AN EXTENSION OF THE ACID RAIN THEOREM

ZHENJIANG HU
Department of Information Engineering, The University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan
E-mail: hu@ipl.t.u-tokyo.ac.jp

HIDEYA IWASAKI
Department of Computer Science, Tokyo University of Agriculture and Technology
2–24–16 Naka-cho, Koganei, Tokyo 184, Japan
E-mail: iwasaki@ipl.ei.tuat.ac.jp

MASATO TAKEICHI
Department of Information Engineering, The University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113, Japan
E-mail: takeichi@u-tokyo.ac.jp

ABSTRACT

Program *fusion* (or *deforestation*) is a well-known transformation whereby compositions of several pieces of code are fused into a single one, resulting in an efficient functional program without intermediate data structures. Recent work has made it clear that fusion transformation is especially successful if recursions are expressed in terms of *hylomorphisms*. The point of this success is that fusion transformation proceeds merely based on a simple but effective rule called *Acid Rain Theorem* [10]. However, there remains a problem. The Acid Rain Theorem can only handle hylomorphisms inducting over a *single* data structure. For hylomorphisms, like *zip*, which induct over *multiple* data structures, it will leave some of the data structures remained which should be removed. In this paper, we extend the Acid Rain Theorem so that it can deal with such hylomorphisms, enabling more intermediate data structures to be eliminated.

1. Introduction

Functional programming 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.

A classical toy example [11] of this is to compute the **s**um of the **s**quares of the numbers **f**rom 1 to n. We might write it as:

$$ssf\ \ n = sum\,(map\ square\,(upto\,(1, n))).$$

This generates the list of numbers from 1 to n, forms the second list by squaring each number in the first list, and returns the sum of the numbers in the second. The modularity of this functional program comes from dividing into three components: generate the numbers, square the numbers, and calculate the sum. This in turn depends on the use of intermediate lists to communicate between these components: *upto* $(1, n)$ passes the list $[1, 2, \cdots, n]$ to *map square*, which passes the list $[1, 4, \cdots, n^2]$ to *sum*. Unfortunately, all these intermediate lists need to be produced, traversed, and discarded, degrading execution time dreadfully.

Program *fusion* (or *deforestation*) [11, 1] is a well-known transformation whereby compositions of several pieces of code are fused into a single one, resulting in an efficient functional program without intermediate data structures. Recent work [4, 9, 7, 10] has made it clear that fusion transformation is especially successful if recursions are expressed in terms of *hylomorphisms*. The point of this success is that fusion transformation can be done merely based on a cheap rule called *Acid Rain Theorem* [10], which tells us that under a certain condition the composition of two hylomorphisms can be merged into a single one and the intermediate data structure produced by one hylomorphism and then consumed by the other are eliminated. For example, transforming the above *ssf* with the Acid Rain Theorem gives (Section 3.1):

$$\begin{array}{lcl} \mathit{ssf}\ n & = & \mathit{ssf}'\ (1, n) \\ \textbf{where} & & \\ \mathit{ssf}'\ (m, n) & = & \text{case}\ (m > n)\ \text{of} \\ & & \quad \mathit{True} \rightarrow 0 \\ & & \quad \mathit{Flase} \rightarrow \mathit{square}\ m + \mathit{ssf}'\,(m + 1, n) \end{array}$$

which is more efficient because all intermediate lists have been eliminated.

Despite its practical and effective use in program fusion, the Acid Rain Theorem has a major problem in dealing with hylomorphisms inducting over multiple data structures (Section 3.2). Basically, the Acid Rain Theorem does not provide any mechanism to standardize the consumption of multiple data structures simultaneously, although it shows how to standardize the way of consuming or producing a single data structure in terms of a proper polymorphic function. Therefore, for hylomorphisms which induct over *multiple* data structures it will leave some of the data structures remained which are expected to be removed.

The goal of this paper is to extend the Acid Rain Theorem so that it can deal with hylomorphisms inducting over multiple data structures, eliminating more intermediate data structures. Our main contributions are as follows.

- We provide a novel way to standardize the consumption of multiple data structures simultaneously (Section 4.1) by defining a combination of some polymorphic functions. Note that standardizing the way of producing several data structures can be simply captured by products of several functions; each producing a single data structure.

- We extend the Acid Rain Theorem (Section 4.2) so that it can deal with the fusion of hylomorphisms inducting over multiple data structures in a shortcut way. Several examples are given showing the promise of our work.

- Compared with the previous work on the manipulation of recursions over multiple data structures as discussed in [2] and more generally studied in [5, 6], our newly proposed approach is more practical in that the fusion transformation can be done by a simple substitution without need for deriving a new function satisfying a certain equation (i.e., fusible condition). This is the spirit of the Acid Rain Theorem in contrast to the general fusion laws (as in the Hylo Fusion Theorem).

The organization of the paper is as follows. After reviewing previous works on program calculation and introducing some notational conventions in Section 2, we demonstrate how the Acid Rain Theorem is used in practical program fusion transformation and explain the problem in Section 3. Then in Section 4.2, we propose our main contributions of this paper showing how to standardize the way of consuming multiple data structures and how to generalize the Acid Rain Theorem so that it enables effective fusion for hylomorphisms over multiple data structures. Finally, we explain related works and make concluding remarks in Section 5 and Section 6, respectively.

2. Preliminary

Before addressing how to generalize the Acid Rain Theorem, we review previous works on program calculation [8, 3, 10]. Rather than being involved in a precise discussion in terms of category theory, we shall impose emphasis on the results which will be used later and the notational conventions to which we hope that the readers could get used.

2.1. Functors

Endofunctors can capture both data structure and control structure in a type definition. In this paper, we assume that all 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 2.1 (Identity)
The identity functor I on type X and its operation on functions are defined as follows.

$$\begin{array}{lcl} I\,X & = & X \\ I\,f & = & f \end{array} \qquad \Box$$

Definition 2.2 (Constant)
The constant functor $!A$ on type X and its operation on functions are defined as follows.

$$\begin{array}{lcl} !A\,X & = & A \\ !A\,f & = & id \end{array}$$

where id stands for the identity function. $\Box$

Definition 2.3 (Product)
The product $X \times Y$ of two types X and Y and its operation to functions are defined as follows.

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

Some related operators are:

$$\begin{array}{lcl} \pi_1\ (a,b) & = & a \\ \pi_2\ (a,b) & = & b \\ (f \vartriangle g)\ a & = & (f\,a,\ g\,a). \end{array}$$ $\Box$

Definition 2.4 (Separated Sum)
The separated sum $X + Y$ of two types X and Y and its operation to functions are defined as follows.

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

Some related operators are:

$$\begin{array}{lcl} (f \triangledown g)\ (1,x) & = & f\ x \\ (f \triangledown g)\ (2,y) & = & g\ y. \end{array}$$ $\Box$

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

$$\begin{array}{lcl} +_{i=1}^{n} X_i & = & \cup_{i=1}^{n}(\{\mathsf{i}\} \times X_i) \\ (+_{i=1}^{n} f_i)\,(\mathsf{j}, x) & = & (\mathsf{j},\ f_j\,x). \end{array}$$

2.2. Data Types as Initial Fixed Points of Functors

A data type is a collection of data constructors denoting how each element of the data type can be constructed in a finite way. The definition of a data type can be captured by an endofunctor [3]. Let's look at a concrete example. Consider the data type of *cons lists* with elements of type A, which is usually defined by

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

In our framework, we shall use the following endofunctor to capture the recursive structure of the data type:

$$F_{L_A} = !\mathbf{1} + !A \times I$$

where $\mathbf{1}$ denotes the final object, corresponding to $()$[a]. In fact, the definition of F_{L_A} can be automatically derived from the original definition of *List* A [9]. Besides, we use $in_{F_{L_A}}$ to capture the *data constructors* in *List* A by grouping all data constructors with $\triangledown$, i.e.,

$$in_{F_{L_A}} = Nil \triangledown Cons.$$

It follows that $List\ A = in_{F_{L_A}}(F_{L_A}(List\ A))$. The $in_{F_{L_A}}$ has its inverse, denoted by $out_{F_{L_A}}$, which captures the data destructor of *List* A, i.e.,

$$\begin{array}{lcl} out_{F_{L_A}}\ Nil & = & (1, ()) \\ out_{F_{L_A}}\ (Cons(a, as)) & = & (2, (a, as)). \end{array}$$

One advantage of formulating a data type with an endofunctor as the above is that recursive functions induction over it and the transformation rules for these recursive functions can be defined in a quite uniform and concise way, as will be seen in the next section.

2.3. Hylomorphisms

Hylomorphism, a general recursive form covering the well-known *catamorphism* and *anamorphism* as its special cases, is defined in triplet form [10] as follows.

Definition 2.5 (Hylomorphism in triplet form)
Let F and G be two functors. Given $\phi : G\,A \to A$, $\psi : B \to F\,B$ and natural transformation $\eta : F \stackrel{\cdot}{\to} G$, the hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F} : B \to A$ is defined as the least fixed point of the following equation.

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

□

Hylomorphisms (Hylo for short) are powerful in description in that practically every recursion of interest (e.g., primitive recursions) can be specified by them [6]. They are considered to be an ideal intermediate recursive form for calculating efficient functional programs.

Hylomorphisms enjoy many useful transformation laws. One useful law is called *Hylo shift law*:

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

showing that a natural transformation can be shifted inside a hylomorphism. It is very useful for transformation inside a hylomorphism.

For program fusion, hylomorphisms possess the general laws called the *Hylo Fusion Theorem*.

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

Theorem 2.1 (Hylo Fusion)

Left Fusion Law:

$$\frac{f \circ \phi = \phi' \circ G\, f}{f \circ [\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi', \eta, \psi]\!]_{G,F}}$$

Right Fusion Law:

$$\frac{\psi \circ g = F\, g \circ \psi'}{[\![\phi, \eta, \psi]\!]_{G,F} \circ g = [\![\phi, \eta, \psi']\!]_{G,F}} \qquad \Box$$

These laws are quite general in the sense that the functions to be fused with, e.g., f and g in Theorem 2.1, can be any functions. If f and g are restricted to specific hylomorphisms, we can have the following simple but practical *Acid Rain Theorem*[10].

Theorem 2.2 (Acid Rain)

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

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

$\Box$

3. Acid Rain Theorem for Hylo Fusion

In this section, we shall demonstrate how the Acid Rain Theorem is used in practical program fusion transformation and explain what the problem is.

3.1. An Example

Let's explain briefly how the Acid Rain Theorem works with an example, more detailed discussion can be found in [10]. The precondition of the Acid Rain Theorem is that every recursion has to be described in terms of hylomorphisms where the way of producing and/or consuming data structures are standardized by polymorphic functions. For instance, to fuse the program *ssf* given in Introduction, we have to define *sum*, *map square* and *upto* by the following hylomorphisms, as shown below.

$$\begin{array}{lcl}
sum & = & [\![0 \triangledown plus,\ id,\ out_{F_{L_A}}]\!]_{F_{L_A},F_{L_A}} \\
map\ square & = & [\![in_{F_{L_A}},\ id + (square \times id),\ out_{F_{L_A}}]\!]_{F_{L_A},F_{L_A}} \\
upto & = & [\![in_{F_{L_A}},\ id,\ \psi]\!]_{F_{L_A},F_{L_A}} \\
 & & \textbf{where} \\
 & & \quad \psi = \lambda(m,n).\, \text{case } (m > n) \text{ of} \\
 & & \qquad\qquad True \to (1, ()) \\
 & & \qquad\qquad False \to (2, (m, (m + 1, n)))
\end{array}$$

In [6], an automatic algorithm has been proposed to derive such hylomorphisms from general recursive definitions. To help the reader be familiar with hylo notation, let's inline the above hylomorphism for *sum*. According to the definition of hylomorphism, we know that

$$sum = (0 \triangledown plus) \circ (id \circ F_{L_A} sum) \circ out_{F_{L_A}}.$$

Notice that $F_{L_A} sum = id + id \times sum$. Now, expanding $out_{F_{L_A}}$ by case analysis gives:

$$\begin{array}{lcl}
sum\ Nil & = & ((0 \triangledown plus) \circ F_{L_A} sum)\ (1, ())\\
 & = & ((0 \triangledown plus)\ (1, ())\\
 & = & 0\\
sum\ (Cons(a, as)) & = & ((0 \triangledown plus) \circ F_{L_A} sum)\ (2, (a, as))\\
 & = & ((0 \triangledown plus)\ (1, (a, sum\ as))\\
 & = & plus\,(a, sum\ as)
\end{array}$$

which is a recursive definition as we usually see.

Now we return to our main line of application of the Acid Rain Theorem. Let

$$ssf\ \ n = ssf'\,(1, n).$$

We demonstrate the fusion transformation by the following calculation.

$$\begin{array}{ll}
 & ssf'\\
= & \quad \{\ \text{definition of } ssf\ \}\\
 & sum \circ map\ \ square \circ upto\\
= & \quad \{\ \text{standardize recursions with hylos (we omit subscript } F_{L_A} \text{ below)}\ \}\\
 & [\![0 \triangledown plus,\ id,\ out]\!] \circ [\![in,\ id + (square \times id),\ out]\!] \circ [\![in,\ id,\ \psi]\!]\\
= & \quad \{\ \text{fuse the first two hylos by the Acid Rain Theorem}\ \}\\
 & [\![0 \triangledown plus, id + (square \times id),\ out]\!] \circ [\![in,\ id,\ \psi]\!]\\
= & \quad \{\ \text{fuse the two hylos again by the Acid Rain Theorem}\ \}\\
 & [\![0 \triangledown plus, id + (square \times id),\ \psi]\!]
\end{array}$$

Inlining the above hylomorphism can give exactly the same efficient program in Introduction. It is very attractive to eliminate intermediate data structures in such a simple but effective way.

3.2. The Problem

Despite its practical and effective use in program fusion, the Acid Rain Theorem has a major problem in dealing with hylomorphisms inducting over *multiple* data structures. In fact, the transformation rules in this theorem requires that a hylomorphism

$$[\![\phi, \eta, \psi]\!]_{G,F} : A \rightarrow B$$

which can be fused with other hylomorphisms should be in the form of

$$[\![\tau\, in_{F_B},\, \eta,\, \psi]\!]_{G,F} \quad \text{or} \quad [\![\phi,\, \eta,\, \sigma\, out_{F_A}]\!]_{G,F}.$$

Here, F_A and F_B are functors defining types A and B respectively, and τ and σ are polymorphic functions standardizing the way in which the resulting data of type B are produced and the way in which the input data of type A are consumed by the hylomorphism respectively.

Even though the above two restrictive hylomorphic forms are very powerful, there are still some important functions that cannot be effectively captured by them. For instance, the hylomorphisms like *zip* and *take*, which essentially consume (induct over) *multiple* data structures at the same time, cannot not be standardized by a polymorphic function σ as above. Hence, some or all of these intermediate data structures will be remained after fusion by the Acid Rain Theorem. For example, let's consider the following program:

$$zmm\ x\ y = zip\ (map\ double\ x)\ (map\ square\ y)$$

where *zip* is a function to turn a pair of lists into a list of pairs which is usually defined by:

```
zip  =  λx.λy. case x of
                  Nil → Nil
                  Cons(a, as) → case y of
                                  Nil → Nil
                                  Cons(b, bs) → Cons((a, b), zip as bs)
```

If we rewrite it into a hylomorphism in the above forms, we can only standardize the way of consuming one parameter but have to ignore the other. As a result, the application of the Acid Rain Theorem on the program *zmm* will remove only one intermediate data structure produced by one function, e.g., *map double*, while remaining that produced by the other function, e.g., *map square*.

This limitation of the Acid Rain Theorem has already been recognized elsewhere. In [4, 7], it is said that although considering *zip* as a list producer is straightforward it cannot use *foldr/build* rule (i.e., a specialization of the Acid Rain Theorem on the data structure of lists.) so that *both* input lists to *zip* may be removed. However, no satisfactory solution has been given yet.

4. Generalizing the Acid Rain Theorem

In this section, we show how to standardizing the way of consuming multiple data structures simultaneously, and how to generalize the Acid Rain Theorem so that it can enable effective fusion for hylomorphisms inducting over multiple data structures.

4.1. Standardization of Consumption of Multiple Data Structures

As discussed before, the main idea of the Acid Rain Theorem is that the way of producing and consuming data structures should be standardized by some polymorphic functions. For the effective fusion of functions over multiple data structures, we need to find a method to standardize their way of consuming them.

Without loss of generality, we consider how to standardize the way of consuming *two* data structures at the same time. That is, we are looking for a way to express a hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F} : (A \times B) \to C$ into $[\![\phi, \eta, \psi']\!]_{G,F} : (A \times B) \to C$ so that ψ' naturally describe the way of consuming A and B at the same time. For this purpose, we assume that F_A and out_A are the functor and data destructor for type A, and that F_B and out_B are the functor and data destructor for type B. We define a polymorphic function $\bowtie$ for merging two structures into one:

$$\bowtie\, : \; \forall X, Y.\, (F_1 X \times F_2 Y) \to F\,(X \times Y)$$

where F_1, F_2 and F are functors. One natural property [12] with this polymorphic function is that for any f and g we have

$$\bowtie \circ (F_1\, f \times F_2\, g) = F\,(f \times g) \circ \bowtie\, .$$

Associated with $\bowtie$, we define a lifted operator[b]

$$(f \mathbin{\hat{\bowtie}} g)\; (x, y) = (f\;\, x) \bowtie (g\;\, y).$$

It soon follows that

$$\begin{aligned} (F_1 f) \mathbin{\hat{\bowtie}} (F_2\, g) &= F\,(f \times g) \circ (id \mathbin{\hat{\bowtie}} id) && (1) \\ (f_1 \circ g_1) \mathbin{\hat{\bowtie}} (f_2 \circ g_2) &= (f_1 \mathbin{\hat{\bowtie}} f_2) \circ (g_1 \times g_2) && (2) \end{aligned}$$

Now combining the techniques for standardizing the way of consuming one data structure, we use the following form for ψ' to standardize the way of simulatneously consuming two data structures:

$$\frac{\begin{array}{lcl} \sigma_A & : & \forall X.\, (X \to F_A\, X) \to (X \to F_1\, X) \\ \sigma_B & : & \forall X.\, (X \to F_B\, X) \to (X \to F_2\, X) \\ \bowtie & : & \forall X, Y.\, (F_1 X \times F_2\, Y) \to F\,(X \times Y) \end{array}}{\sigma_A\, out_{F_A} \mathbin{\hat{\bowtie}}\, \sigma_B\, out_{F_B}}$$

a combination of three polymorphic functions, namely $\bowtie$, σ_A and σ_B.

4.2. Extension of the Acid Rain Theorem

After formulating the induction of multiple data structures simultaneously, we are ready to extend the Acid Rain Theorem.

[b]For concise presentation, we use the infix notation for $\bowtie$ and $\hat{\bowtie}$, i.e., $\bowtie\; (x, y) = x \bowtie y$ and $\hat{\bowtie}\; (f \times g) = f \mathbin{\hat{\bowtie}} g$.

Theorem 4.1 (Extended Acid Rain)

$$\frac{\begin{array}{lcl} \sigma_A & : & \forall X.\,(X \to F_A\,X) \to (X \to F_1\,X) \\ \sigma_B & : & \forall X.\,(X \to F_B\,X) \to (X \to F_2\,X) \\ \bowtie & : & \forall X,Y.\,(F_1 X \times F_2\,Y) \to F\,(X \times Y) \\ \mathit{hylo} & = & [\![\phi,\,\eta,\,\sigma_A\,\mathit{out}_{F_A} \bowtie \sigma_B\,\mathit{out}_{F_B}]\!]_{G,F} \\ \mathit{hylo}_A & = & [\![\mathit{in}_{F_A},\,\eta_A,\,\psi_A]\!]_{F_A,M} \\ \mathit{hylo}_B & = & [\![\mathit{in}_{F_B},\,\eta_B,\,\psi_B]\!]_{F_B,N} \end{array}}{\mathit{hylo} \circ (\mathit{hylo}_A \times \mathit{hylo}_B) = [\![\phi,\,\eta,\,\sigma_A\,(\eta_A \circ \psi_A) \bowtie \sigma_B\,(\eta_B \circ \psi_B)]\!]_{G,F}}$$

Proof: We may follow the similar proof as in [10] according to the Free Theorem [12]. Here we prove it in another way by using the Right Fusion Law in the Hylo Fusion Theorem. We argue by the following calculation.

$$\begin{array}{cl} & (\sigma_A\,\mathit{out}_{F_A} \bowtie \sigma_B\,\mathit{out}_{F_B}) \circ (\mathit{hylo}_A \times \mathit{hylo}_B) \\ = & \quad \{\ (2)\ \} \\ & (\sigma_A\,\mathit{out}_{F_A} \circ \mathit{hylo}_A) \bowtie (\sigma_B\,\mathit{out}_{F_B} \circ \mathit{hylo}_B) \\ = & \quad \{\ \text{To be proved below}\ \} \\ & (F_1\,\mathit{hylo}_A \circ \sigma_A\,(\eta_A \circ \psi_A)) \bowtie (F_2\,\mathit{hylo}_B \circ \sigma_B\,(\eta_B \circ \psi_B)) \\ = & \quad \{\ (2)\ \} \\ & (F_1\,\mathit{hylo}_A \bowtie F_2\,\mathit{hylo}_B) \circ (\sigma_A\,(\eta_A \circ \psi_A) \times \sigma_B\,(\eta_B \circ \psi_B)) \\ = & \quad \{\ (1)\ \} \\ & F\,(\mathit{hylo}_A \times \mathit{hylo}_B)\ \circ (\mathit{id} \bowtie \mathit{id})\ \circ (\sigma_A\,(\eta_A \circ \psi_A) \times \sigma_B\,(\eta_B \circ \psi_B)) \\ = & \quad \{\ (2)\ \} \\ & F\,(\mathit{hylo}_A \times \mathit{hylo}_B)\ \circ (\sigma_A\,(\eta_A \circ \psi_A) \bowtie \sigma_B\,(\eta_B \circ \psi_B)) \end{array}$$

According to the Hylo Fusion Theorem, we soon have the result of

$$\mathit{hylo} \circ (\mathit{hylo}_A \times \mathit{hylo}_B) = [\![\phi,\,\eta,\,\sigma_A\,(\eta_A \circ \psi_A) \bowtie \sigma_B\,(\eta_B \circ \psi_B)]\!]_{G,F}$$

as we'd like to prove. Now to complete the above proof, we still need to show that

$$\begin{array}{lcl} \sigma_A\,\mathit{out}_{F_A} \circ \mathit{hylo}_A & = & F_1\,\mathit{hylo}_A \circ \sigma_A\,(\eta_A \circ \psi_A) \\ \sigma_B\,\mathit{out}_{F_B} \circ \mathit{hylo}_B & = & F_2\,\mathit{hylo}_B \circ \sigma_B\,(\eta_B \circ \psi_B). \end{array}$$

In the following, we shall only prove the first equation because the second can be proved in a similar way. The Free Theorem [12] associated with the type of σ_A is that

$$\frac{\psi_1 \circ g = F_A\,g \circ \psi_2}{\sigma_A\,\psi_1 \circ g = F_1\,g \circ \sigma_A \psi_2}.$$

By taking $\psi_1 := \mathit{out}_{F_A}$, $g := \mathit{hylo}_A$ and $\psi_2 := \eta_A \circ \psi_A$, this rule is instantiated to

$$\frac{\mathit{out}_{F_A} \circ \mathit{hylo}_A = F_A\,\mathit{hylo}_A \circ (\eta_A \circ \psi_A)}{\sigma_A\,\mathit{out}_{F_A} \circ \mathit{hylo}_A = F_1\,\mathit{hylo}_A \circ \sigma_A(\eta_A \circ \psi_A)}.$$

This premise holds because

$$\begin{array}{ll}
 & out_{F_A} \circ hylo_A = F_A\ hylo_A \circ (\eta_A \circ \psi_A) \\
\equiv & \quad \{\ in_{F_A} \text{ is the inversion of } out_{F_A}\ \} \\
 & hylo_A = in_{F_A} \circ F_A\ hylo_A \circ (\eta_A \circ \psi_A) \\
\equiv & \quad \{\ \text{Definition of Hylomorphism}\ \} \\
 & hylo_A = [\![in_{F_A},\ id,\ \eta_A \circ \psi_A]\!]_{F_A, F_A} \\
\equiv & \quad \{\ \text{Hylo Shift Law}\ \} \\
 & hylo_A = [\![in_{F_A},\ \eta_A,\ \psi_A]\!]_{F_A, M} \\
\equiv & \quad \{\ \text{Assumption}\ \} \\
 & True
\end{array}$$

□

4.3. An Example

To see how the extended Acid Rain Theorem in action, let's consider the fusion of the following program which is more complicated than the program *zmm* in Section 3.2 in that *zip* not only consumes its arguments but also produces a data structure which will be consumed by other function.

$$mzmm\ x\ y = map\ plus\ (zip\ (map\ double\ x)\ (map\ square\ y))$$

We would like to show that after being standardized, the hylomorphism inducting over multiple data structures can be well fused with other hylomorphisms either from its left or from its right. For this purpose, first of all we need to express each recursion in a suitable hylomorphic form whose producing and consuming of data are standardized if possible. So, we define $zip'\,(x, y) = zip\, x\, y$ and thus have

$$\begin{array}{lcl}
zip' & : & (List\, A \times List\, B) \to List\,(A \times B) \\
zip' & = & [\![in_{F_{L_{A\times B}}},\ id,\ out_{F_{L_A}} \bowtie out_{F_{L_B}}]\!]_{F_{L_{A\times B}}, F_{L_{A\times B}}} \\
 & & \text{where} \\
 & & x \bowtie y = \text{case } (x,\ y) \text{ of} \\
 & & \qquad ((2,(a,as)),\ (2,(b,bs))) \to (2,((a,b),(as,bs))) \\
 & & \qquad \text{otherwise} \to (1,()).
\end{array}$$

We also have the following hylomorphisms for other recursions by the algorithm in [6] (Note we assume $plus : A \times B \to C$).

$$\begin{array}{lcl}
map\ plus & = & [\![in_{F_{L_C}},\ id + (plus \times id),\ out_{F_{L_{A\times B}}}]\!]_{F_{L_C}, F_{L_{A\times B}}} \\
map\ double & = & [\![in_{F_{L_A}},\ id + (double \times id),\ out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\
map\ square & = & [\![in_{F_{L_B}},\ id + (square \times id),\ out_{F_{L_B}}]\!]_{F_{L_B}, F_{L_B}}
\end{array}$$

We are ready to apply fusion transformation to *mzmm x y* based on the Extended Acid Rain Theorem, as shown in the following calculation.

$$\begin{array}{cl}
 & mzmm\ x\ y \\
= & \quad \{\text{ Definition of } mzmm\ \} \\
 & map\ plus\ (zip\ (map\ double\ x)\ (map\ square\ y)) \\
= & \quad \{\text{ Definition of } zip' \text{ and } \times\ \} \\
 & (map\ plus \circ zip' \circ (map\ double \times map\ square))\ (x, y)
\end{array}$$

Now we concentrate on the transformation of the front function part.

$$\begin{array}{cl}
 & map\ plus \circ zip' \circ (map\ double \times map\ square) \\
= & \quad \{\text{ Replacing each function with its hylo }\} \\
 & [\![in_{F_{L_C}},\ id + (plus \times id),\ out_{F_{L_{A\times B}}}]\!]_{F_{L_C}, F_{L_{A\times B}}} \circ \\
 & [\![in_{F_{L_{A\times B}}},\ id,\ out_{F_{L_A}} \bowtie out_{F_{L_B}}]\!]_{F_{L_{A\times B}}, F_{L_{A\times B}}} \circ \\
 & ([\![in_{F_{L_A}},\ id + (double \times id),\ out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \times \\
 & \quad [\![in_{F_{L_B}},\ id + (square \times id),\ out_{F_{L_B}}]\!]_{F_{L_B}, F_{L_B}}) \\
= & \quad \{\text{ Extended Acid Rain Theorem }\} \\
 & [\![in_{F_{L_C}},\ id + (plus \times id),\ out_{F_{L_{A\times B}}}]\!]_{F_{L_C}, F_{L_{A\times B}}} \circ \\
 & [\![in_{F_{L_{A\times B}}},\ id,\ ((id + (double \times id)) \circ out_{F_{L_A}}) \bowtie \\
 & \qquad\qquad ((id + (square \times id)) \circ out_{F_{L_B}})]\!]_{F_{L_{A\times B}}, F_{L_{A\times B}}} \\
= & \quad \{\text{ Acid Rain Theorem }\} \\
 & [\![in_{F_{L_C}},\ id + (plus \times id),\ ((id + (double \times id)) \circ out_{F_{L_A}}) \bowtie \\
 & \qquad\qquad\qquad ((id + (square \times id)) \circ out_{F_{L_B}})]\!]_{F_{L_C}, F_{L_{A\times B}}} \\
= & \quad \{\text{ Simple transformation }\} \\
 & [\![in_{F_{L_{A\times B}}},\ id + (plus \circ (double \times square) \times id),\ out_{F_{L_A}} \bowtie out_{F_{L_B}}]\!]_{F_{L_{A\times B}}, F_{L_{A\times B}}}
\end{array}$$

By inlining the definition of hylomorphism and simplifying, we get the following efficient version for *mzmm*.

$$\begin{array}{lcl}
mzmm\ x\ y & = & hylo\ (x, y) \\
\textbf{where} & & \\
hylo\ (x, y) & = & \text{case } (out_{F_{L_A}}\ x,\ out_{F_{L_B}}\ y) \text{ of} \\
 & & \quad ((2, (a, as)),\ (2, (b, bs))) \\
 & & \qquad \rightarrow Cons\ (plus\ (double\ a,\ square\ b),\ hylo\ (as, bs)) \\
 & & \quad \text{otherwise} \\
 & & \qquad \rightarrow Nil
\end{array}$$

We've successfully eliminated all data structures in the original programs, which is impossible if we merely use the original Acid Rain Theorem. It is also interesting to see that the final hylomorphism is in a good form to be fused with other hylomorphisms.

5. Related Work

Compositional style of functional programming provides a modular way for writing clear programs, but it comes at the price of unnecessary computations for producing and consuming some intermediate data structures passed between each separated modules. Program fusion is a process to merge the composition of modules into one by eliminating these intermediate data structures. It has been shown that such fusion is especially successful if each module is structured by some specific generic recursive patterns such as hylomorphisms. However, most studies are only devoted to the manipulation of recursions inducting over a single recursive structure.

The importance of manipulation of recursions over multiple data structures has been recognized in [2], where an extended catamorphic form and corresponding fusion theorems are proposed. A more general study can be found in [5, 6] which made it clear that this extended catamorphic form is nothing more than a special case of hylomorphism which is the most generic recursive form suitable to be fused with other functions. However, the above two approaches are more or less impractical to be embedded in a real compiler of a functional language; the former imposes too much restriction on the programs (i.e., potential normalizable program) so that the fusion transformation can be fully automatic while the later, though quite general, has difficulty in applying the fusion transformation automatically because it needs to find a new function satisfying the fusible condition in the General Hylo Fusion Theorem. It is the Acid Rain Theorem (a generation of the *fold/build* rule [4]) that makes the fusion transformation more practical in the sense that the rule application proceeds as a simple substitution. To this end, the consumption and the production of data structures are standardized in a specific way. However the Acid Rain Theorem has its limitation. In [4, 7], it was pointed out that one of the most serious problems with the Acid Rain Theorem is that it cannot be effectively applied to recursions inducting over multiple data structures such as *zip* and *take*. It was said that although considering *zip* as a list producer is straightforward it cannot use *foldr/build* rule so that *both* input lists to *zip* may be removed. Although an attempt was made in [10] to solve this problem, there is something unclear in their discussion which in fact leaves the problem open.

We came across this problem when we are implementing our HYLO Calculator, a system for optimizing functional programs based on transformation over hylomorphisms. We are surprised to find that among other well-know functions such as *zip* and *take*, many functions in the standard benchmark programs in fact induct over multiple data structure, which motivated us to extend the Acid Rain Theorem as discussed in this paper.

We are now working on the extension of our algorithm [6] to transform the user-defined programs into the form required by the Extended Acid Rain Theorem. We believe that such algorithm will not be so difficult to give.

6. Concluding Remarks

In this paper, we show how the consumption of multiple data structures can be naturally captured by a combination of three polymorphic functions, based on which we extend the Acid Rain Theorem so that it can handle hylomorphisms over multiple data structures. Our extended Acid Rain Theorem can be successfully applied to eliminate more intermediate data structures in functional programs with recursions over multiple data structures.

Acknowledgements

We would like to thank Yoshiyuki Onoue and the anonymous referees who provided many detailed and useful comments for improvements on earlier versions of this paper.

References

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

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

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

[4] 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.

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

[6] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA, May 1996. ACM Press.

[7] 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.

[8] 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, Massachuetts, August 1991.

[9] 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.

[10] 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.

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

[12] P. Wadler. Theorems for free. In *Proc. Conference on Functional Programming and Computer Architecture*, pages 347–359, 1989.