi : (a, b) 
H (a, b) \) be an
\( H \)-coalgebra. A mutual hylo morphism \( [\phi, \psi]_H \) 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 \( \text{rmostR} \) and \( \text{rmostL} \) as a mutual hylo-
morphism:

\[ \text{rmostR}, \text{rmostL} = (\text{rmostR}, \text{rmostL}) \big[ (\text{id} \times \text{id}), (\text{id} \times \text{id}) \big] (\text{G}_1, \text{G}_2) \]
\[ \text{where} \ G_1 = \overline{\text{F}} + \overline{\text{G}} \]
\[ G_2 = \Pi_1 + \Pi_2 \]
\[ \psi_1 :: \text{Rose a} \to G_1 (\text{Rose a}, [\text{Rose a}]) \]
\[ \psi_2 :: \text{Rose a} \to G_2 (\text{Rose a}, [\text{Rose a}]) \]
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 \( \text{rmostR} \) without mutual recursion:

\[
\text{rmostR} (\text{Rose } a \ [\ ] ) = a
\]

\[
\text{rmostR} (\text{Rose } a \ [x]) = \text{last} (\text{map} \ \text{rmostR} \ [x])
\]

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 \( \text{last} \circ \text{map} \ \text{rmostR} \) and then specialize the result \( \text{last} \circ \text{map} \ \text{rmostR} \) for the non-recursive parameter \( \text{rmostR} \).

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:

\[
\text{sumR} :: \text{Rose} \ \text{Int} \to \text{Int} \\
\text{sumR} (\text{Rose } a \ [x]) = a + \text{sum} (\text{map} \ \text{sumR} \ [x])
\]

\[
\text{sum} (\ [\ ] ) = 0
\]

We could express this function with mutual recursion by first fusing \( \text{sum} \circ \text{map} \ \text{sumR} \).

\[
\text{sumR} (\text{Rose } a \ [x]) = a + \text{sm} \ \text{sumR} \ [x]
\]

and then specializing \( \text{sm} \ \text{mapR} \) for the argument \( \text{mapR} \) obtaining

\[
\text{sumR} (\text{Rose } a \ [x]) = a + \text{sm} \text{sumR} \ [x]
\]

\[
\text{sm} (\ [\ ] ) = 0
\]

\[
\text{sm} (\ [x]) = \text{sumR} \ [x] + \text{sm} \text{sumR} \ [x]
\]

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 \( \text{intersp} \), shown in Section 3. The function \( \text{intersp} \) could be written either as a mutual fold

\[
\text{intersp} e = (\langle \phi_1, \phi_2 \rangle)F
\]

where

\[
F = (T + \bar{\pi} \times \text{Pi}2, \bar{T} + \bar{\pi} \times \Pi2)
\]

\[
\phi_1 (1, ()) = [ ]
\]

\[
\phi_1 (2, (x : vs)) = e : x : vs
\]

\[
\phi_2 (1, ()) = [ ]
\]

\[
\phi_2 (2, (x : vs)) = e : x : vs
\]

which mimics the following definition of \( \text{intersp} \):

\[
\text{intersp} :: a \to [a] \\
\text{intersp} e [ ] = [ ]
\]

\[
\text{intersp} e (x : vs) = x : \text{intersp} e vs
\]

or it could be written as a mutual unfold

\[
\text{intersp} e = (\langle \psi_1, \psi_2 \rangle)F
\]

where

\[
F = (T + \bar{\pi} \times \text{Pi}2, \bar{T} + \bar{\pi} \times \Pi1)
\]

\[
\psi_1 (1, ()) = (1, ())
\]

\[
\psi_1 (x : xs) = (2, (x, xs))
\]

\[
\psi_2 (1, ()) = (1, ())
\]

\[
\psi_2 vs = (2, (e, vs))
\]

which mimics this other definition:

\[
\text{intersp} :: a \to [a] \\
\text{intersp} e [ ] = [ ]
\]

\[
\text{intersp} e (x : vs) = x : \text{intersp} e vs
\]

Derivation of \( \tau \)

Having a mutual hylomorphism \( \langle \phi_1, \phi_2 \rangle \), we might want to derive an equivalent one of the form \( \langle \tau_1, \tau_2 \rangle \) when composed with a mutual fold \( \langle \psi_1, \psi_2 \rangle F \). Indeed, we will be deriving a transformer \( \tau \) that is also given as a pair: \( \tau (a) = (\tau_1 (a), \tau_2 (a)) \). This is how we obtain \( \tau \):

\[
\tau :: a \to [a] \\
\tau (a) = (\text{cases} a) (\langle \psi_1, \psi_2 \rangle F, \langle \phi_1, \phi_2 \rangle)
\]

where \( T \) 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 \( \text{Rose} \) we may find the application of constructors of type \( \text{Rose} a \).

Derivation of \( \sigma \)

Having a mutual hylomorphism \( \langle \phi, \psi_1, \psi_2 \rangle \), we might want to derive an equivalent one of the form \( \langle \sigma, \sigma (\text{out} F) \rangle \) when composed with a mutual unfold \( \langle \psi \rangle C \). The transformer \( \sigma \) is calculated as follows:

\[
\sigma :: a \to [a] \\
\sigma (a) = (\text{cases} a) (\langle \psi_1, \psi_2 \rangle F, \langle \phi_1, \phi_2 \rangle)
\]

where \( S \) is the algorithm presented in Figure 4. Once more, algorithm \( S \) is the dual of the derivation algorithm for \( \tau \). Through its sub-index, \( B_0 \) keeps track of which type the constructors being abstracted in the pattern belong to.

Example 2. Let’s suppose we want to fuse \( r m f = \text{rmostR} \circ \text{mapR} \) where function \( \text{mapR} \) applies a function \( f \) to all the elements in a tree of type \( \text{Rose} a \):

\[
\text{mapR} f = (\langle \psi_1, \psi_2 \rangle)F
\]

where

\[
F = (T + \bar{\pi} \times \Pi1, \bar{T} + \bar{\pi} \times \Pi2)
\]

\[
\psi_1 (\text{Rose } a \ [x]) = (f a, xs)
\]

\[
\psi_2 (1, ()) = (1, ())
\]

\[
\psi_2 (x : xs) = (2, (x, xs))
\]
If we derive derive \( \sigma \) from the coalgebra of \( \text{rmostR} \), we obtain:

\[
(\text{rmostR, rmostL}) = [([id \triangleright id, id \vdash id], \sigma(\text{out}_F))]_{(G_1, G_2)}
\]

where

\[
\begin{align*}
\sigma(\beta) &= (\sigma_1(\beta), \sigma_2(\beta)) \\
\sigma_1 :& \forall a. \forall b. ((a, b) \rightarrow F(a, b)) \rightarrow ((a, b) \rightarrow G_1(a, b)) \\
\sigma_1(\beta_1, \beta_2) &= \lambda v \rightarrow \text{case } v \text{ of } \\
& \quad \beta_1(a, \beta_2((1, 1))) \rightarrow (1, a) \\
\beta_1(a, zs) & \rightarrow (2, zs) \\
\sigma_2 &: \forall a. \forall b. ((a, b) \rightarrow F(a, b)) \rightarrow ((a, b) \rightarrow G_2(a, b)) \\
\sigma_2(\beta_1, \beta_2) v &= \text{case } v \text{ of } \\
& \quad \beta_2(x, (2, 1, (1))) \rightarrow (1, x) \\
& \quad \beta_2(x, zs) \rightarrow (2, zs)
\end{align*}
\]

Applying the hylo-unfold law for mutual hylomorphisms and inlining yields:

\[
\begin{align*}
\text{rm} f &= [([id \triangleright id, id \vdash id], (\sigma_1(\psi_1', \psi_2'), \sigma_2(\phi_1', \phi_2')))_{(G_1, G_2)}] \\
\text{rm} f (\text{Rose } a []) &= f a \\
\text{rm} f (\text{Rose } a zs) &= \text{rmL } f x \\
\text{rm} f (\text{Rose } a zs) &= \text{rmL } f x \\
\text{rm} f (\text{Rose } a zs) &= \text{rmL } f x \\
\text{rm} f (\text{Rose } a zs) &= \text{rmL } f x
\end{align*}
\]

\[\square\]

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 pre-defined. 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:

\[
\begin{align*}
\phi_f & \text{ is strict} \\
f :: \forall b. (F b \rightarrow b) \rightarrow a \rightarrow b & \Rightarrow (\phi_f \circ f \text{ in}_F = f \phi) \\
g :: \forall b. (b \rightarrow F b) \rightarrow b \rightarrow a & \Rightarrow g \text{ out}_F \circ (\psi) = g \psi
\end{align*}
\]

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 \( \text{zip} \circ \text{filter} \) or \( \text{last} \circ \text{filter} \). HFusion is also capable of fusing such compositions, but an additional trick is required in order to treat \( \text{filter} \) as an unfold [5].

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.ting.edu.vn/inco/projects/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 \( \text{unzip} \)) 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


