Deriving Structural Hylomorphisms From Recursive Definitions

Zhenjiang Hu [*] Hideya Iwasaki [†] Masato Takeichi [‡]

Summary.

In functional programming, small programs are often "glued" together to construct a complex program. Program fusion is an optimizing process whereby these small programs are fused into a single one and intermediate data structures are removed. Recent work has made it clear that the process is especially successful if the recursive definitions are expressed in terms of hylomorphisms. In this paper, we propose an algorithm which can automatically turn all practical recursive definitions into structural hylomorphisms making program fusion be easily applied.

1 Introduction

The compositional style of functional programming has many advantages of clarity and higher level of modularity. It constructs a complex program by gluing components which are relatively simple, easier to write, and potentially more reusable. However, some data structures, which are constructed in one component and consumed in another but never appear in the result of the whole program, give rise to the problem of efficiency.

Consider a toy example of function *all* which tests whether all the elements of a list satisfy the given predicate p. It may be defined as follows.

$$all\ p = and \circ map\ p$$
$$\text{where } and = \lambda xs.\,\text{case } xs \text{ of } Nil \to True;\ \ Cons\,(a, as) \to a \wedge (and\ as)$$

Here p is applied to all the elements of the list producing an intermediate list of Booleans which are then "anded" together by the function *and* producing a single Boolean result. To make the function *all* be computed efficiently, it is expected to fuse *and* and *map* p together to have the following new definition where the intermediate list of Booleans is not produced.

$$all\ p = \lambda xs.\,\text{case } xs \text{ of } Nil \to True;\ \ Cons\,(a, as) \to p\,a \wedge (all\ as)$$

[*] Department of Information Engineering, Graduate School of Engineering, The University of Tokyo (hu@ipl.t.u-tokyo.ac.jp) .

[†] Educational Computer Centre, The University of Tokyo (iwasaki@rds.ecc.u-tokyo.ac.jp) .

[‡] Department of Mathematical Engineering and Information Physics, Faculty of Engineering, The University of Tokyo (takeichi@u-tokyo.ac.jp) .

There are two kinds of approaches dealing with such fusion. One, first proposed by Wadler as called *deforestation*, aims to fuse *arbitrary* functions by *fold-unfold* transformations, keeping track of function calls and using clever control to avoid infinite unfolding[Chi92, Wad88]. The other, quite differently, makes use of some *specific* forms such as *catamorphisms* (or called *folds*), *anamorphisms* (or called *unfolds*) and *hylomorphisms* and finds how they interact[GLJ93, SF93, TM95].

The second approach has been proved to be more practical in a real implementation in compilers, although at first sight it seems less general than the former. Its theoretical basis can be found in the study of *Constructive Algorithmics*[Fok92, Mal90, MFP91] which will be outlined in Section 2. In constructive algorithmics, data types are categorically defined as initial fixed points of functors, and functions from one data type to another are represented as structure-preserving maps between algebras. By doing so, an orderly structure can be imposed on the program and such structure can be exploited to facilitate program fusion.

However, this approach imposes recursive structures of specific forms on programs, which is unrealistic in practical functional programming. One attempt has been made by Launchbury and Sheard[LS95]. They gave an algorithm to turn recursive definitions into so-called *build-cata* forms (i.e. catamorphisms with constructors being parameterized) so that the *shortcut deforestation* technique becomes applicable. One major problem still left is that the build-cata form is too restrictive to describe some kinds of practical recursive definitions and therefore many intermediate data structures cannot be removed.

The purpose of this paper is to demonstrate how practical recursive definitions can be automatically turned into hylomorphism, general forms covering all those in [MFP91, SF93, TM95], which makes program fusion transformation be applied better. The main contribution of this work is as follows.

- We propose an algorithm that can automatically turn almost *all* recursive definitions to structural hylomorphisms. With the use of fusion systems[SF93, KL94, TM95] successfully developed for hylomorphisms, we can improve a larger class of programs freely defined by programmers.
- Our algorithm is guaranteed to be correct, and to terminate with a successful hylomorphism. To the contrary, Launchbury and Sheard's derivation algorithm of build-catas from recursive definitions may fail to give build-catas[LS95]. Their algorithm has to be on the alert against failure of the "two-stage fusion" and gives up the derivation in case failure occurs.
- Our algorithm structures hylomorphisms so that the Hylo Fusion and Acid Rain Theorems (Section 2) can be effectively applied. Particularly, we propose a new theorem for deriving polymorphic functions (Section 5.2), namely τ and σ, for the use of Acid Rain Theorem.
- Our algorithm does not limit its use to the fusion of recursions inducting over a single data structure. It is helpful for the fusion of recursions over multiple data structures without introducing new fusion theorems as in [FSZ94] (Section 4.4). Moreover, it is also useful for other program optimizations

such as removal of multiple traversal over the same data structures[Tak87]. This is because general optimization rules are much easier to be defined over hylomorphisms rather than over the recursions.

This paper is organized as follows. In Section 2 we review the previous work in Constructive Algorithmics, the theoretical basis of hylomorphisms. Section 3 defines a simple language for the description of recursive definitions. We discuss our algorithm in Section 4 and 5. In Section 4 we define our algorithm for deriving hylomorphisms from recursive definitions, and in Section 5 we give the algorithm for structuring hylomorphisms so that the two fusion theorems can be effectively applied. Section 6 discusses the related work and Section 7 gives the conclusion.

2 Background

In this section, we review previous work on constructive algorithmics and explain some basic facts which provide theoretical basis of our method. Throughout this paper, our default category C is a $\mathcal{CPO}$, the category of complete partial orders with continuous functions.

2.1 Functors

Endofunctors on category C (functors from C to C) are used to capture both data structure and control structure in a type definition. In this paper, we assume that all the data types are defined by endofunctors which are only built up by the following four basic functors. Such endofunctors are known as *polynomial functors*.

Definition 1 (Identity) The identity functor I on type X and its operation on functions are defined as follows.

$$I\ X = X,\ I\ f = f$$

□

Definition 2 (Constant) The constant functor $!A$ on type X and its operation on functions are defined as follows.

$$!A\ X = A,\ !A\ f = id$$

where id stands for the identity function. □

Definition 3 (Product) The product $X \times Y$ of two types X and Y and its operation to functions are defined as follows.

$$\begin{array}{ll} X \times Y & = \{(x, y) \mid x \in X,\ y \in Y\} \\ (f \times g)\ (x, y) & = (f\ x,\ g\ y) \end{array}$$

Some related operators are:

$$\begin{array}{l} \pi_1\ (a, b) = a, \quad \pi_2\ (a, b) = b \\ (f\ \triangle\ g)\ a = (f\ a,\ g\ a). \end{array}$$

□

Definition 4 (Separated Sum) The separated sum $X + Y$ of two types X and Y and its operation to functions are defined as follows.

$$\begin{array}{ll} X + Y & = \{1\} \times X \cup \{2\} \times Y \\ (f + g)\ (1, x) & = (1, f\ x) \\ (f + g)\ (2, y) & = (2, g\ y) \end{array}$$

Some related operators are:

$$\begin{array}{l} \iota_1\ a = (1, a), \quad \iota_2\ b = (2, b) \\ (f \triangledown g)\ (1, x) = f\ x \\ (f \triangledown g)\ (2, y) = g\ y. \end{array}$$

□

Although the product and the separated sum are defined over 2 parameters, they can be naturally extended for n parameters. For example, the separated sum over n parameters can be defined by $+_{i=1}^{n} X_i = \cup_{i=1}^{n} \{\mathsf{i}\} \times X_i$ and $(+_{i=1}^{n} f_i)\,(\mathsf{j}, x) = (\mathsf{j},\ f_j\ x)$.

2.2 Data Types as Initial Fixed Points of Functors

A data type is a collection of operations (data constructors) denoting how each element of the data type can be constructed in a finite way, and via these data constructors functions on the type may be defined. So a data type is a particular algebra, one distinguished property is categorically known as initiality of the algebra. Let C be a category and F be an endofunctor from C to C.

Definition 5 (F-algebra) An *F-algebra* is a pair (X, ϕ), where X is an object in C, called the carrier of the algebra, and ϕ is a morphism from object $F\ X$ to object X denoted by $\phi :: F\ X \to X$, called the operation of the algebra. □

Definition 6 (F-homomorphism) Given F-algebras (X, ϕ) and (Y, ψ), the *F homomorphism* from (X, ϕ) to (Y, ψ) is a morphism h from object X to object Y in category C satisfying $h \circ \phi = \psi \circ F\ h$. □

Definition 7 (Category of F-algebras) The *category of F-algebras* has as its objects the F-algebras and has as its morphisms all F-homomorphisms between F-algebras. Composition in the category of F-algebra is taken from C, and so are the identities. □

It is known that an initial object in the category of F-algebras exists when F is a polynomial functors[Mal90]. The representative for the initial algebra is denoted by μF. Let $(T, in_F) = \mu F$, μF defines a data type T with the *data constructor* $in_F : F\ T \to T$. Function $out_F : T \to F\ T$ is the inverse of in_F and it destructs its argument and is therefore called *data destructor*. To be concrete, consider the data type of cons lists given by the following equation with elements of type A:

$$List\ A = Nil \mid Cons\ (A, List\ A).$$

It is categorically defined as the initial object of $(List\ a,\ Nil \triangledown Cons)$[1] in the category of L_A-algebras, where L_A is the endofunctor defined by $L_A\ = !\mathbf{1} + !A \times I$ ($\mathbf{1}$ is

[1] Strictly speaking, the *Nil* should be written as $\lambda().Nil$. In this paper, the function with the form of $\lambda().t$ will be simply denoted as t.

the terminal object in C). Here, the data constructor and the data destructor are as follows.

$$\begin{array}{l} in_{L_A} \;= Nil \triangledown Cons \\ out_{L_A} = \lambda xs.\ \text{case}\ xs\ \text{of}\ Nil\ \to (1,());\ Cons\ (a,as) \to (2,(a,as)) \end{array}$$

2.3 Hylomorphisms over data types

Hylomorphisms in triplet form[TM95] are defined as follows.

Definition 8 (Hylomorphism in triplet form) Given morphisms $\phi : G\,A \to A$, $\psi : B \to F\,B$ and natural transformation $\eta : F \dot{\to} G$, the hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ is defined as the the least morphism $f : B \to A$ satisfying the following equation.

$$f = \phi \circ (\eta \circ F\,f) \circ \psi$$

□

Hylomorphisms are powerful in description in that practically every recursion of interest can be specified by hylomorphisms[BdM94]. Hylomorphisms are quite general in that many useful forms are their special cases as defined below. Note that we sometimes omit the subscripts G and F when it is clear from the context.

Definition 9 (Catamorphism, Anamorphism, Map)
Let $(T_F, in_F) = \mu F$, $(T_G, in_G) = \mu G$.

$$\begin{array}{lcl} (\!|_|\!)_F & : & \forall A.\ (F\,A \to A) \to T_F \to A \\ (\!|\phi|\!)_F & = & [\![\phi, id, out_F]\!]_{F,F} \\ \\ [\!(_)\!]_F & : & \forall A.\ (A \to F\,A) \to A \to T_F \\ [\!(\psi)\!]_F & = & [\![in_F, id, \psi]\!]_{F,F} \\ \\ \langle\!|_|\!\rangle_{G,F} & : & (F \dot{\to} G) \to T_F \to T_G \\ \langle\!|\eta|\!\rangle_{G,F} & = & [\![in_G, \eta, out_F]\!]_{G,F} \end{array}$$

□

Catamorphisms $(\!|_|\!)$ are generalized foldr (or reduces) operators that substitute the constructor of a data type with other operation of the same signature. Dually, anamorphisms $[\!(_)\!]$ are generalized unfold (or generations) operators. Maps $\langle\!|_|\!\rangle$ apply a natural transformation on the data structure. Maps can be represented as both catamorphisms and anamorphisms based on the following Hylo Shift Theorem.

Theorem 1 (Hylo Shift)

$$[\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi \circ \eta, id, \psi]\!]_{F,F} = [\![\phi, id, \eta \circ \psi]\!]_{G,G}.$$

□

The Hylo Shift Theorem shows that some computations can be shifted within a hylomorphism. For program fusion, hylomorphisms possess the general laws called the *Hylo Fusion Theorem.*

Theorem 2 (Hylo Fusion)

$$\begin{array}{ll} \text{Left Fusion Law:} & f \circ \phi = \phi' \circ G\,f \implies f \circ [\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi', \eta, \psi]\!]_{G,F} \\ \text{Right Fusion Law:} & \psi \circ g = F\,g \circ \psi' \implies [\![\phi, \eta, \psi]\!]_{G,F} \circ g = [\![\phi, \eta, \psi']\!]_{G,F} \end{array}$$

□

$Decl$	$::=$	$v = b$	(recursive) function definition
b	$::=$	$\lambda v_s.$ case t_0 *of* r	definition body
v_s	$::=$	$v \mid (v_1, \cdots, v_n)$	argument
r	$::=$	$p_1 \to t_1; \cdots; p_n \to t_n$	alternatives
t	$::=$	v	variable
	$\mid$	$(t_1, \cdots, t_n)$	term tuple
	$\mid$	$v\ t$	function application
	$\mid$	$C\ t$	constructor application
p	$::=$	$C\ p$	pattern
	$\mid$	$(p_1, \cdots, p_n)$	pattern tuple
	$\mid$	v	variable

Fig. 1 The language for Recursive Definitions

These laws are quite general in the sense the functions to be fused with, e.g., f and g in Theorem 2, can be any functions. If f and g are restricted to specific hylomorphisms, we could have the following simple but practical *Acid Rain Theorem*[TM95].

Theorem 3 (Acid Rain)

Cata–Hylo Fusion Law:
$$\frac{\tau : \forall A.\ (F\,A \to A) \to F'\,A \to A}{[\![\phi, \eta_1, out_F]\!]_{G,F} \circ [\![\tau in_F, \eta_2, \psi]\!]_{F',L} = [\![\tau(\phi \circ \eta_1), \eta_2, \psi]\!]_{F',L}}$$

Hylo–Ana Fusion Law:
$$\frac{\sigma : \forall A.\ (A \to F\,A) \to A \to F'\,A}{[\![\phi, \eta_1, \sigma out_F]\!]_{G,F'} \circ [\![in_F, \eta_2, \psi]\!]_{F,L} = [\![\phi, \eta_1, \sigma(\eta_2 \circ \psi)]\!]_{G,F'}}$$

□

3 Language

To demonstrate our techniques, we use the language given in Fig. 1 for the description of *recursive* definitions. It is nothing special except that functions are defined in an un-curried way. In order to simplify our presentation, we restrict ourselves to single-recursive data types and functions without mutual recursions, since the standard tupling technique can transform mutual recursive definitions to non-mutual ones. We also assume that recursive function calls are not nested and only occur in the terms of the alternatives in the definition body.

Several simple examples of recursive definitions are given below. The function *sum* sums up all the elements in a list, the function *upto* generates a list of natural numbers between two given numbers, and the function *zip* turns a pair of lists into a list of pairs.

$$\begin{aligned} sum :&\ \ List\ a \to Int \\ sum =&\ \lambda xs.\ \text{case}\ xs\ \text{of} \\ &\qquad Nil \to 0; \\ &\qquad Cons\,(a, as) \to plus\,(a, sum\,as) \end{aligned}$$

$$
\begin{array}{lll}
upto & : & Int \times Int \to List\ Int \\
upto & = & \lambda(b,e).\ \text{case}\ b < e\ \text{of} \\
 & & \quad True \to Nil; \\
 & & \quad False \to Cons\,(b, upto\,(plus\,(b,1),e)) \\
zip & : & List\ A \times List\ B \to List\ (A,B) \\
zip & = & \lambda(xs,ys).\ \text{case}\ (xs,ys)\ \text{of} \\
 & & \quad (Nil,_) \to Nil; \\
 & & \quad (Cons\,(a,as), Nil) \to Nil; \\
 & & \quad (Cons\,(a,as), Cons(b,bs)) \to Cons\,((a,b), zip\,(as,bs))
\end{array}
$$

Here *plus* is the function adding two integers. Next, let's consider definitions of higher order functions, such as

$$
\begin{array}{l}
map : (A \to B) \to List\ A \to List\ B \\
map\ g = \lambda xs.\ \text{case}\ xs\ \text{of} \\
\qquad\qquad Nil \to Nil; \\
\qquad\qquad Cons\,(a,as) \to Cons\,(g\,a, map\,g\,as)
\end{array}
$$

which is not a valid definition in our language since the defined function $map\ g$ is not a variable. The simplest way to solve this problem is to consider $map\ g$ as a "packed" variable and to consider g as a global function. That is, the above definition is considered something like

$$
\begin{array}{l}
map_g = \lambda xs.\ \text{case}\ xs\ \text{of} \\
\qquad\qquad Nil \to Nil; \\
\qquad\qquad Cons\,(a,as) \to Cons\,(g\,a, map_g\,as).
\end{array}
$$

Viewed in this way, $map\ g$ can be regarded as a valid definition. Below is another similar example.

$$
\begin{array}{l}
foldr1 : (B \times C \to C) \to List\ B \to C \\
foldr1\ \oplus = \lambda xs.\ \text{case}\ xs\ \text{of}\ Nil \to error; \\
\qquad\qquad\qquad\qquad\qquad Cons\,(a, Nil) \to a; \\
\qquad\qquad\qquad\qquad\qquad Cons\,(a,as) \to a \oplus (foldr1\ \oplus\ as)
\end{array}
$$

4 Expressing Recursions as Hylomorphisms

In this section, we propose an algorithm to turn recursive definitions into hylomorphisms. Consider the following typical recursive definition of function f.

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

If we can transform the right hand side to $\phi \circ F\,f \circ \psi$, it soon follows that $f = [\![\phi, id, \psi]\!]_{F,F}$ from the definition of hylomorphisms.

4.1 Main idea

The trick to do so is to turn each term t_i $(i = 1, \cdots, n)$ into $g_i\,t'_i$, a suitable function being applied to a new term. Put it in more detail, suppose that we have got that $t_i = g_i\,t'_i$, the original definition becomes:

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

Extracting all g_i's out and adding tag i to t'_i can give a compositional description of f:

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

If g_i can be expressed as $\phi_i \circ F_i\, f$ where F_i is some functor, it soon follows that

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

Now replacing it in the above compositional description of f gives:

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

According to the definition of hylomorphisms, we have $f = [\![\phi, id, \psi]\!]_{F,F}$, where

$$\begin{aligned} F &= F_1 + \cdots + F_n \\ \phi &= \phi_1 \triangledown \cdots \triangledown \phi_n \\ \psi &= \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1, t'_1); \cdots; p_n \to (\mathsf{n}, t'_n). \end{aligned}$$

The essential point of our algorithm is, therefore, to derive a function ϕ_i, a functor F_i and a new term t'_i from each term t_i satisfying $t_i = (\phi_i \circ F\, f)\ t'_i$.

4.2 Deriving ϕ_i, F_i and t'_i from t_i

Our algorithm to derive ϕ_i, F_i and t'_i from t_i is informally as follows.

1. Identify all the occurrences of recursive calls in t_i, say they are $f\ t_{i_1}, \cdots, f\ t_{i_{l_i}}$;
2. Find all the free variables in t_i bound in the definition except in $t_{i_1}, \cdots t_{i_{l_i}}$, say they are $v_{i_1}, \cdots, v_{i_{k_i}}$;
3. Define t'_i by tupling all the arguments of the recursive calls obtained in Step 1 and the free variables obtained in step 2, i.e. $t'_i = (v_{i_1}, \cdots, v_{i_{k_i}}, t_{i_1}, \cdots, t_{i_{l_i}})$.
4. Define F_i according to the construction of t'_i by $F_i = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i}$, where $I_1 = \cdots = I_{l_i} = I$ and Γ returns the type of the given variable.
5. Define ϕ_i by abstracting all recursive function calls in t_i by

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

   where $v'_{i_1}, \cdots, v'_{i_{l_i}}$ are new variables used for replacing those occurrences of recursive calls $f\ t_{i_1}, \cdots, f\ t_{i_{l_i}}$.

4.3 Algorithm for Deriving Hylomorphisms

The derivation algorithm described above is summarized in Fig. 2. The main algorithm is $\mathcal{A}$ which turns a recursive definition into a hylomorphism. The algorithm $\mathcal{A}$ calls the algorithm $\mathcal{D}$ to process each term t_i returning a triple, i.e.,

$$(\{v_{i_1}, \cdots, v_{i_{k_i}}\}, \{(v'_{i_1}, f\ t_{i_1}), \cdots, (v'_{i_{l_i}}, f\ t_{i_{l_i}})\}, t_i[f\ t_{i_1} \mapsto v'_{i_1}, \cdots, f\ t_{i_{l_i}} \mapsto v'_{i_{l_i}}]) = \mathcal{D}[\![t_i]\!].$$

The algorithm $\mathcal{D}$ actually implements most of the algorithm in Section 4.2. It is worth noting that every time an occurrence of recursive call is found, a fresh variable is allocated for the replacement. The algorithm $\mathcal{A}$ is correct and guaranteed to terminate with a successful hylomorphism as result.

$$
\begin{array}{l}
\mathcal{A}[\![f = \lambda vs.\text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!] = (f = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F}) \\
\quad \text{where } F = F_1 + \cdots + F_n \\
\qquad F_i = !\mathbf{1}, \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } k_i = l_i = 0 \\
\qquad\quad = !(\Gamma(v_{i_1})) \times \cdots \times !(\Gamma(v_{i_{k_i}})) \times I_1 \times \cdots \times I_{l_i}, \quad (I_1 = \cdots = I_{l_i} = I), \quad \text{otherwise} \\
\qquad \phi_i = \lambda\ (v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t''_i \\
\qquad \psi = \lambda vs.\text{case } t_0 \text{ of } p_1 \to (1, t'_1); \cdots; p_n \to (\mathsf{n}, t'_n) \\
\qquad t'_i = (v_{i_1}, \cdots, v_{i_{k_i}}, t_{i_1}, \cdots, t_{i_{l_i}}) \\
\qquad \text{where} \\
\qquad\quad (\{v_{i_1}, \cdots, v_{i_{k_i}}\}, \{(v'_{i_1}, f\ t_{i_1}), \cdots, (v'_{i_{l_i}}, f\ t_{i_{l_i}})\}, t''_i) = \mathcal{D}[\![t_i]\!], \quad (i = 1, \cdots, n) \\
\qquad\quad \Gamma(v) = \text{return } v\text{'s type} \\
\\
\qquad\quad \mathcal{D}[\![v]\!] \qquad\quad = \text{if } v \text{ is a global variable then } (\{\}, \{\}, v) \text{ else } (\{v\}, \{\}, v) \\
\qquad\quad \mathcal{D}[\![(t_1, \cdots, t_n)]\!] = (s_1 \cup \cdots \cup s_n, c_1 \cup \cdots \cup c_n, (t'_1, \cdots, t'_n)) \\
\qquad\qquad\qquad\quad \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!], \quad i = 1, \cdots, n \\
\qquad\quad \mathcal{D}[\![v\ t]\!] \qquad = \text{if } v = f \text{ then } (\{\}, \{(u, f\ t)\}, u) \text{ else } (s_v \cup s_t, c_v \cup c_t, t_v\ t_t) \\
\qquad\qquad\qquad\quad \text{where } (s_v, c_v, t_v) = \mathcal{D}[\![v]\!],\ (s_t, c_t, t_t) = \mathcal{D}[\![t]\!] \\
\qquad\qquad\qquad\qquad u \text{ is a fresh variable} \\
\qquad\quad \mathcal{D}[\![C\ t]\!] \qquad = (s, c, C\ t') \text{ where } (s, c, t') = \mathcal{D}[\![t]\!]
\end{array}
$$

Fig. 2 Algorithm for Deriving Hylomorphisms

Theorem 4 The algorithm $\mathcal{A}$ is correct.

Proof: First, $[\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F}$ is a correct definition of hylomorphism because (1) F is a polynomial functor; (2) ϕ has the type of $F\,T_o \to T_o$ and ψ has the type of $T_i \to F\,T_i$, as easily verified, where T_i and T_o denote f's input type and output type respectively; (3) id is a natural transformation from F to F.

Next, we argue that $f = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F}$ by the following calculation.

$$
\begin{array}{ll}
 & f = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F} \\
\equiv & \quad \{\ \text{Definition of hylomorphism}\ \} \\
 & f = \phi_1 \triangledown \cdots \triangledown \phi_n \circ id \circ F\,f \circ \psi \\
\equiv & \quad \{\ \text{Definition of } \psi \text{ and } F\ \} \\
 & f = \phi_1 \triangledown \cdots \triangledown \phi_n \circ (F_1 + \cdots + F_n)\,f \circ \\
 & \qquad (\lambda vs.\text{case } t_0 \text{ of } p_1 \to (1, t'_1); \cdots; p_n \to (\mathsf{n}, t'_n)) \\
\equiv & \quad \{\ \text{Promote function into case expression}\ \} \\
 & f = \lambda v_s.\text{case } t_0 \text{ of } p_1 \to (\phi_1 \circ F_1\,f)\ t'_1; \cdots; p_n \to (\phi_n \circ F_n\,f)\ t'_n \\
\equiv & \quad \{\ \text{Definition of } t'_i\text{'s}\ \} \\
 & f = \lambda v_s.\text{case } t_0 \text{ of} \\
 & \qquad p_1 \to (\phi_1 \circ F_1\,f)\ (v_{1_1}, \cdots, v_{1_{k_1}}, t_{1_1}, \cdots, t_{1_{l_1}}); \\
 & \qquad \cdots; \\
 & \qquad p_n \to (\phi_n \circ F_n\,f)\ (v_{n_1}, \cdots, v_{n_{k_n}}, t_{n_1}, \cdots, t_{n_{l_n}}) \\
\equiv & \quad \{\ \text{Definition of } F_i \text{ and } \phi_i\ \} \\
 & f = \lambda v_s.\text{case } t_0 \text{ of} \\
 & \qquad p_1 \to (\lambda(v_{1_1}, \cdots, v_{1_{k_1}}, v'_{1_1}, \cdots, v'_{1_{l_1}}).\ t''_1)\ (v_{1_1}, \cdots, v_{1_{k_1}}, f\,t_{1_1}, \cdots, f\,t_{1_{l_1}}); \\
 & \qquad \cdots; \\
 & \qquad p_n \to (\lambda(v_{n_1}, \cdots, v_{n_{k_n}}, v'_{n_1}, \cdots, v'_{n_{l_n}}).\ t''_n)\ (v_{n_1}, \cdots, v_{n_{k_n}}, f\,t_{n_1}, \cdots, f\,t_{n_{l_n}})
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \text{ Simplication } \} \\
& f = \lambda v_s.\text{case } t_0 \text{ of} \\
& \quad p_1 \to t_1''[v'_{1_1} \mapsto f\ t_{1_1}, \cdots, v'_{1_{l_1}} \mapsto f\ t_{1_{l_1}}]; \\
& \quad \cdots; \\
& \quad p_n \to t_n''[v'_{n_1} \mapsto f\ t_{n_1}, \cdots, v'_{n_{l_n}} \mapsto f\ t_{n_{l_n}}] \\
\equiv & \{ \text{ Property of } \mathcal{D} \text{ algorithm } \} \\
& f = \lambda v_s.\text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n
\end{array}
$$

□

4.4 Examples

To see the algorithm $\mathcal{A}$ in action, consider some examples defined in Section 3.

We begin by considering a quite simple definition of sum. In this case, we know that $t_1 = 0$ and $t_2 = plus\,(a, sum\,as)$. Applying $\mathcal{D}$ to t_1 and t_2 gives

$$
\begin{array}{ll}
\mathcal{D}[\![0]\!] & = (\{\}, \{\}, 0) \\
\mathcal{D}[\![plus\,(a, sum\,as)]\!] & = (\{a\}, \{(v'_1, sum\,as)\}, plus\,(a, v'_1)).
\end{array}
$$

It follows from $\mathcal{A}$ that

$$
\begin{array}{ll}
t'_1 = (), & \phi_1 = \lambda().\,0 = 0 \\
t'_2 = (a, as), & \phi_2 = \lambda(a, v'_1).\,plus\,(a, v'_1)
\end{array}
$$

and

$$
\begin{array}{l}
\psi = \lambda xs.\text{ case } xs \text{ of } Nil \to (1, ());\ Cons\,(a, as) \to (2, (a, as)) = out_F \\
F = !\mathbf{1} + !Int \times I
\end{array}
$$

Therefore we derive the following hylomorphism for sum:

$$
sum = [\![0 \triangledown \lambda(a, v'_1).\,plus\,(a, v'_1), id, out_F]\!]_{F,F}.
$$

The above hylomorphism is also catamorphism $([\![0 \triangledown \lambda(a, v'_1).\,plus\,(a, v'_1)]\!])_F$.

Our second example is to deal with the recursive definition of $foldr1\ \oplus$. This also appeared in [GLJ93] for the illustration of the limitation of their shortcut deforestation algorithm because $foldr1\ \oplus$ cannot be specified as a catamorphism (since it does not treat all $Cons$ cells identically). With the algorithm $\mathcal{A}$, we can get the following hylomorphism.

$$
\begin{array}{l}
foldr1\ \oplus = [\![\phi, id, \psi]\!]_{F,F} \\
\quad \text{where } F = !\mathbf{1} + !B + !B \times I \\
\qquad \phi = \phi_1 \triangledown \phi_2 \triangledown \phi_3 \\
\qquad\quad \text{where } \phi_1 = error \\
\qquad\qquad \phi_2 = \lambda a.a = id \\
\qquad\qquad \phi_3 = \lambda(a, v'_1).\ (a \oplus v'_1) = \oplus \\
\qquad \psi = \lambda xs.\text{ case } xs \text{ of } Nil \to (1, ()); \\
\qquad\qquad\qquad\qquad Cons\ (a, Nil) \to (2, (a)); \\
\qquad\qquad\qquad\qquad Cons\ (a, as) \to (3, (a, as))
\end{array}
$$

Section 6 will show how it helps removing more intermediate data structures than [GLJ93].

Next we'd like to to show that our derivation algorithm does not restrict to the recursive definitions inducting over a single parameter. Consider the recursive definition of *zip* which inducts over multiple parameters. Applying algorithm $\mathcal{A}$ will give the following hylomorphism.

$$
\begin{array}{l}
zip = [\![Nil \triangledown Nil \triangledown \lambda(a,b,p).Cons((a,b),p), id, \psi]\!]_{F,F} \\
\quad \text{where } F = !\mathbf{1} + !\mathbf{1} + !A \times !B \times I \\
\qquad \psi = \lambda(xs, ys).\ \text{case } (xs, ys) \text{ of} \\
\qquad\qquad (Nil, _) \to (1, ()); \\
\qquad\qquad (Cons\,(a, as), Nil) \to (2, ()); \\
\qquad\qquad (Cons\,(a, as), Cons(b, bs)) \to (3, (a, b, (as, bs)))
\end{array}
$$

The advantage of this transformation is that *zip* now can be fused with other functions by the Hylo Fusion Theorem. Compared with Fegaras's approach[FSZ94] where some new fusion theorems were intentively developed, our algorithm makes it unnecessary to obtain the same effect. More detail discussion can be found in [HIT95].

Finally, we give some other examples.

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

$$
\begin{array}{rcl}
upto & = & [\![in_{L_{Int}}, id, \psi]\!]_{L_{Int}, L_{Int}} \quad \text{where} \\
\psi & = & \lambda(b, e).\ \text{case } b < e \text{ of} \\
 & & \quad True \to (1, ()); \\
 & & \quad False \to (2, (b, (plus\ (b, 1), e)))
\end{array}
$$

5 Restructuring Hylomorphisms

Once a hylomorphism is got, it can be fused with other functions. But not all hylomorphisms have good structures for program fusion to be effectively applied. In this section, we show how to structure the given hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ by arranging ϕ, η and ψ well inside it, so that it could be fused with other functions effectively by the two fusion theorems, namely the Hylo Fusion and the Acid Rain Theorems.

5.1 Suitable Hylomorphisms for Hylo Fusion Theorem

The effective use of the Hylo Fusion Theorem requires that *ϕ (ψ) in a hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ contain as much computation as possible* for the Left (Right) Fusion Law. As an example, consider the following program foo, where N represents the type of natural numbers with two constructors $Zero$ and $Succ$.

$$
\begin{array}{lcl}
foo: & & List\ N \to List\ N \\
foo & = & h \circ [\![in_{L_N}, id, (id + Succ \times id) \circ out_{L_N}]\!]_{L_N, L_N} \\
h & = & \lambda xs.\ \text{case } xs \text{ of} \\
 & & \quad Nil \to Nil; \\
 & & \quad Cons\,(Zero, Nil) \to Nil; \\
 & & \quad Cons\,(Zero, Cons\,(x', xs')) \to Cons\,(Succ\ x', h\ xs'); \\
 & & \quad Cons\,(Succ\ x', xs') \to Cons\,(Succ\ x', h\ xs')
\end{array}
$$

To fuse foo by the Left Fusion Law, we have to derive ϕ' from the equation $h \circ in_{L_N} = \phi' \circ L_N\, h$. But such ϕ' cannot be derived because of the lack of information in in_{L_N}. To solve this problem, we shift some computations to in_{L_N} and get the following program:

$$foo = h \circ [\![in_{L_N} \circ (id + Succ \times id), id, out_{L_N}]\!]_{L_N, L_N}.$$

Because of the knowledge that the value of $Succ\, x$ cannot be $Zero$ and thus the second and third branchs of the case expression in h cannot be taken, we can derive that $\phi' = Nil \triangledown (Cons \circ (Succ \times id))$ and have $foo = [\![\phi', id, out_{L_N}]\!]_{L_N, L_N}$.

Similar cases might happen to the Right Fusion Law.

5.2 Suitable Hylomorphisms for Acid Rain Theorem

The Acid Rain Theorem expects that ϕ and ψ in the hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F} : A \to B$ to be described as $\tau\, in_{F_B}$ and $\sigma\, out_{F_A}$ respectively. Here, τ and σ are polymorphic functions and F_A and F_B are functors defining types A and B respectively. Our Laws for deriving such τ and σ are as follows.

Theorem 5 (Deriving Polymorphic Function) Under the above conditions, τ and σ are defined by the following two laws.

$$\frac{\forall \alpha.\ \ ([\![\alpha]\!])_{F_B} \circ \phi = \phi' \circ G\, ([\![\alpha]\!])_{F_B}}{\tau = \lambda \alpha.\ \phi'}$$

$$\frac{\forall \beta.\ \ \psi \circ [\![(\beta)]\!]_{F_A} = F\, [\![(\beta)]\!]_{F_A} \circ \psi'}{\sigma = \lambda \beta.\ \psi'}$$

Proof sketch: We only show the correctness of the above law for defining τ.

(1) τ has the type of $\forall A.\ (F_B\, A \to A) \to G\, A \to A$ as required because τ is defined for all $\alpha : F_B\, A \to A$ with any A;

(2) We prove that $\phi = \tau\, in_{F_B}$ by the following calculation.

$$\begin{array}{ll} & \phi = \tau\, in_{F_B} \\ \equiv & \quad \{ \text{ Definition of } \tau \ \} \\ & \phi = \phi'[\alpha \mapsto in_{F_B}] \\ \equiv & \quad \{ \text{ Since } ([\![in_{F_B}]\!])_{F_B} = id \text{ and } G\, id = id \ \} \\ & ([\![in_{F_B}]\!])_{F_B} \circ \phi = \phi'[\alpha \mapsto in_{F_B}] \circ G\, ([\![in_{F_B}]\!])_{F_B} \\ \Leftarrow & \quad \{ \text{ Assumption } \} \\ & \forall \alpha.\ \ ([\![\alpha]\!])_{F_B} \circ \phi = \phi' \circ G\, ([\![\alpha]\!])_{F_B} \end{array}$$ □

Because in using the Acid Rain Theorem we have to derive ϕ' (ψ') from ϕ (ψ) for *any* α (β), it is expected that ϕ *(*ψ*) is simple (contains few computations).*

5.3 Algorithm for Structuring Hylomorphisms

Generally, the behavior of a hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ could be understood as follows: ψ generates a recursive structure, η operates on the elements of the structure, and ϕ manipulates on the recursive structure. It is possible for ϕ and ψ

to have the computation that η can do. They are said to have the *least computation* if they do not have computation that η can do.

A hylomorphism $[\![\phi, \eta, \psi]\!]$ is said to be *structural* if ϕ and ψ contain the least computation. Structural hylomorphisms are fit for the two fusion theorems. For the Acid Rain Theorem, it is fitful since ϕ and ψ are simple. For the Hylo Fusion Theorem, since ϕ (ψ) contain the least computation, it implies that $\eta \circ \psi$ ($\phi \circ \eta$) contains the most computation and so $[\![\phi, id, \eta \circ \psi]\!]$ ($[\![\phi \circ \eta, id, \psi]\!]$) is suitable for the Right (Left) Fusion Law. In other words, once a structural hylomorphism is got, we can "shift" the natural transformation (η) freely inside the hylomorphism according to which Fusion Law is to be applied.

Our algorithm for structuring the given hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ is to shift computations from ϕ and ψ into η by factorizing ϕ to $\phi' \circ \eta_\phi$ and ψ to $\eta_\psi \circ \psi'$ so that ϕ' and ψ' contain the least computation, resulting in a structural hylomorphism $[\![\phi', \eta_\phi \circ \eta \circ \eta_\psi, \psi']\!]_{G',F'}$. In the following, we give the algorithm for factorizing ϕ to $\phi' \circ \eta_\phi$, while omitting the dual discussion on ψ.

Let $\phi : G\,A \to A$ be given as:

$$\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$$

where $G = G_1 + \cdots + G_n$ and $\phi_i : G_i\,A \to A$. A typical ϕ_i, as in algorithm $\mathcal{A}$, is defined by:

$$\phi_i = \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t_i$$

The variables with type A are explicitly indicated with $'$ attached and called *recursive variables*.

In order to capture the computations in ϕ_i which can be done by a natural transformation, we define *maximal non-recursive subterms* as follows.

Definition 10 (Maximal Non-Recursive Subterm) A term t_{i_j} is said to be a *non-recursive subterm* of t_i if (1) t_{i_j} is a subterm of t_i; (2) t_{i_j} does not include recursive variables. A non-recursive subterm is said to be *maximal* if it is not a subterm of other non-recursive subterms. □

The essence of our algorithm is to factorize ϕ_i into $\phi'_i \circ \eta_{\phi_i}$ so that all the maximal non-recursive subterms in t_i are shifted into η_{ϕ_i}. Informally, the algorithm for factorizing ϕ_i is as follows.

1. Find all the maximal non-recursive subterms in t_i, say t_{i_1}, ..., $t_{i_{m_i}}$.
2. Let t'_i be the term from t_i with each maximal non-recursive subterm t_{i_j} be replaced by a new variable u_{i_j}, i.e., $t'_i = t_i[t_{i_1} \mapsto u_{i_1}, \cdots, t_{i_{m_i}} \mapsto u_{i_{m_i}}]$.
3. Factorize ϕ_i by extracting all the maximal non-recursive subterms out of t_i as follows.

$$\begin{aligned}\phi_i &= \phi'_i \circ \eta_{\phi_i} \\ &\quad \text{where } \phi'_i = \lambda(u_{i_1}, \cdots, u_{i_m}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t'_i \\ &\qquad\quad \eta_{\phi_i} = \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ (t_{i_1}, \cdots, t_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}})\end{aligned}$$

4. Factorize ϕ by grouping the result of ϕ_i, i.e., $\phi = (\phi'_1 \triangledown \cdots \triangledown \phi'_n) \circ (\eta_{\phi_1} + \cdots + \eta_{\phi_n})$.

$$\mathcal{S}[\![[\![\phi_1 \triangledown \cdots \triangledown \phi_n, \eta, \psi]\!]_{G,F}]\!] = [\![\phi'_1 \triangledown \cdots \triangledown \phi'_n, (\eta_{\phi_1} + \cdots + \eta_{\phi_n}) \circ \eta, \psi]\!]_{G',F}$$

where $\phi'_i = \lambda(u_{i_1}, \cdots, u_{i_m}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t'_i$

$\eta_{\phi_i} = \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ (t_{i_1}, \cdots, t_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}})$

$G' = G'_1 + \cdots + G'_n$

$G_i = !\mathbf{1}$, if $m_i = l_i = 0$

$= !(\Gamma(u_{i_1})) \times \cdots \times !(\Gamma(u_{i_{m_i}})) \times I_1 \times \cdots \times I_{l_i}$, $(I_1 = \cdots = I_{l_i} = I)$, otherwise

where $\lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t_i = \phi_i$, (assume $v'_{i_1}, \cdots, v'_{i_{l_i}}$ are recursive parameters)

$(\{(u_{i_1}, t_{i_1}), \cdots, (u_{i_{m_i}}, t_{i_{m_i}})\}, t'_i) = \mathcal{E}[\![t_i]\!]\ \{v'_{i_1}, \cdots, v'_{i_{l_i}}\}$

$\mathcal{E}[\![v]\!]\ s_r$ = if $v \in s_r$ then $(\{\}, v)$ else $(\{(u, v)\}, u)$
where u is a fresh variable

$\mathcal{E}[\![(t_1, \cdots, t_n)]\!]\ s_r$ = if $\forall i,\ Var_{s_r}(t_i)$ then $(\{(u, (t_1, \cdots, t_n))\}, u)$ else $(w_1 \cup \cdots \cup w_n, (t'_1, \cdots, t'_n))$
where $(w_i, t'_i) = \mathcal{E}[\![t_i]\!]\ s_r\ (i = 1, \cdots, n)$, u is a fresh variable

$\mathcal{E}[\![C\ t]\!]\ s_r$ = if $Var_{s_r}(t')$ then $(\{(u, C\ t)\}, u)$ else $(w, C\ t')$
where $(w, t') = \mathcal{E}[\![t]\!]\ s_r$, u is a fresh variable

$\mathcal{E}[\![v\ t]\!]\ s_r$ = if $Var_{s_r}(t'_v) \wedge Var_{s_r}(t'_t)$ then $(\{(u, v\ t)\}, u)$ else $(w_v \cup w_t, t'_v\ t'_t)$
where $(w_v, t'_v) = \mathcal{E}[\![v]\!]\ s_r$, $(w_t, t'_t) = \mathcal{E}[\![t]\!]\ s_r$, u is a fresh variable

$Var_{s_r}(t)$ = t is a variable $\wedge$ $t \notin s_r$

Fig. 3 Algorithm for Structuring Hylomorphisms

The above algorithm is summarized in Fig. 3. The main transformation is $\mathcal{S}$ which in turn calls transformation $\mathcal{E}$ to process every term t_i for factorizing ϕ_i. Similar to the algorithm in Fig. 2, a fresh variable is allocated every time a non-recursive subterm is found. It will be discarded when the corresponding non-recursive subterm turns out to be non-maximal by the predicate Var. The correctness of the algorithm S is omitted, but it should be noted that $\phi'_1 \triangledown \cdots \triangledown \phi'_n : G'\ A \to A$ and $\eta_{\phi_1} + \cdots + \eta_{\phi_n}$ is natural transformation from G to G'.

Theorem 6 The $\eta_{\phi_1} + \cdots + \eta_{\phi_n}$, derived in the algorithm $\mathcal{S}$, is natural transformation from G to G', i.e.,

$$\eta_{\phi_1} + \cdots + \eta_{\phi_n} : G \dot{\to} G'.$$

Proof: We prove it by the following calculation.

$\eta_{\phi_1} + \cdots + \eta_{\phi_n} : G \dot{\to} G'$

$\equiv$ { Definition of natural transformation }

$\forall f.\ (\eta_{\phi_1} + \cdots + \eta_{\phi_n}) \circ G\ f = G'f \circ (\eta_{\phi_1} + \cdots + \eta_{\phi_n})$

$\equiv$ { Definitions of G and G' }

$\forall f.\ \eta_{\phi_1} \circ G_1 f + \cdots + \eta_{\phi_n} \circ G_n f = G'_1 f \circ \eta_{\phi_1} + \cdots + G'_n f \circ \eta_{\phi_n}$

$\Leftarrow$ { trivial }

$\forall f.\ \eta_{\phi_i} \circ G_i f = G'_i f \circ \eta_{\phi_i}$, $(i = 1, \cdots, n)$

$\equiv$ { Definitions of η_{ϕ_i}, G_i and G'_i }

$\forall f.\ \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ (t_{i_1}, \cdots, t_{i_{m_i}}, f\ v'_{i_1}, \cdots, f\ v'_{i_{l_i}})$

$= \lambda(v_{i_1}, \cdots, v_{i_{k_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ (t_{i_1}, \cdots, t_{i_{m_i}}, f\ v'_{i_1}, \cdots, f\ v'_{i_{l_i}})$, $(i = 1, \cdots, n)$

$\equiv$ { obvious }

$True$ $\square$

5.4 Examples

Let's structure the following hylomorphism obtained in Section 4.4 where $g : A \to B$.

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

In this example, $\phi_1 = \lambda().Nil$, and $\phi_2 = \lambda(a, v_1').Cons\,(\underline{g\ a}, v_1')$. Here there is only one maximal non-recursive subterm as underlined. Our algorithm will move out the computation $g\ a$ out of ϕ_2 as

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

and finally give the following structural hylomorphism:

$$map\ g = [\![Nil \triangledown \lambda(u_{2_1}, v_1').\ Cons(u_{2_1}, v_1'), id + \lambda(a, v_1').(g\ a, v_1'), out_{L_A}]\!]_{L_B, L_A}.$$

Moreover, with Theorem 5, we can get $\phi' = \alpha$ and $\tau = id$. So the above hylomorphism becomes $[\![in_{L_B}, id + \lambda(a, v_1').(g\ a, v_1'), out_{L_A}]\!]_{L_B, L_A}$, a structural hylomorphism for the two fusion theorems.

6 Related Work and Discussions

It has been argued that programming with the use of generic control structures which capture patterns of recursions in a uniform way is very significant in program transformation and optimization[GLJ93, MFP91, SF93, TM95]. Our work is much related to these studies. In particular, our work was greatly motivated by Sheard and Fegaras's work[SF93] and Takano and Meijer's[TM95].

Both of them, essentially, work with a *language without general recursion but containing hylomorphisms as basic components*, advocating *structural functional programming*. Sheard and Fegaras implemented a fusion algorithm called *normalization algorithm*[2] based on the similar theorem like the Hylo Fusion Theorem. Takano and Meijer generalized Gill, Launchbury and Peyton Jones's one-step fusion algorithm relying on functions being written in a highly-stylized *build-cata* forms[GLJ93] (i.e., catamorphisms with data constructors being parameterized), and implemented another one-step fusion algorithm based on the Acid Rain Theorem. All have made it clear that the *fusion process is especially successful if the recursive definitions are expressed in terms of hylomorphisms*.

However, as argued by Launchbury and Sheard[LS95], although structural functional programming contributes much to program transformation and optimization, it is unrealistic in real functional programming. It is impractical to force programmers to define their recursive definitions only in terms of the specific forms like hylomorphisms, in which a lot of abstract categorical concepts are embedded.

To remedy this situation, Launchbury and Sheard[LS95] gave an algorithm to turn recursive definitions into build-cata forms. The major problem still left is

[2] Sheard and Fegaras's normalization algorithm first only worked with the language containing folds (i.e., catamorphisms) as basic components. It has been extended to work with languages containing so-called homomorphisms (i.e., hylomorphisms) in [KL94].

that many programs cannot be fused because some recursive definitions cannot be specified in build-cata forms. As an example, consider the program

$$foldr1\ \oplus\ \circ\ map\ g$$

which accepts a list $Cons\,(x_1, \cdots, Cons\,(x_{n-1}, Cons(x_n, Nil))\cdots)$ and returns $g\,x_1 \oplus (\cdots (g\,x_{n-1} \oplus g\,x_n)\cdots)$. Since $foldr1\ \oplus$ is not a catamorphism (Section 4.4), the algorithm in [LS95] will fail and leave the intermediate data structure produced by $map\ g$ remained. Our algorithm can solve this problem. With the results in Sections 4.4 and 5.4, we have

$$[\![error \triangledown id \triangledown \oplus, id, \psi]\!]_{F,F} \circ [\![in_{L_B}, id + \lambda(a, v_1').(g\,a, v_1'), out_{L_A}]\!]_{L_B,L_A},$$

and we can derive a polymorphic function

$$\sigma = \lambda\beta.\ (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times \beta^{-1} \circ \iota_2)) \circ dist) \circ L_A\,\beta \circ \beta$$

from ψ such that $\psi = \sigma out_{L_B}$ according to Theorem 5, where $dist$ is a natural transformation defined as follows.

$$\begin{array}{lcl} dist & : & X \times (Y + Z) \to X \times Y + X \times Z \\ dist(x, (1, y)) & = & (1, (x, y)) \\ dist(x, (2, z)) & = & (2, (x, y)) \end{array}$$

In the following, we demonstrate how the Ana-Hylo Fusion Law is applied to eliminate the intermediate data structure in the program $foldr1\ \oplus\ \circ\ map\ g$.

$$\begin{array}{ll} & foldr1\ \oplus\ \circ\ map\ g \\ = & \quad \{\ \text{Replace } foldr1\ \oplus \text{ and } map\ g \text{ with their hylomorphisms}\ \} \\ & [\![error \triangledown id \triangledown \oplus, id, \sigma out_{L_B}]\!]_{F,F} \circ [\![in_{L_B}, id + g \times id, out_{L_A}]\!]_{L_B,L_A} \\ = & \quad \{\ \text{Acid Rain Theorem (Hylo-Ana Fusion Law)}\ \} \\ & [\![error \triangledown id \triangledown \oplus, id, \sigma((id + g \times id) \circ out_{L_A})]\!]_{F,F} \end{array}$$

Now we focus on the transformation of the third part in the above hylomorphism.

$$\begin{array}{ll} & \sigma((id + g \times id) \circ out_{L_A}) \\ = & \quad \{\ \text{Definition of } \sigma, \text{ let } h = id + g \times id\ \} \\ & (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times (h \circ out_{L_A})^{-1} \circ \iota_2)) \circ dist) \circ L_A\,(h \circ out_{L_A}) \circ h \circ out_{L_A} \\ = & \quad \{\ \text{Definition of } L_A, \text{ and } h.\ \} \\ & (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times (h \circ out_{L_A})^{-1} \circ \iota_2)) \circ dist) \\ & \quad \circ (id + (g \times (h \circ out_{L_A}))) \circ out_{L_A} \\ = & \quad \{\ \text{Definition of } L_A\ \} \\ & (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times (h \circ out_{L_A})^{-1} \circ \iota_2)) \circ dist) \\ & \quad \circ (id + g \times h) \circ L_A out_{L_A} \circ out_{L_A} \\ = & \quad \{\ \text{Definition of } h\ \} \\ & (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times (h \circ out_{L_A})^{-1} \circ \iota_2)) \circ dist) \\ & \quad \circ (id + g \times (id + g \times id)) \circ L_A out_{L_A} \circ out_{L_A} \\ = & \quad \{\ \text{Move } dist \text{ backwards}\ \} \\ & (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times (h \circ out_{L_A})^{-1} \circ \iota_2))) \\ & \quad \circ (id + dist \circ (g \times (id + g \times id))) \circ L_A out_{L_A} \circ out_{L_A} \end{array}$$

$$
\begin{array}{ll}
= & \{\ \text{Transformation property of } dist\ \} \\
& (\iota_1 \triangledown (\iota_2 \circ \pi_1 \triangledown \iota_3 \circ (id \times (h \circ out_{L_A})^{-1} \circ \iota_2))) \\
& \quad \circ (id + (g \times id + g \times (g \times id)) \circ dist) \circ L_A out_{L_A} \circ out_{L_A} \\
= & \{\ \text{Since } (f \triangledown g) \circ (p + q) = f \circ p \triangledown g \circ q,\ out_{L_A}^{-1} = in_{L_A}\ \} \\
& (\iota_1 \triangledown (\iota_2 \circ g \circ \pi_1 \triangledown \iota_3 \circ (g \times (in_{L_A} \circ h^{-1} \circ \iota_2 \circ (g \times id)))) \circ dist) \circ L_A out_{L_A} \circ out_{L_A} \\
= & \{\ \text{Since } h^{-1} \circ \iota_2 = \iota_2 \circ (g \times id)^{-1}\ \} \\
& (\iota_1 \triangledown (\iota_2 \circ g \circ \pi_1 \triangledown \iota_3 \circ (g \times in_{L_A} \circ \iota_2)) \circ dist) \circ L_A out_{L_A} \circ out_{L_A}
\end{array}
$$

It then derived the following hylomorphism:

$$[\![error \triangledown id \triangledown \oplus, id,\ \triangledown \iota_3 \circ (g \times in_{L_A} \circ \iota_2)) \circ dist) \circ L_A out_{L_A} \circ out_{L_A}]\!]_{F,F}.$$

Inlining the above derived hylomorphism, named prg, would give the following familiar program:

$$
\begin{array}{l}
prg = \lambda xs.\ \text{case } xs \text{ of} \\
\qquad\qquad Nil \to error; \\
\qquad\qquad Cons\,(a, Nil) \to g\ a \\
\qquad\qquad Cons\,(a, as) \to g\ a \oplus prg\ as
\end{array}
$$

where the intermediate data structure produced by $map\,g$ no longer exists. Compared with Launchbury and Sheard's algorithm, ours is more general and powerful.

Our work is also related to the discussion on the fusion of recursions inducting over multiple data structures. Rather than introducing new fusion theorems as in [FSZ94], we can handle it by turning them into hylomorphisms.

7 Conclusion

We have successfully given the algorithm to turn recursive definitions to structural hylomorphisms in order to enable program fusion. Our algorithm is a *two-stage abstraction*, abstracting recursive function calls to derive hylomorphisms and abstracting maximal non-recursive subterms to structure hylomorphisms. A prototype of our algorithm has been implemented and tested extensively, showing its promise to be embedded in a real system. Finally, our algorithm gives the evidence that hylomorphisms can specify all recursions of interests as claimed in [BdM94].

Acknowledgement

This paper owes much to the thoughtful and helpful discussions with Akihiko Takano. Thanks are also to Fer-Jan de Vries for reading the manuscript and making a number of helpful suggestions.

References

[BdM94] R.S. Bird and O. de Moor. Relational program derivation and context-free language recognition. In A.W. Roscoe, editor, *A Classical Mind*, pages 17–35. Prentice Hall, 1994.

[Chi92] W. Chin. Safe fusion of functional expressions. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, Ca., June 1992.

[Fok92] M. Fokkinga. *Law and Order in Algorithmics*. Ph.D thesis, Dept. INF, University of Twente, Netherlands, 1992.

[FSZ94] L. Fegaras, T. Sheard, and T. Zhou. Improving programs which recurse over multiple inductive structures. In *Proc. PEPM'94*, June 1994.

[GLJ93] A. Gill, J. Launchbury, and S.P. Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[HIT95] Z. Hu, H. Iwasaki, and M. Takeichi. Making recursions manipulable by constructing medio-types. Technique report METR 95–04, Faculty of Engineering, University of Tokyo, June 1995.

[KL94] R.B. Kieburtz and J. Lewis. Algebraic design language. Technical Report OGI, Tech-report 94-002, Dept. of Computer Sceience and Engineering, Oregon Graduate Institution of Science and Technology, 1994.

[LS95] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, La Jolla, California, June 1995.

[Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, (14):255–279, August 1990.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture (LNCS 523)*, pages 124–144, Cambridge, Massachusetts, August 1991.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, June 1993.

[Tak87] M. Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.

[TM95] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. of ESOP (LNCS 300)*, pages 344–358, 1988.