TOWARDS MANIPULATION OF MUTUALLY RECURSIVE FUNCTIONS

HIDEYA IWASAKI
Department of Computer Science
Tokyo University of Agriculture and Technology
2–24–16, Naka-cho, Koganei, Tokyo 184–8588, JAPAN
E-mail: iwasaki@ipl.ei.tuat.ac.jp

ZHENJIANG HU, MASATO TAKEICHI
Department of Information Engineering
The University of Tokyo
7–3–1, Hongo, Bunkyo-ku, Tokyo 113–8656, JAPAN
E-mail: {hu,takeichi}@ipl.t.u-tokyo.ac.jp

In functional programming, Constructive Algorithmics is one of the promising approaches to program transformation, especially to fusion, in which a concept called hylomorphism plays quite an important role. However, previous studies have mainly focused on programs constructed by single recursive functions, whereas programs constructed by mutual recursion have been little investigated, particularly with respect to their practical use for optimizing functional programs. In this paper, we shall formalize mutual recursive data types in terms of bifunctors and extend hylomorphisms to describe mutual recursive functions on this type. It is shown that theorems (transformation rules) for original hylomorphisms also hold while keeping the same form for extended (mutual) hylomorphisms. As a result, program transformation by calculation, which mechanically applies these theorems, is also possible without adding any special rules for mutual recursion.

1 Introduction

In functional programming, small but useful functions are composed together to constitute a large program. Unfortunately, programs described in this fashion cannot be expected to be efficient both in time and space, mainly because many intermediate data structures which never appear in the final result may be produced and consumed through functional compositions. Therefore it is important to transform programs to eliminate such data structures.

One of the promising approaches to solve this problem is by means of *Constructive Algorithmics* [6,7], in which a concept called *hylomorphism* plays an important role. In this approach, a small but powerful set of transformation rules (theorems) over hylomorphisms are applied to the target program to fuse functional compositions. This kind of transformation is called *fusion*. An important point is that once (a part of) the target program is expressed in some specific form such as hylomorphism, the transformation process is nothing more than an automated calculation.

In Constructive Algorithmics, data types are categorically defined by func-

tors [1,6,7,11] which uniformly capture their recursive structures, and functions with some specific form such as *catamorphism* (fold), *anamorphism* (unfold) and *hylomorphism* (generalized concept including both catamorphism and anamorphism) have been the main targets of program transformation.

Recent studies [2,3,5,10,11] have shown that fusion transformation by Constructive Algorithmics is effective for programs constructed by single recursive functions (defined over single recursive data types). On mutual recursion, however, few studies [1,6] have been done, particularly on practical aspects for fusion. Sheard and Fegaras [10] discussed mutual recursion, but restricted to catamorphisms. Since transformation rules proposed so far are unsatisfactory for the fusion (for example, *shortcut deforestation* [2,11]) of mutual recursive functions, the application area of Constructive Algorithmics is limited. Figure 1(a) shows an example of mutual recursive types T_1 and T_2 with data constructors $\mathtt{C}_{ij}$. Functions defined over them cannot be captured in normal hylomorphisms because of their mutual dependency.

The purpose of this paper is to extend the concept of hylomorphism so that Constructive Algorithmics can be applied to the transformation of programs including mutual recursions. We focus mainly upon the practical application of Constructive Algorithmics, aiming at a more systematic approach for the fusion of mutual recursive functions. Our main contributions on this work can be summerized as follows.

- We propose an extension of hylomorphisms based on the concept of *bifunctor*s (functors with two arguments), so that mutual recursive data types and functions can be successfully formalized.

- By extending hylomorphism on mutual recursive types as a product of mutual recursive functions, it is shown that useful theorems (Hylo Fusion and Acid Rain theorems [11]) for normal hylomorphisms are also valid in the same form for mutual hylomorphisms. Therefore, fusion transformation by calculation, which mechanically applies these theorems, is also possible without adding any special rules for mutual recursive case.

- While previous discussions to deal with mutual recursion have been devoted to catamorphism, our work also focuses on its dual (anamorphic) aspect. This work is, therefore, more practical than previous work in that ours can deal with both catamorphism and anamorphism with respect to mutual recursive types. We give an example of fusion transformation (Section 5.2), which would be impossible by manipulation of catamorphism alone.

- Our idea can be specialized to deal with a product of single recursive functions, which helps to eliminate multiple data structures that are simulatneously traversed. We clearly demonstrate the relationship between

$$T_1\ a = \mathtt{C}_{11}\ a \mid \mathtt{C}_{12}\ (T_1\ a,\ T_2\ a)$$
$$T_2\ a = \mathtt{C}_{21}\ a \mid \mathtt{C}_{22}\ (T_2\ a,\ T_1\ a)$$

(a) Example A: Mutual Recursive Types

$$S_1\ a = \mathtt{D}_{11}\ a \mid \mathtt{D}_{12}\ (a,\ S_1\ a)$$
$$S_2\ a = \mathtt{D}_{21} \mid \mathtt{D}_{22}\ (a,\ a,\ S_2\ a)$$

(b) Example B: Two Single Recursive Types

Figure 1: Examples of type definitions

mutual and normal hylomorphisms by *Mutualizing Theorem* for this special case.

In this paper, we often use the terms *mutual* and *normal* (such as *mutual hylomorphism* and *normal case*) to distinguish between mutual and single recursions. In order to explain our idea, data types in Figure 1 are often referred to throughout this paper.

This paper is organized as follows. Section 2 gives an overview of Constructive Algorithmics for the normal (single recursive) case, as an aid for understanding our work. In Section 3, we give a theoretical basis of the extension for mutual recursive case, together with mutual hylomorphism and related theorems. In Section 4, we consider tupling of two normal hylomorphisms and show that it can be treated within our formalization. Some concrete examples, both for mutual case and tupling, are given in Section 5. Section 6 discusses some related work and Section 7 gives conclusion.

2 Preliminary

Before introducing bifunctor and mutual hylomorphism, we briefly review the formalization of normal hylomorphism. More detailed discussion can be found in [1,6,7,11].

In this paper, our default category $\mathcal{C}$ is CPO. In the normal case, functors (known as *polynomial functor*s) from $\mathcal{C}$ to $\mathcal{C}$ are constructed by the following four operations. Here, X and Y denote types, p and q denote functions, and id represents the polymorphic identity function.

1. (Constant) $!a\ X = a$, $!a\ p = id$
2. (Identity) $I\ X = X$, $I\ p = p$
3. (Product) $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$
   $(p \times q)\ (x, y) = (p\ x,\ q\ y)$

4. (Separated Sum) $X + Y = \{1\} \times X \cup \{2\} \times Y$
$(p + q)\ (1, x) = (1,\ p\ x),\ (p + q)\ (2, x) = (2,\ q\ x)$
$(p \triangledown q)\ (1, x) = p\ x,\ (p \triangledown q)\ (2, x) = q\ x$

By overloading, the product ($\times$) and separated sum ($+$) on categories also denote those on functors.

$$(F \times G)\ X = F\ X \times G\ X, \quad (F \times G)\ p = F\ p \times G\ p$$
$$(F + G)\ X = F\ X + G\ X, \quad (F + G)\ p = F\ p + G\ p$$

Polynomial functors characterize single recursive structures. As a simple example, a type definition of the list structure whose elements have type a is given below.

$$List\ a = \texttt{Nil} \mid \texttt{Cons}\ (a,\ List\ a)$$

It has two data constructors: `Nil` representing the empty list, and `Cons` being a cons cell with two fields. Polynomial functor L_a which corresponds to this type definition is:

$$L_a = !\mathbf{1} + !\,a \times I$$

where $\mathbf{1}$ stands for the terminal object in $\mathcal{C}$. Hereafter we will omit the subscript a since it should be clear from the context. It may be better to write `Nil` () other than `Nil` because it has no argument, but we interpret `Nil` as `Nil` ().

Definition 1 (F–Algebra and F–homomorphism) F–Algebra is a pair (X, ϕ), where X is an object in $\mathcal{C}$ called *carrier*, and $\phi :: F\ X \to X$ is a morphism. F–homomorphism from (X, ϕ) to (U, ψ) is a morphism $h :: X \to U$ satisfying $h \circ \phi = \psi \circ F\ h$. ∎

Definition 2 (Category of F–Algebras) The category of F–Algebras has F–Algebras as its objects and F–homomorphisms as its arrows. Composition and identities are taken from $\mathcal{C}$. ∎

Let (T, in_F) be the initial object (initial algebra) in this category, which is known to exist if F is a polynomial functor. This initial algebra defines the data type T equipped with a data constructor denoted by $in_F :: F\ T \to T$. The inverse of in_F is a data destructor, which is denoted by $out_F :: T \to F\ T$.

For the above example of the list structure, the initial object of the category of L–algebras is $(List\ a,\ in_L)$ with its data constructor $in_L = \texttt{Nil} \triangledown \texttt{Cons}$. Data destructor out_L is defined as follows.

$$\begin{aligned} out_L = \lambda xs\ .\ \texttt{case}\ xs\ \texttt{of}\ \texttt{Nil} &\to (1, ()) \\ \texttt{Cons}\ (a,\ as) &\to (2, (a,\ as)) \end{aligned}$$

Upon this theoretical basis, functions with a specific form called *hylomorphism*s are defined.

Definition 3 (Hylomorphism) Given $\phi :: G\ B \to B$, $\psi :: A \to F\ A$ and natural transformation $\eta :: F \dot{\to} G$, hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ is defined as the least $f :: A \to B$ satisfying the equation of

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

We write this least fixed point by the notation $\mu(\lambda f\ .\ \phi \circ (\eta \circ F\ f) \circ \psi)$. ∎

Catamorphism and anamorphism are special cases of hylomorphism.

$$(\!| \phi |\!)_F = [\![\phi, id, out_F]\!]_{F,F}$$
$$[\!(\psi)\!]_F = [\![in_F, id, \psi]\!]_{F,F}$$

As for hylomorphisms, we have the following theorems.

Theorem 4 (Hylo Shift) Let $\eta :: F \dot{\to} G$ be a natural transformation. Then

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

This theorem guarantees that the natural transformation in the middle of a hylomorphism can be freely shifted within it. The next theorem states that under some condition, any function composed with a hylomorphism can be promoted to another hylomorphism.

Theorem 5 (Hylo Fusion)

Left Fusion: $f \circ \phi = \phi' \circ G\ f \implies f \circ [\![\phi, \eta, \psi]\!]_{G,F} = [\![\phi', \eta, \psi]\!]_{G,F}$

Right Fusion: $\psi \circ g = F\ g \circ \psi' \implies [\![\phi, \eta, \psi]\!]_{G,F} \circ g = [\![\phi, \eta, \psi']\!]_{G,F}$ ∎

When the composed functions are represented in some restricted form of hylomorphisms, they can be fused only by a simple substitution based on the following Acid Rain Theorem.

Theorem 6 (Acid Rain)

Cata–Hylo Fusion:

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

Hylo–Ana Fusion:

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

The latter two theorems are quite significant because they provide basic rewriting rules for fusion transformation which eliminate functional compositions. It is worth noting that once the preconditions of each theorem are satisfied, the application of the theorem to the target program can be automated.

3 Formalization

Our formalization for mutual recursive data types and functions follows a similar approach to that for normal case explained in the previous section.

3.1 F*–algebra*

Without loss of generality, we shall consider the case where *two* recursive types are mutually defined. For this case, functors with two arguments (*bifunctor*s) are used in our formalization.

Definition 7 (Bifunctor) Bifunctor F takes a pair of two objects, X and Y, into $\mathsf{F}\ (X, Y)$ and a pair of two functions, p and q ($p :: A \to D,\ q :: B \to E$), into $\mathsf{F}\ (p, q) :: \mathsf{F}\ (A, B) \to \mathsf{F}\ (D, E)$, preserving identity and composition.

$$\mathsf{F}\ (id,\ id) = id$$
$$\mathsf{F}\ (p, q) \circ \mathsf{F}\ (r, s) = \mathsf{F}\ (p \circ r,\ q \circ s) \qquad \blacksquare$$

In this paper, we assume that bifunctors are constructed by the following four basic operations, where X and Y denote types, p and q denote functions.

1. (Constant) $!a\ (X, Y) = a,\ !a\ (p, q) = id$
2. (Projection) $\Pi_1\ (X, Y) = X,\ \Pi_1\ (p, q) = p$
   $\Pi_2\ (X, Y) = Y,\ \Pi_2\ (p, q) = q$
3. (Product) Same as that of polynomial functor.
4. (Separated Sum) Same as that of polynomial functor.

It should be noted that the identity functor in defining polynomial functors is replaced with two projection functors. Strictly speaking, the constant functor in this definition is slightly different from that in the definition of polynomial functor in that it takes two arguments. But we use the same notation, since no confusion could be caused.

In order to formalize mutually recursive types and functions, we consider bifunctors of the type $\mathcal{C} \times \mathcal{C} \to \mathcal{C} \times \mathcal{C}$, where $\mathcal{C}$ denotes CPO. Since $\mathcal{C} \times \mathcal{C}$ preserves the properties of $\mathcal{C}$, the discussion in the previous section is also valid in our formalization. To avoid confusion, we express bifunctors of this type in sans serif font such as F or G, while we express normal ones in math-italic font such as F or G. In addition, we shall let $(_, _)$ denote a product of two CPO's. Operationally speaking, $(p, q)\ (x, y) = (p\ x,\ q\ y)$. Like the case of $\times$, this notation is overloaded to denote a binary operator on functors.

For example, types T_1 and T_2 in Example A (Figure 1) can be formalized by defining the following functor F.

$$\text{For Objects:}\quad \mathsf{F}\ (X, Y) = (a + X \times Y,\ a + Y \times X)$$
$$\text{For Functions:}\quad \mathsf{F}\ (p, q) = (id + p \times q,\ id + q \times p)$$

For Example B, types of S_1 and S_2 can be formalized by the following bifunctor G.

$$\text{For Objects:}\quad \mathsf{G}\ (X, Y) = (a + a \times X,\ \mathbf{1} + a \times a \times Y)$$
$$\text{For Functions:}\quad \mathsf{G}\ (p, q) = (id + id \times p,\ id + id \times id \times q)$$

These functors are expressed simply as:

$$\mathsf{F} = (!a + \Pi_1 \times \Pi_2,\ !a + \Pi_2 \times \Pi_1)$$
$$\mathsf{G} = (!a + !a \times \Pi_1,\ !\mathbf{1} + !a \times !a \times \Pi_2).$$

Similar to the normal case, natural transformation is defined as follows.

Definition 8 (Natural Transformation) Let F and G be two bifunctors and $\eta :: \mathsf{F}\ (X, Y) \to \mathsf{G}\ (X, Y)$ be a polymorphic function. If η has the property of

$$\forall p, q : \quad \eta \circ \mathsf{F}\ (p, q) = \mathsf{G}\ (p, q) \circ \eta,$$

then such η is called a *natural transformation*, and written as $\eta :: \mathsf{F} \dot{\to} \mathsf{G}$. ∎

Definition 9 (F–algebra and F–homomorphism) F–algebra is a pair $((X, Y), \phi)$, where (X, Y) is an object in $\mathcal{C} \times \mathcal{C}$ called a carrier, and $\phi :: \mathsf{F}\ (X, Y) \to (X, Y)$ is a morphism. F–homomorphism from $((X, Y), \phi)$ to $((U, V), \xi)$ is a pair $(f, g) :: (X, Y) \to (U, V)$ which satisfies the following equation.

$$(f, g) \circ \phi = \xi \circ \mathsf{F}\ (f, g)$$ ∎

Definition 10 (Category of F–algebras) Consider a category whose objects are F–algebras and whose arrows are F–homomorphisms, with identity and composition taken from $\mathcal{C} \times \mathcal{C}$. It has the initial algebra $((T_1, T_2), in_{\mathsf{F}})$, where T_1 and T_2 are types determined by F, and in_{F}, data constructor, has the type of:

$$in_{\mathsf{F}} :: \mathsf{F}\ (T_1, T_2) \to (T_1, T_2).$$

The inverse of in_{F}, data destructor denoted as out_{F}, has the following type.

$$out_{\mathsf{F}} :: (T_1, T_2) \to \mathsf{F}\ (T_1, T_2)$$ ∎

For instance, in_{F} and out_{F} for Example A are as follows.

$$\begin{aligned}
&in_{\mathsf{F}} = (\mathsf{C}_{11} \triangledown \mathsf{C}_{12},\ \mathsf{C}_{21} \triangledown \mathsf{C}_{22})\\
&out_{\mathsf{F}} = (o_1, o_2) \quad \texttt{where}\\
&\quad o_1\ x = \texttt{case}\ x\ \texttt{of}\ \mathsf{C}_{11}\ x \to (1, x)\\
&\qquad\qquad\qquad\ \ \mathsf{C}_{12}\ (x, y) \to (2,\ (x, y))\\
&\quad o_2\ x = \texttt{case}\ x\ \texttt{of}\ \mathsf{C}_{21}\ x \to (1, x)\\
&\qquad\qquad\qquad\ \ \mathsf{C}_{22}\ (x, y) \to (2,\ (x, y))
\end{aligned}$$

3.2 Mutual Hylomorphism

Definition 11 (Mutual Hylomorphism) When $\phi :: \mathsf{G}\ (D, E) \to (D, E)$, $\psi :: (A, B) \to \mathsf{F}\ (A, B)$ and natural transformation $\eta :: \mathsf{F} \dot{\to} \mathsf{G}$ are given, hylomorphism $[\![\phi, \eta, \psi]\!]_{\mathsf{G},\mathsf{F}}$ with respect to bifunctors F and G is defined as the least (f, g) satisfying

$$(f, g) = \phi \circ (\eta \circ \mathsf{F}\ (f, g)) \circ \psi.$$

$$
\begin{aligned}
& f_1\ (\mathtt{C}_{11}\ x) = \phi_{11}\ x \\
& f_1\ (\mathtt{C}_{12}\ (x, y)) = \phi_{12}\ (f_1\ x,\ f_2\ y) \\
& f_2\ (\mathtt{C}_{21}\ x) = \phi_{21}\ x \\
& f_2\ (\mathtt{C}_{22}\ (x, y)) = \phi_{22}\ (f_2\ x,\ f_1\ y) \\
& (f_1, f_2) = (\![\ (\phi_1, \phi_2)\ ]\!)_{\mathsf{F}} \\
& \qquad \mathtt{where}\ \phi_1 = \phi_{11} \triangledown \phi_{12} \text{ and } \phi_2 = \phi_{21} \triangledown \phi_{22}
\end{aligned}
$$

(a) catamorphism

$$
\begin{aligned}
& k_1\ x = \mathtt{if}\ E_0\ \mathtt{then}\ \mathtt{C}_{11}\ E_1\ \mathtt{else}\ \mathtt{C}_{12}\ (k_1\ E_2,\ k_2\ E_3) \\
& k_2\ x = \mathtt{if}\ E_4\ \mathtt{then}\ \mathtt{C}_{21}\ E_5\ \mathtt{else}\ \mathtt{C}_{22}\ (k_2\ E_6,\ k_1\ E_7) \\
& \psi_1\ x = \mathtt{if}\ E_0\ \mathtt{then}\ (\mathtt{1}, E_1)\ \mathtt{else}\ (\mathtt{2}, (E_2, E_3)) \\
& \psi_2\ x = \mathtt{if}\ E_4\ \mathtt{then}\ (\mathtt{1}, E_5)\ \mathtt{else}\ (\mathtt{2}, (E_6, E_7)) \\
& (k_1, k_2) = [\!(\ (\psi_1, \psi_2)\ )\!]_{\mathsf{F}} \\
& \qquad \mathtt{where}\ E_0 \sim E_7 \text{ denote some expressions}
\end{aligned}
$$

(b) anamorphism

Figure 2: Definitions of mutual catamorphism and anamorphism

That is,

$$[\![\ \phi, \eta, \psi\ ]\!]_{\mathsf{G},\mathsf{F}} = \mu(\lambda(f, g)\ .\ \phi \circ (\eta \circ \mathsf{F}\ (f, g)) \circ \psi) \qquad \blacksquare$$

Using a mutual hylomorphism, mutual catamorphism and anamorphism are defined in the same manner as the normal case.

$$
\begin{aligned}
(\![\ \phi\ ]\!)_{\mathsf{F}} &= [\![\ \phi, id, out_{\mathsf{F}}\ ]\!]_{\mathsf{F},\mathsf{F}} \\
[\!(\ \psi\ )\!]_{\mathsf{F}} &= [\![\ in_{\mathsf{F}}, id, \psi\ ]\!]_{\mathsf{F},\mathsf{F}}
\end{aligned}
$$

They are characterized by the following equations.

$$
\begin{aligned}
(\![\ \phi\ ]\!)_{\mathsf{F}} \circ in_{\mathsf{F}} &= \phi \circ \mathsf{F}\ (\![\ \phi\ ]\!)_{\mathsf{F}} \\
out_{\mathsf{F}} \circ [\!(\ \psi\ )\!]_{\mathsf{F}} &= \mathsf{F}\ [\!(\ \psi\ )\!]_{\mathsf{F}} \circ \psi
\end{aligned}
$$

Note that mutual versions of catamorphism and anamorphism also represent the products of functions.

To see the recursive forms of catamorphism and anamorphism defined in this fashion, we return to our example A. Figure 2 shows definitions of mutual catamorphism and anamorphism defined over mutual recursive types T_1 and T_2 determined by bifunctor F.

3.3 Transformation Theorems

Based on our formalization discussed above, transformation rules (theorems) described in Section 2 can be extended to the mutual case preserving the same

form.

Theorem 12 (Mutual Hylo Shift)

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

Proof: Obvious from the definitions of hylomorphism and natural transformation. ∎

Theorem 13 (Mutual Hylo Fusion)

Left Fusion:

$$(f,g) \circ \phi = \phi' \circ \mathsf{G}\ (f,g) \implies (f,g) \circ [\![\phi, \eta, \psi]\!]_{\mathsf{G},\mathsf{F}} = [\![\phi', \eta, \psi]\!]_{\mathsf{G},\mathsf{F}}$$

Right Fusion:

$$\psi \circ (f,g) = \mathsf{F}\ (f,g) \circ \psi' \implies [\![\phi, \eta, \psi]\!]_{\mathsf{G},\mathsf{F}} \circ (f,g) = [\![\phi, \eta, \psi']\!]_{\mathsf{G},\mathsf{F}}$$

Proof: The proof is the same as that of Hylo Fusion for the normal case. ∎

Theorem 14 (Mutual Acid Rain)

MutualCata–MutualHylo Fusion:

$$\tau :: \forall A\ .\ (\mathsf{F}\ A \to A) \to \mathsf{F}'\ A \to A \implies$$
$$[\![\phi, \eta_1, out_{\mathsf{F}}]\!]_{\mathsf{G},\mathsf{F}} \circ [\![\tau\ in_{\mathsf{F}},\ \eta_2,\ \psi]\!]_{\mathsf{F}',\mathsf{M}} = [\![\tau\ (\phi \circ \eta_1),\ \eta_2,\ \psi]\!]_{\mathsf{F}',\mathsf{M}}$$

MutualHylo–MutualAna Fusion:

$$\sigma :: \forall A\ .\ (A \to \mathsf{F}\ A) \to A \to \mathsf{F}'\ A \implies$$
$$[\![\phi,\ \eta_1,\ \sigma\ out_{\mathsf{F}}]\!]_{\mathsf{G},\mathsf{F}'} \circ [\![in_{\mathsf{F}}, \eta_2, \psi]\!]_{\mathsf{F},\mathsf{M}} = [\![\phi,\ \eta_1,\ \sigma\ (\eta_2 \circ \psi)]\!]_{\mathsf{G},\mathsf{F}'}$$

MutualCata–NormalHylo Fusion:

$$\tau :: \forall A\ .\ (\mathsf{F}\ A \to A) \to F'\ A \to A \implies$$
$$[\![\phi, \eta_1, out_{\mathsf{F}}]\!]_{\mathsf{G},\mathsf{F}} \circ [\![\tau\ in_{\mathsf{F}},\ \eta_2,\ \psi]\!]_{F',M} = [\![\tau\ (\phi \circ \eta_1),\ \eta_2,\ \psi]\!]_{F',M}$$

NormalHylo–MutualAna Fusion:

$$\sigma :: \forall A\ .\ (A \to \mathsf{F}\ A) \to A \to F'\ A \implies$$
$$[\![\phi,\ \eta_1,\ \sigma\ out_{\mathsf{F}}]\!]_{G,F'} \circ [\![in_{\mathsf{F}}, \eta_2, \psi]\!]_{\mathsf{F},\mathsf{M}} = [\![\phi,\ \eta_1,\ \sigma\ (\eta_2 \circ \psi)]\!]_{G,F'}$$

Proof: The proof of this theorem is similar to that of normal Acid Rain Theorem [11,12]. ∎

Note that NormalCata–MutualHylo and MutualHylo–NormalAna Fusions are invalid because of type inconsistency. In Sections 5.2 and 5.3, we will see how this theorem works for fusion transformation.

4 Mutualization of Two Single Recursions

In this section, we would like to focus on formalizing a product of single recursive functions within the framework of mutual recursive case. It is useful for eliminating multiple data structures that are simultaneously traversed (Section 5.2).

Polynomial functors G_1 and G_2 associated with S_1 and S_2 in Example B (Figure 1) are:

$$G_1 = !a + !a \times I$$
$$G_2 = !\mathbf{1} + !a \times !a \times I$$

As was stated in the previous section, regarding the set of S_1 and S_2 as a special case of mutual recursive data types gives the following bifunctor G.

$$\begin{array}{lll} \text{For Objects:} & \mathsf{G}\,(X,Y) & = (a + a \times X,\ \mathbf{1} + a \times a \times Y) \\ & & = (G_1\ X,\ G_2\ Y) \\ \text{For Functions:} & \mathsf{G}\,(p,q) & = (id + id \times p,\ id + id \times id \times q) \\ & & = (G_1\ p,\ G_2\ q) \end{array}$$

Definition 15 (Bifunctor Constructing Operator $\otimes$) Let F_1 and F_2 be two polynomial functors. Binary operator $\otimes$ is defined as:

$$\mathsf{F} = F_1 \otimes F_2 \quad \Longleftrightarrow \quad \begin{array}{l} \mathsf{F}\,(X,Y) = (F_1\ X,\ F_2\ Y) \\ \mathsf{F}\,(p,q) = (F_1\ p,\ F_2\ q). \end{array}$$

where F denotes the bifunctor constructed by $\otimes$. ∎

For instance, G in Example B is expressed as $\mathsf{G} = G_1 \otimes G_2$. Then the following relationships about *in*'s and *out*'s of G_1, G_2 and G are satisfied.

$$(in_{G_1},\ in_{G_2}) = in_{\mathsf{G}}$$
$$(out_{G_1},\ out_{G_2}) = out_{\mathsf{G}}$$

In addition, suppose that catamorphisms g_1 on S_1 and g_2 on S_2 are defined as:

$$\begin{array}{l} g_1\ (\mathsf{D}_{11}\ x) = \xi_{11}\ x \\ g_1\ (\mathsf{D}_{12}\ (x, xs)) = \xi_{12}\ (x,\ g_1\ xs) \\ g_2\ \mathsf{D}_{21} = \xi_{21}\ () \\ g_2\ (\mathsf{D}_{22}\ (x, y, xs)) = \xi_{22}\ (x,\ y,\ g_2\ xs), \end{array}$$

that is, $g_1 = (\!|\, \xi_1 \,|\!)_{G_1}$ and $g_2 = (\!|\, \xi_2 \,|\!)_{G_2}$ where $\xi_1 = \xi_{11} \triangledown \xi_{12}$ and $\xi_2 = \xi_{21} \triangledown \xi_{22}$. Then (g_1, g_2) turns out to be a mutual catamorphism $(\!|\, (\xi_1, \xi_2)\, |\!)_{\mathsf{G}}$ satisfying

$$((\!|\, \xi_1 \,|\!)_{G_1},\ (\!|\, \xi_2 \,|\!)_{G_2}) = (\!|\, (\xi_1, \xi_2)\, |\!)_{\mathsf{G}}.$$

This example suggests that it would be possible to represent the product of two single recursive functions as a mutual recursion. In fact, this relationship on catamorphisms can be extended to hylomorphisms, which is summarized as the following theorem.

Theorem 16 (Mutualzing) Having normal hylomorphisms $[\![\, \phi_1, \eta_1, \psi_1 \,]\!]_{G_1,F_1}$ and $[\![\, \phi_2, \eta_2, \psi_2 \,]\!]_{G_2,F_2}$, we have the following equations:

$$\begin{array}{l} ([\![\, \phi_1, \eta_1, \psi_1 \,]\!]_{G_1,F_1},\ [\![\, \phi_2, \eta_2, \psi_2 \,]\!]_{G_2,F_2}) = [\![\, (\phi_1, \phi_2),\ (\eta_1, \eta_2),\ (\psi_1, \psi_2) \,]\!]_{\mathsf{G},\mathsf{F}} \\ (in_{F_1},\ in_{F_2}) = in_{\mathsf{F}},\ (in_{G_1},\ in_{G_2}) = in_{\mathsf{G}} \\ (out_{F_1},\ out_{F_2}) = out_{\mathsf{F}},\ (out_{G_1},\ out_{G_2}) = out_{\mathsf{G}} \end{array}$$

where F and G stand for bifunctors defined as:

$$\mathsf{F} = F_1 \otimes F_2, \quad \mathsf{G} = G_1 \otimes G_2.$$

Proof Outline: Let $P(f)$, $Q(g)$ and $R(f,g)$ be

$$\begin{aligned} P(f) &= \phi_1 \circ (\eta_1 \circ F_1\ f) \circ \psi_1 \\ Q(g) &= \phi_2 \circ (\eta_2 \circ F_2\ g) \circ \psi_2 \\ R(f,g) &= (\phi_1, \phi_2) \circ ((\eta_1, \eta_2) \circ \mathsf{F}\ (f,g)) \circ (\psi_1, \psi_2). \end{aligned}$$

Since $\mathsf{F} = F_1 \otimes F_2$, we have

$$\begin{aligned} (P(f),\ Q(g)) &= (\phi_1, \phi_2) \circ (\eta_1, \eta_2) \circ (F_1\ f,\ F_2\ g) \circ (\psi_1, \psi_2) \\ &= R(f,g). \end{aligned}$$

Applying the least fixed point induction immediately leads to the following equation,

$$(\mu(\lambda f\ .\ P(f)),\ \mu(\lambda g\ .\ Q(g))) = \mu(\lambda(f,g)\ .\ R(f,g))$$

so the first equation is proved. Considering $\mathsf{G} = G_1 \otimes G_2$ and $\mathsf{F} = F_1 \otimes F_2$, the relationships among *in*'s are easy to see. Since *out* is the inverse of *in*, the relationships among *out*'s are also obvious. ∎

5 Examples

In this section, we will give three examples to illustrate the fusion transformation based on our approach.

5.1 Example of Parse Trees

Consider the following types defining parse trees for a simple functional language.

$$\begin{aligned} \mathtt{Exp}\ (a,b) = &\ \mathtt{Val}\ b \\ \mid &\ \mathtt{Ide}\ a \\ \mid &\ \mathtt{Let}\ (\mathtt{Dec}\ (a,b),\ \mathtt{Exp}\ (a,b)) \\ \mid &\ \mathtt{App}\ (\mathtt{Exp}\ (a,b),\ \mathtt{Exp}\ (a,b)) \\ \mathtt{Dec}\ (a,b) = &\ \mathtt{Var}\ (a,\ \mathtt{Exp}\ (a,b)) \\ \mid &\ \mathtt{Fun}\ (a,\ a,\ \mathtt{Exp}\ (a,b)) \end{aligned}$$

In this type definition, a is the type of locally defined variables or functions in a `Let` statement, and b is the "grounded" type of an expression. Corresponding to these mutually recursive types, bifunctor F is defined as:

$$\mathsf{F} = (!b + !a + \Pi_2 \times \Pi_1 + \Pi_1 \times \Pi_1,\ !a \times \Pi_1 + !a \times !a \times \Pi_1).$$

Now consider the function *nme* which accepts an expression of the type $\mathtt{Exp}\ (a,b)$ and returns the list of names which are locally defined within its `Let` statements. Then *nme*, together with the auxiliary function *nmd* on $\mathtt{Dec}\ (a,b)$, is inductively defined on the recursive structures of `Exp` and `Dec`, as shown in Figure 3(a). In this definition, $+\!\!+$ is the list concatenating function, and $[x]$ is the abbreviation of $\mathtt{Cons}(x, \mathtt{Nil})$. The target program for fusion transformation

$$
\begin{aligned}
&f = \mathit{length} \circ \mathit{nme}\\
&\mathit{nme}\,(\mathtt{Val}\ v) = \mathtt{Nil}\\
&\mathit{nme}\,(\mathtt{Ide}\ x) = \mathtt{Nil}\\
&\mathit{nme}\,(\mathtt{Let}\ (d, e)) = \mathit{nmd}\,d +\!\!+ \mathit{nme}\,e\\
&\mathit{nme}\,(\mathtt{Apply}\ (e_1, e_2)) = \mathit{nme}\,e_1 +\!\!+ \mathit{nme}\,e_2\\
&\mathit{nmd}\,(\mathtt{Var}\ (x, e)) = [x] +\!\!+ \mathit{nme}\,e\\
&\mathit{nmd}\,(\mathtt{Fun}\ (x, y, e)) = [x] +\!\!+ \mathit{nme}\,e
\end{aligned}
$$

(a) Original program

$$
\begin{aligned}
&f\ (\mathtt{Val}\ v) = 0\\
&f\ (\mathtt{Ide}\ x) = 0\\
&f\ (\mathtt{Let}\ (d, e)) = h\,d + f\,e\\
&f\ (\mathtt{Apply}\ (e_1, e_2)) = f\,e_1 + f\,e_2\\
&h\ (\mathtt{Var}\ (x, e)) = 1 + f\,e\\
&h\ (\mathtt{Fun}\ (x, y, e)) = 1 + f\,e
\end{aligned}
$$

(b) Final program

Figure 3: Program on a simple parse tree

is $f = \mathit{length} \circ \mathit{nme}$, where *length* is the standard function returning the length of a list.

First, we express (nme, nmd) as a mutual hylomorphism.

$$
\begin{aligned}
&(\mathit{nme},\ \mathit{nmd}) = [\![\,(\phi_e,\ \phi_d),\ \mathit{id},\ \mathit{out}_{\mathsf{F}}\,]\!]_{\mathsf{F},\mathsf{F}}\\
&\phi_e = \phi_{e1} \triangledown \phi_{e2} \triangledown \phi_{e3} \triangledown \phi_{e4}\\
&\quad \phi_{e1}\ v = \phi_{e2}\ x = \mathtt{Nil}\\
&\quad \phi_{e3}\ (p_1, p_2) = \phi_{e4}\ (p_1, p_2) = p_1 +\!\!+ p_2\\
&\phi_d = \phi_{d1} \triangledown \phi_{d2}\\
&\quad \phi_{d1}\ (x, p) = \phi_{d2}\ (x, y, p) = [x] +\!\!+ p
\end{aligned}
$$

Now that *nme* is successfully reexpressed in a mutual hylomorphism, we fuse the composition of $\mathit{length} \circ \mathit{nme}$. To apply the Mutual Hylo Fusion Theorem, we have to discover g, ϕ'_e and ϕ'_d satisfying the precondition of the theorem:

$$(\mathit{length},\ g) \circ (\phi_e,\ \phi_d) = (\phi'_e,\ \phi'_d) \circ \mathsf{F}\ (\mathit{length},\ g).$$

The above equation can be divided into two equations.

$$
\begin{aligned}
&\mathit{length} \circ \phi_e = \phi'_e \circ (\mathit{id} + \mathit{id} + g \times \mathit{length} + \mathit{length} \times \mathit{length})\\
&g \circ \phi_d = \phi'_d \circ (\mathit{id} \times \mathit{length} + \mathit{id} \times \mathit{id} \times \mathit{length})
\end{aligned}
$$

Omitting the detailed procedure of our derivation, we can discover that

$$
\begin{array}{l}
g = \mathit{length} \\
\phi'_e = \phi'_{e1} \triangledown \phi'_{e2} \triangledown \phi'_{e3} \triangledown \phi'_{e4} \\
\quad \phi'_{e1}\ v = \phi'_{e2}\ x = 0 \\
\quad \phi'_{e3}\ (m,n) = \phi'_{e4}\ (m,n) = m + n \\
\phi'_d = \phi'_{d1} \triangledown \phi'_{d2} \\
\quad \phi'_{d1}\ (x,n) = \phi'_{d2}\ (x,y,n) = 1 + n
\end{array}
$$

and we have succeeded in the fusion transformation.

$$(\mathit{length},\ \mathit{length}) \circ (\mathit{nme},\ \mathit{nmd}) = [\![(\phi'_e\ \ \phi'_d),\ id,\ \mathit{out}_{\mathsf{F}}]\!]_{\mathsf{F},\mathsf{F}}$$

Letting $h = \mathit{length} \circ \mathit{nmd}$, we obtain the program in Figure 3(b), which never generates any intermediate list structures.

5.2 Example of zip

The purpose of the second example is to demonstrate how the Mutualizing and Mutual Acid Rain theorems work in the process of fusion transformation.

Consider the function *zip* which turns a pair of two lists into a list of pairs. It was referred in [2] as a limitation of the transformation proposed there in that both inputs of *zip* cannot be deforested. (Only one of them can be deforested.) Takano and Meijer[11] also showed a similar example about *zip*, but there was something unclear in their discussion. Mutualizing and Mutual Acid Rain theorems to a tuple of two single recursions can solve this problem.

The target program of our transformation is *zip* whose inputs, two list structures, are supplied by the results of *maptw* and *mapsq*, where *maptw* and *mapsq* are functions which double and square each element in a given list, respectively. The entire program is given in Figure 4(a). What we would like to do is to transform $f = \mathit{zip} \circ (\mathit{maptw},\ \mathit{mapsq})$ into an efficient program without any intermediate data structures generated by $\mathit{maptw}\ xs$ and $\mathit{mapsq}\ ys$.

The first step is to derive a normal hylomorphism from the definition of *zip*. Since the detailed algorithm of this derivation can be found in [3]. we only show the resultant hylomorphism.

$$
\begin{array}{l}
\mathit{zip} = [\![\phi, \eta, \psi]\!]_{M_2, M_1} \\
\psi\ (xs, ys) = \mathtt{case}\ (xs, ys)\ \mathtt{of} \\
\quad (\mathtt{Nil}, _) \to (\mathtt{1}, ()) \\
\quad (\mathtt{Cons}\ (a, as), \mathtt{Nil}) \to (\mathtt{2}, ()) \\
\quad (\mathtt{Cons}\ (a, as), \mathtt{Cons}\ (b, bs)) \to (\mathtt{3},\ (a, b, (as, bs))) \\
\eta = \eta_1 + \eta_2 + \eta_3 \\
\quad \eta_1\ () = \eta_2\ () = () \\
\quad \eta_3\ (a, b, p) = ((a, b), p) \\
\phi = \phi_1 \triangledown \phi_2 \triangledown \phi_3 \\
\quad \phi_1\ () = \phi_2\ () = \mathtt{Nil} \\
\quad \phi_3\ (x, p) = \mathtt{Cons}\ (x, p)
\end{array}
$$

$$
\begin{array}{l}
f\ (xs, ys) = zip\ (maptw\ xs,\ mapsq\ ys) \\
zip\ (xs, ys) = \texttt{case}\ (xs, ys)\ \texttt{of} \\
\quad (\texttt{Nil}, _) \to \texttt{Nil} \\
\quad (\texttt{Cons}\ (a, as),\ \texttt{Nil}) \to \texttt{Nil} \\
\quad (\texttt{Cons}\ (a, as),\ \texttt{Cons}\ (b, bs)) \\
\qquad \to \texttt{Cons}\ ((a, b),\ zip\ (as, bs))
\end{array}
$$

(a) Original program

$$
\begin{array}{l}
f\ (xs, ys) = \texttt{case}\ (xs, ys)\ \texttt{of} \\
\quad (\texttt{Nil}, _) \to \texttt{Nil} \\
\quad (\texttt{Cons}\ (a, as),\ \texttt{Nil}) \to \texttt{Nil} \\
\quad (\texttt{Cons}\ (a, as),\ \texttt{Cons}\ (b, bs)) \\
\qquad \to \texttt{Cons}\ ((2 * a,\ b * b),\ f\ (as, bs))
\end{array}
$$

(b) Final program

Figure 4: Program using *zip*

M_1 and M_2 are polynomial functors introduced during this derivation procedure.

$$
\begin{aligned}
M_1 &= !\mathbf{1} + !\mathbf{1} + !a \times !b \times I \\
M_2 &= !\mathbf{1} + !\mathbf{1} + !(a, b) \times I
\end{aligned}
$$

Following a similar derivation process, both *maptw* and *mapsq* are represented in the form of normal hylomorphisms using the functor L introduced in Section 2.

$$
\begin{aligned}
maptw &= [\![\, in_L,\ \eta_t,\ out_L \,]\!]_{L,L}, \quad \eta_t = id + \lambda(a, v')\ .\ (2 * a,\ v') \\
mapsq &= [\![\, in_L,\ \eta_s,\ out_L \,]\!]_{L,L}, \quad \eta_s = id + \lambda(b, v')\ .\ (b * b,\ v')
\end{aligned}
$$

So if we define $\mathsf{L} = L \otimes L$, applying the Mutualizing Theorem gives:

$$
\begin{aligned}
(maptw,\ mapsq) &= [\![\, (in_L,\ in_L),\ (\eta_t,\ \eta_s),\ (out_L,\ out_L) \,]\!]_{\mathsf{L},\mathsf{L}} \\
&= [\![\, in_{\mathsf{L}},\ (\eta_t,\ \eta_s),\ out_{\mathsf{L}} \,]\!]_{\mathsf{L},\mathsf{L}}
\end{aligned}
$$

and f has been represented in a compositional style of normal and mutual hylomorphisms.

$$
f = [\![\, \phi, \eta, \psi \,]\!]_{M_2,M_1} \circ [\![\, in_{\mathsf{L}}, (\eta_t,\ \eta_s), out_{\mathsf{L}} \,]\!]_{\mathsf{L},\mathsf{L}}
$$

In order to apply the Mutual Acid Rain Theorem, ψ must be reexpressed in the form of $\psi = \sigma\ out_{\mathsf{L}}$. Rewriting the definition of ψ gives:

$$
\begin{array}{l}
\psi\ (xs, ys) = \texttt{case}\ out_{\mathsf{L}}\ (xs, ys)\ \texttt{of} \\
\quad ((1, ()), _) \to (1, ()) \\
\quad ((2, (a, as)), (1, ())) \to (2, ()) \\
\quad ((2, (a, as)), (2, (b, bs))) \to (3, (a, b, (as, bs))),
\end{array}
$$

and it would be appropriate to define σ as:

$$\begin{array}{l}\sigma\ p\ x = \mathtt{case}\ p\ x\ \mathtt{of}\\ \quad ((1,()),_) \to (1,())\\ \quad ((2,(a,as)),(1,())) \to (2,())\\ \quad ((2,(a,as)),(2,(b,bs))) \to (3,(a,b,(as,bs))).\end{array}$$

Then σ has the type of $\forall A\ .\ (A \to \mathsf{L}\ A) \to A \to M_1\ A$, and we have succeeded in deriving σ such that $\psi = \sigma\ out_{\mathsf{L}}$, and the Mutual Acid Rain Theorem (NormalHylo–MutualAna Fusion) can be applied.

$$\begin{array}{l} f = [\![\phi,\ \eta,\ \sigma\ out_{\mathsf{L}}\ ]\!]_{M_2,M_1} \circ [\![\ in_{\mathsf{L}},\ (\eta_t,\ \eta_s),\ out_{\mathsf{L}}\ ]\!]_{\mathsf{L},\mathsf{L}}\\ \quad = [\![\phi,\ \eta,\ \sigma\ ((\eta_t,\ \eta_s) \circ out_{\mathsf{L}})\ ]\!]_{M_2,M_1}\\ \quad = [\![\phi,\ \eta,\ \sigma\ (\eta_t \circ out_L,\ \eta_s \circ out_L)\ ]\!]_{M_2,M_1}\end{array}$$

Taking the relationships

$$\begin{array}{rl}\eta_t \circ out_L = \lambda xs\ .\ \mathtt{case}\ xs\ \mathtt{of}\ \mathtt{Nil} & \to (1,())\\ \qquad\qquad\qquad\qquad\qquad\quad \mathtt{Cons}\ (a,\ as) & \to (2,(2*a,\ as))\\ \eta_s \circ out_L = \lambda ys\ .\ \mathtt{case}\ ys\ \mathtt{of}\ \mathtt{Nil} & \to (1,())\\ \qquad\qquad\qquad\qquad\qquad\quad \mathtt{Cons}\ (b,\ bs) & \to (2,(b*b,\ bs))\end{array}$$

into account, the third part within the resulting hylomorphism is:

$$\begin{array}{l}\sigma\ (\eta_t \circ out_L,\ \eta_s \circ out_L)\\ \quad = \lambda(xs,ys)\ .\ \mathtt{case}\ (xs,ys)\ \mathtt{of}\\ \qquad (\mathtt{Nil},_) \to (1,())\\ \qquad (\mathtt{Cons}\ (a,\ as),\ \mathtt{Nil}) \to (2,())\\ \qquad (\mathtt{Cons}\ (a,\ as),\ \mathtt{Cons}\ (b,\ bs)) \to (3,(2*a,\ b*b,\ (as,bs)))\end{array}$$

Finally rewriting $[\![\phi,\ \eta,\ \sigma\ (\eta_t \circ out_L,\ \eta_s \circ out_L)\ ]\!]_{M_2,M_1}$ to a familiar form yields the final program (Figure 4(b)) which completely eliminates the intermediate data structures.

5.3 Example of Logic Formulae

Our final example gives a more practical fusion example extracted from [9]. The target program takes as input logic formulae with the type:

$$\begin{array}{rl}\mathtt{Formula} = & \mathtt{Sym\ Char}\\ \mid & \mathtt{Not\ Formula}\\ \mid & \mathtt{Con\ (Formula,\ Formula)}\\ \mid & \mathtt{Dis\ (Formula,\ Formula)}\\ \mid & \mathtt{Imp\ (Formula,\ Formula)}\\ \mid & \mathtt{Eqv\ (Formula,\ Formula)}\end{array}$$

and yields as output their equivalents where implication and equivalence are eliminated by the rules

$$\begin{array}{l}A \to B \Rightarrow \neg A \vee B\\ A \equiv B \Rightarrow (A \to B) \wedge (B \to A)\end{array}$$

$$
\begin{aligned}
&f = \mathit{negin} \circ \mathit{elim} \\
&\mathit{elim}\ (\texttt{Sym}\ c) = \texttt{Sym}\ c \\
&\mathit{elim}\ (\texttt{Not}\ x) = \texttt{Not}\ (\mathit{elim}\ x) \\
&\mathit{elim}\ (\texttt{Con}\ (x, y)) = \texttt{Con}\ (\mathit{elim}\ x,\ \mathit{elim}\ y) \\
&\mathit{elim}\ (\texttt{Dis}\ (x, y)) = \texttt{Dis}\ (\mathit{elim}\ x,\ \mathit{elim}\ y) \\
&\mathit{elim}\ (\texttt{Imp}\ (x, y)) = \texttt{Dis}\ (\texttt{Not}\ (\mathit{elim}\ x),\ \mathit{elim}\ y) \\
&\mathit{elim}\ (\texttt{Eqv}\ (x, y)) = \texttt{Con}\ (\texttt{Dis}\ (\texttt{Not}\ (\mathit{elim}\ x),\ \mathit{elim}\ y), \\
&\qquad\qquad\qquad\qquad \texttt{Dis}\ (\texttt{Not}\ (\mathit{elim}\ y),\ \mathit{elim}\ x)) \\
&\mathit{negin}\ (\texttt{Sym}\ c) = \texttt{Sym}\ c \\
&\mathit{negin}\ (\texttt{Not}\ (\texttt{Not}\ x)) = \mathit{negin}\ x \\
&\mathit{negin}\ (\texttt{Not}\ (\texttt{Con}\ (x, y))) = \texttt{Dis}\ (\mathit{negin}\ (\texttt{Not}\ x),\ \mathit{negin}\ (\texttt{Not}\ y)) \\
&\mathit{negin}\ (\texttt{Not}\ (\texttt{Dis}\ (x, y))) = \texttt{Con}\ (\mathit{negin}\ (\texttt{Not}\ x),\ \mathit{negin}\ (\texttt{Not}\ y)) \\
&\mathit{negin}\ (\texttt{Con}\ (x, y)) = \texttt{Con}\ (\mathit{negin}\ x,\ \mathit{negin}\ y) \\
&\mathit{negin}\ (\texttt{Dis}\ (x, y)) = \texttt{Dis}\ (\mathit{negin}\ x,\ \mathit{negin}\ y)
\end{aligned}
$$

(a) Original program

$$
\begin{aligned}
&f\ (\texttt{Sym}\ c) = \texttt{Sym}\ c \\
&f\ (\texttt{Not}\ x) = g\ x \\
&f\ (\texttt{Con}\ (x, y)) = \texttt{Con}\ (f\ x,\ f\ y) \\
&f\ (\texttt{Dis}\ (x, y)) = \texttt{Dis}\ (f\ x,\ f\ y) \\
&f\ (\texttt{Imp}\ (x, y)) = \texttt{Dis}\ (g\ x,\ f\ y) \\
&f\ (\texttt{Eqv}\ (x, y)) = \texttt{Con}\ (\texttt{Dis}\ (g\ x,\ f\ y),\ \texttt{Dis}\ (g\ y,\ f\ x)) \\
&g\ (\texttt{Sym}\ c) = \texttt{Not}\ (\texttt{Sym}\ c) \\
&g\ (\texttt{Not}\ x) = f\ x \\
&g\ (\texttt{Con}\ (x, y)) = \texttt{Dis}\ (g\ x,\ g\ y) \\
&g\ (\texttt{Dis}\ (x, y)) = \texttt{Con}\ (g\ x,\ g\ y) \\
&g\ (\texttt{Imp}\ (x, y)) = \texttt{Con}\ (f\ x,\ g\ y) \\
&g\ (\texttt{Eqv}\ (x, y)) = \texttt{Dis}\ (\texttt{Con}\ (f\ x,\ g\ y),\ \texttt{Con}\ (f\ y,\ g\ x))
\end{aligned}
$$

(b) Final program

Figure 5: Program on logic formulae

and negation is moved inside. The program is shown in Figure 5(a).

By the introduction of an auxiliary function $negin' = negin \circ \texttt{Not}$, we can transform the definition of *negin* into mutual recursive one with $negin'$.

$$
\begin{aligned}
&negin\ (\texttt{Sym}\ z) = \texttt{Sym}\ z \\
&negin\ (\texttt{Not}\ x) = negin'\ x \\
&negin\ (\texttt{Con}\ (x, y)) = \texttt{Con}\ (negin\ x,\ negin\ y) \\
&negin\ (\texttt{Dis}\ (x, y)) = \texttt{Dis}\ (negin\ x,\ negin\ y) \\
&negin'\ (\texttt{Sym}\ z) = \texttt{Not}\ (\texttt{Sym}\ z) \\
&negin'\ (\texttt{Not}\ x) = negin\ x \\
&negin'\ (\texttt{Con}\ (x, y)) = \texttt{Dis}\ (negin'\ x,\ negin'\ y) \\
&negin'\ (\texttt{Dis}\ (x, y)) = \texttt{Con}\ (negin'\ x,\ negin'\ y)
\end{aligned}
$$

In order to represent them within our framework, we introduce some mutually defined types according to the above definitions. Then (*negin*, $negin'$) becomes a mutual hylomorphism. By applying the Mutual Acid Rain Theorem (MutualCata–NormalHylo Fusion) to the composition of this hylomorphism and *elim* represented in a normal hylomorphism, we get our efficient program.

We omit the detailed derivation process, but only show the final result in Figure 5(b). In the transformed program, the intermediate data structure between *elim* and *negin* has been successfully eliminated, which would be impossible without employing mutual hylomorphisms.

6 Discussion and Related Work

In [6], Malcolm proposed a categorical approach to Constructive Algorithmics. Putting the categorical base on SET, he discussed mutual recursion using a concrete example of tree and forest without a detailed formalization. Although our default category is CPO other than SET, our work can be regarded as a generalization of his work, expanding his approach to anamorphism and hylomorphism over mutual recursive data types.

Fokkinga [1] proposed the concept of *mutumorphism* for structuring mutual functions inducting over the same data structure. Our formalization assumes that functions expressed by a mutual hylomorphism may be defined on different data structures, possibly of different types. In order to deal with functions inducting over the same data as mutumorphisms do, we have to duplicate the shared data into a pair, then proceed with transformations within our framework. After that, the resulting function may be reexpressed in terms of mutumorphism to eliminate multiple traversal of the same data structures.

Sheard and Fegaras [10] described catamorphisms over mutual recursive data types, and proposed corresponding fusion theorem (they call it "Promotion Theorem"). In contrast to their theorem, our Fusion Theorem treats mutual recursive functions into a single pair. In addition, since they did not discuss

the duality of catamorphisms, it is impossible to fuse the *zip* program given in Section 5.2.

For fusing the composition between a hylomorphism (traversing multiple data structures simultaneously) and a pair of two hylomorphisms, our previous work in [4] gave a generalization of the (normal) Acid Rain Theorem which has a different form compared with the original one. From the viewpoint of program transformation, therefore, this "generalized" theorem was introduced as a new transformation rule to deal with this kind of fusion. On the contrary, the extended theorems for mutual recursion proposed in this paper preserve the original form and keep the set of transformation rules unchanged. The applicability of this work is expected to be much wider, because it is based on the general mutual recursive structure.

This work is inspired by Takano and Meijer [11] who proposed the hylomorphism in triplet form and the (normal) Acid Rain Theorem. We have followed and extended their approach, with the result that our Mutual Acid Rain Theorem is a more generalized version of foldr/build rule in [2].

7 Conclusion

In this paper, we proposed a natural extension of hylomorphism to formally deal with mutual recursion by means of bifunctors. All the concepts and rules for single recursive case have been shown to be also effective in the mutual case. Although the minimal case (two data types are depending on each other) was discussed in this paper, this discussion applies to arbitrary number of mutual dependencies by replacing bifunctors with n-ary functors. Finally we gave three examples to illustrate our idea.

We are now developing a calculational-based program transformation system HYLO Calculator [8] based on our previous research [3]. In this system, hylomorphisms are regarded as the standard form of recursive functions, equipped with powerful fusion rules such as Hylo Fusion and Acid Rain. Currently its target programs for transformation are those composed only by single recursions. We plan to extend it to be able to deal with mutually dependent recursions exploiting our mutual hylomorphism proposed in this paper. Thanks to the uniformity of single and mutual recursions, the extension of the system is expected to need not much effort.

Acknowledgements

We thank Akihiko Takano for his valuable discussions with us. Also we would like to thank the anonymous referees who provided many comments for improvement of this paper. This work is partially supported by Grant-in-Aid for Scientific Research (No. 09245207, No. 09780254) of the Ministry of Education, Science and Culture of Japan.

References

1. Fokkinga, M.: Law and Order in Algorithmics, Ph.D thesis, Dept.Inf. University of Twente (1992).
2. Gill, A., Launchbury, J. and Jones, S.P.: A Short Cut to Deforestation, *Proc. FPCA'93*, ACM Press, pp.223–232 (1993).
3. Hu, Z., Iwasaki, H. and Takeichi, M.: Deriving Structural Hylomorphisms from Recursive Definitions, *Proc. ICFP'96*, ACM Press, pp.73–82 (1996).
4. Hu, Z., Iwasaki, H. and Takeichi, M.: An Extension of Acid Rain Theorem, *Proc. 2nd Fuji International Workshop on Functional and Logic Programming* (1996).
5. Launchbury, J. and Sheard, T.: Warm Fusion: Deriving Build-Catas from Recursive Definitions, *Proc. FPCA'95*, ACM Press, pp.314–323 (1995).
6. Malcolm, G: Data structures and Program Transformation, *Science of Computer Programming*, Vol.14, Nos.2–3, pp.255–279 (1990).
7. Meijer, E., Fokkinga, M. and Paterson, R.: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire, *Proc. FPCA'91*, LNCS 523, Springer–Verlag, pp.124–144 (1991).
8. Onoue, Y., Hu, Z., Iwasaki, H. and Takeichi, M.: A Calculational Fusion System HYLO, *Proc. IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, Chapman&Hall, pp.76–106 (1997).
9. Runciman, C. and Wakeling, D.: Heap Profiling of Lazy Functional Programs, *Journal of Functional Programming*, Vol.3, No.2, pp.217–245 (1993).
10. Sheard, T. and Fegaras, L.: A Fold for All Seasons, *Proc. FPCA'93*, ACM Press, pp.233–242 (1993).
11. Takano, A. and Meijer, E.: Shortcut Deforestation in Calculational Form, *Proc. FPCA'95*, ACM Press, pp.306–313 (1995).
12. Wadler, P.: Theorems for Free!, *Proc. FPCA'89*, ACM, pp.347–359 (1989).