

# Shortcut Deforestation in Calculational Form

Akihiko Takano  
Hitachi Advanced Research Lab  
Hatoyama, Saitama 350-03  
Japan  
takano@harl.hitachi.co.jp

Erik Meijer  
Utrecht University  
PO Box 80089, 3508 TB Utrecht  
The Netherlands  
erik@cs.ruu.nl

## Abstract

In functional programming, intermediate data structures are often used to “glue” together small programs. Deforestation is a program transformation to remove these intermediate data structures automatically. We present a simple algorithm for deforestation based on two fusion rules for hylomorphism, an expressive recursion pattern. A generic notation for hylomorphisms is introduced, where natural transformations are explicitly factored out, and it is used to represent programs. Our method successfully eliminates intermediate data structures of any algebraic type from a much larger class of compositional functional programs than previous techniques.

## 1 Introduction

In functional programming, programs are often constructed by “gluing” together small components, using intermediate data structures to convey information between them. Such data are constructed in one component and later consumed in another component, but never appear in the result of the whole program. The compositional style of programming has many advantages of clarity and higher level of modularity, but these intermediate data structures give rise to an efficiency problem.

Inspired by Turchin’s early work on the supercompiler [TNT82, Tur86], Wadler [Wad88] introduced the idea of *deforestation* to tackle this problem. His algorithm for deforestation eliminates arbitrary tree-like intermediate data structures (including lists) when applied to treeless programs. There have been various attempts to extend his method ([Chi92, Sør94]), but still major drawbacks remain. All these algorithms basically have to keep track of all function calls occurred previously, and suitably introduce a definition of recursive function on detecting a repetition. This corresponds to the fold step of Burstall and Darlington ([BD77]). The process of keeping track of function calls and the clever control to avoid infinite unfolding

introduces substantial cost and complexity in algorithms, which prevent deforestation to be adopted as part of the regular optimizations in any serious compilers of functional languages.

Recently two new approaches to deforestation have been proposed [GLPJ93, SF93]. Both of them pick up the function *fold* as a useful template to capture the structure of programs, and apply transformations only to programs written in terms of the fold function. Both techniques do not require any global analysis to guarantee termination and the applicability of their rules of transformation can be checked locally. Because their theoretical basis can be found in the study on *Constructive Algorithmics*<sup>1</sup> ([Mee86, MFP91, Mee92, Mei92, Fok92, Jeu93]), we baptise them as *deforestation in calculational form*.

Although the method in [GLPJ93] is limited to the specific data structure of lists, it was shown clearly that this calculation-based deforestation is more practical than the original style deforestation and its extensions. By using the *foldr/build* rule as the basis to standardize the structure of consuming/producing functions of lists, their transformation is the repetitive applications of the single rule of cancellation for a pair of *foldr* and *build*. Each application of the rule can be seen as a canned application of *unfold/simplify/fold* in the traditional deforestation. In [GLPJ93] the rule and its correctness proof are given only in the specific context of lists, and the extension to the other data structures is simply suggested. Once embedded in the proper theoretical framework it becomes clear how to generalize their method to other data structures.

Sheard and Fegaras [SF93] demonstrated that folding can be defined for many algebraic types definable in languages like ML (i.e. mutually recursive sum-of-product types). Their *normalization algorithm* automatically calculates a potentially normalizable fold program (analogous to a treeless program) into its canonical form. The algorithm is essentially based on the so-called fusion theorem, and repetitively replaces the nested application of two fold functions with one fold. They also gave definitions of other recursive forms such as generalized unfolds (derive function) and primitive recursion together with their corresponding fusion theorems. The normalization algorithms for these recursive patterns were not given.

In this paper we show that a single transformation rule (and its dual), the *Acid Rain Theorem* [Mei94], elegantly generalizes the *foldr/build* to *any* algebraic

<sup>1</sup>For more info check out the WWW page <http://www.cs.utwente.nl/~fokkinga/algorithmics.html>

data types. We introduce a generic notation for *hylomorphisms*, which generalizes both folds and unfolds and use it to represent the structure of programs. We show the acid rain theorem can be stated as the fusion rules for hylomorphisms with no side condition. Based on these rules, we introduce a new deforestation algorithm to eliminate intermediate data structures of any algebraic type from much larger class of compositional functional programs than the deforestation algorithms above.

The contribution of this paper is as follows:

- Our technique is applicable to any functional program in compositional style, and removes intermediate data of any algebraic type. We propose a generic notation for *hylomorphisms* to explicitly factor out the natural transformation, and the structure of the program is represented using this notation. Our new representation facilitates us to state the acid rain theorem and the rules of transformation in uniform way.
- Our technique is a direct generalization of [GLPJ93]. Thanks to the categorical characterization of data types, the theorem naturally covers the dual of the `foldr/build` theorem. Our optimization is based on two simple local transformations: the generalization of fold/build cancellation rule and the dual of it. The technique is also cheap and practical to implement in real compilers.
- Our method is more powerful than the method in [GLPJ93] even when restricted to the list data structure. Typically, the function `zip`, which could not be deforested in both parameters by their method, is not an exception any more. Our method successfully deforests `zip` in both of its parameters (Section 4.5).
- Our method also generalizes the result of [SF93] by adopting hylomorphism as the basic form to represent programs. Because fold (catamorphism) and unfold (dual of catamorphism) are instances of hylomorphism, our method does not only work on fold programs but also on the programs built up from fold and its dual. Our technique can also be extended to work on primitive recursion.

This paper is organized as follows. In section 2 we review the previous work in program calculation which is the theoretical base of our method. In section 3 we introduce a triplet notation for hylomorphisms and the fusion rules for them, which are the key rules of our method. In section 4 our transformation algorithm is defined and applied to some examples. In section 5 we discuss related work.

## 2 Program Calculation

In this section we briefly review the previous work on constructive algorithmics ([MFP91, Mei92, Fok92, Mei94]) and explain some basic facts which provides the theoretical basis of our deforestation algorithm. In this paper our default category  $\mathcal{C}$  for types is  $\mathcal{CPO}$ , the category of complete partial orders with continuous functions. This choice facilitates us to handle arbitrary recursive equations in the framework close to lazy functional programming languages.

## 2.1 Functors

Endofunctors on  $\mathcal{C}$  (functors from  $\mathcal{C}$  to  $\mathcal{C}$ ) capture the signature of (algebraic) data types. In this paper we assume that all data types are defined by functors whose operation on functions are continuous. In  $\mathcal{CPO}$  all functors defined using basic functors below and *type functors* (map functors) satisfy this condition. The definition of type functors is given in section 2.3. Basic functors we assume are *id* (identity),  $\underline{A}$  (constants),  $\times$  (product),  $A_{\perp}$  (strictify) and  $+$  (separated sum). We give the definitions of product and separated sum functors and related combinators.

**Definition 2.1** *The product  $A \times B$  of two types  $A$  and  $B$  and its operation to functions are defined as:*

$$\begin{aligned} A \times B &= \{(a, b) \mid a \in A, b \in B\} \\ (f \times g)(a, b) &= (f a, g b) \end{aligned}$$

*The following combinators (left/right projections and split  $\Delta$ ) are related to the product functor:*

$$\begin{aligned} \text{exl}(a, b) &= a \\ \text{exr}(a, b) &= b \\ (f \Delta g) a &= (f a, g a) \end{aligned}$$

*$f \times g = (f \circ \text{exl}) \Delta (g \circ \text{exr})$  characterizes their relation. The standard notation for  $f \Delta g$  in category theory is  $\langle f, g \rangle$ .*

**Definition 2.2** *The separated sum  $A + B$  of two types  $A$  and  $B$  and its operation to functions are defined as:*

$$\begin{aligned} A + B &= (\{0\} \times A \cup \{1\} \times B)_{\perp} \\ (f + g)_{\perp} &= \perp \\ (f + g)(0, a) &= (0, f a) \\ (f + g)(1, b) &= (1, g b) \end{aligned}$$

*The following combinators (left/right injections and *junc*  $\nabla$ ) are related to the separated sum functor:*

$$\begin{aligned} \text{inl } a &= (0, a) \\ \text{inr } b &= (1, b) \\ (f \nabla g)_{\perp} &= \perp \\ (f \nabla g)(0, a) &= f a \\ (f \nabla g)(1, b) &= g b \end{aligned}$$

*$f + g = (\text{inl} \circ f) \nabla (\text{inr} \circ g)$  characterizes their relation. The standard notation for  $f \nabla g$  in category theory is  $[f, g]$ .*

## 2.2 Data Types as Initial Fixed Points of Functors

Let  $F$  be an endofunctor on  $\mathcal{C}$ . An *F-algebra* is a strict function of type  $FA \rightarrow A$ . The set  $A$  is called the *carrier* of the algebra. Dually, an *F-co-algebra* is a (not necessarily strict) function of type  $A \rightarrow FA$ . An *F-homomorphism*  $h : A \rightarrow B$  from F-algebra  $\varphi : FA \rightarrow A$  to  $\psi : FB \rightarrow B$  is a function which satisfies  $h \circ \varphi = \psi \circ Fh$ . We use a concise notation  $h : \varphi \rightarrow_F \psi$  to represent this property. Category  $\mathcal{ALG}(F)$  is the category whose objects are F-algebras and whose morphisms are F-homomorphisms. Dually,  $\mathcal{COALG}(F)$  is the category of F-co-algebras with F-co-homomorphisms.

The nice thing to work in  $\mathcal{CPO}$  is that  $\mathcal{ALG}(F)$  has an initial object and  $\mathcal{COALG}(F)$  has a final object, and their carriers coincide. By Scott's inverse limit construction to construct fixed points of functors, we get an F-algebra  $in_F : F\mu F \rightarrow \mu F$  which is initial in  $\mathcal{ALG}(F)$ , and an F-co-algebra  $out_F : \mu F \rightarrow F\mu F$  which is final in  $\mathcal{COALG}(F)$ . They are each other's inverses, and establish an isomorphism  $\mu F \cong F\mu F$  in  $\mathcal{C}$ . They also satisfy the equation  $\mu(\lambda f . in_F \circ Ff \circ out_F) = id_{\mu F}$ . Here  $\mu$  is the fix point operator that satisfies:  $\mu h = h(\mu h)$ . We say the type  $\mu F$  is the (*algebraic data type*) defined by the functor  $F$ .

Type declarations define data types and initial algebras. For example,

$$nat ::= Zero \mid Succ \ nat$$

declares  $in_N = Zero \nabla Succ : N \ nat \rightarrow nat$  is the initial N-algebra, where  $N$  is the functor  $N = \perp + id$  (i.e.  $N A = \perp + A$  and  $N h = id_{\perp} + h$ ) and  $nat = \mu N$ . Here,  $\perp$  is the terminal object in  $\mathcal{C}$  and  $Zero : \perp \rightarrow nat$  is a constant.

Data types can be parametrized. For example, the type declaration of the list with elements of type  $A$  :

$$list \ A ::= Nil \mid Cons(A, list \ A)$$

declares  $in_{L_A} = Nil \nabla Cons : L_A(list \ A) \rightarrow list \ A$  is the initial  $L_A$ -algebra, with the functor  $L_A = \perp + \underline{A} \times id$  (i.e.  $L_A B = \perp + (A \times B)$  and  $L_A h = id_{\perp} + (id_A \times h)$ ). As the final  $L_A$ -co-algebra, we can take:

$$out_{L_A} = (id_{\perp} + hd \triangle tl) \circ is\_nil? : list \ A \rightarrow L_A(list \ A)$$

Here  $p?$  injects a value  $x$  of type  $A$  into the union type  $A+A$  according as the result of  $p \ x$ . The definition of  $out_{L_A}$  above corresponds to

$$\lambda x. \text{if } is\_nil \ x \text{ then } \perp \text{ else } (hd \ x, tl \ x)$$

We sometimes write  $L(A, B)$  instead of  $L_A(B)$ , where we think  $L$  is a bifunctor (i.e.  $L(A, B) = \perp + (A \times B)$  and  $L(f, h) = id_{\perp} + (f \times h)$ ).

Every parametrized type constructor is associated with a certain functor, called a *type functor* (a map functor). For example, the type functor  $list$  coincides the familiar  $map$  function. The type functor can be defined in general using the notion of catamorphism. We will give the definition when it is available in the next section.

### 2.3 Catamorphisms and Anamorphisms

Initiality of  $in_F$  in  $\mathcal{ALG}(F)$  means: for any F-algebra  $\varphi : FA \rightarrow A$ , there exists a unique F-homomorphism  $h : in_F \rightarrow_F \varphi$ . This homomorphism is called *catamorphism* and denoted by  $(\lfloor \varphi \rfloor)_F$ . Dually, the finality of  $out_F$  in  $\mathcal{COALG}(F)$  means: for any F-co-algebra  $\psi : A \rightarrow FA$ , there exists a unique F-co-homomorphism  $h : \psi \rightarrow_F out_F$  called *anamorphism*, and is denoted by  $(\lceil \psi \rceil)_F$ . These two morphisms can equivalently be defined as least fixed points:

$$\begin{aligned} (\lfloor - \rfloor)_F & : (FA \rightarrow A) \rightarrow \mu F \rightarrow A \\ (\lfloor \varphi \rfloor)_F & = \mu(\lambda f . \varphi \circ Ff \circ out_F) \\ (\lceil - \rceil)_F & : (A \rightarrow FA) \rightarrow A \rightarrow \mu F \\ (\lceil \psi \rceil)_F & = \mu(\lambda f . in_F \circ Ff \circ \psi) \end{aligned}$$

We sometimes omit the suffix  $F$  when it is clear from the context.

From these fixed point definitions and the properties of  $\mu$ ,  $in_F$  and  $out_F$ , it is easy to see the following equalities hold:

$$\begin{aligned} (\lfloor \varphi \rfloor)_F & = \varphi \circ F(\lfloor \varphi \rfloor)_F \circ out_F \\ (\lfloor \varphi \rfloor)_F \circ in_F & = \varphi \circ F(\lfloor \varphi \rfloor)_F \\ (\lceil \psi \rceil)_F & = in_F \circ F(\lceil \psi \rceil)_F \circ \psi \\ out_F \circ (\lceil \psi \rceil)_F & = F(\lceil \psi \rceil)_F \circ \psi \end{aligned}$$

Catamorphisms are generalized fold operations that substitute the constructors of a data type with other operations of same signature. Catamorphisms provide a standard way to consume a data structure, and dually, anamorphisms offer a standard way for constructing data structures. Anamorphisms are generalized unfold operations. It has been argued in [GLPJ93, SF93] that many standard functions over data structures can be represented using catamorphisms.

We are now ready to give the definition of type functors in general. Given an initial algebra  $in : F(A, TA) \rightarrow TA$ , the type functor  $T$  is defined by

$$Tf = (\lfloor in \circ F(f, id) \rfloor).$$

For example for the example of lists,

$$\begin{aligned} in_{L_A} & : L(A, list \ A) \rightarrow list \ A \\ list \ f & = ((Nil \nabla Cons) \circ L(f, id))_{L_A} \\ & = ((Nil \nabla Cons) \circ (id_{\perp} + (f \times id)))_{L_A} \\ & = (Nil \nabla (Cons \circ (f \times id)))_{L_A}. \end{aligned}$$

By expanding the definition of the catamorphism, we get the definition of  $map$  function on lists.

### 2.4 Hylomorphisms

A *hylomorphism*  $(\lfloor \varphi, \psi \rfloor)_F$  is what you get by composing a fold with an unfold:  $(\lfloor \varphi \rfloor)_F \circ (\lceil \psi \rceil)_F$ . Equivalently [MFP91], a hylomorphism is the fixed shape of recursion that comes with a particular functor.

$$\begin{aligned} (\lfloor -, \rceil)_F & : (FA \rightarrow A) \times (B \rightarrow FB) \rightarrow B \rightarrow A \\ (\lfloor \varphi, \psi \rfloor)_F & = \mu(\lambda f . \varphi \circ Ff \circ \psi) \end{aligned}$$

It is obvious from definitions that catamorphisms and anamorphisms are special cases of hylomorphisms:  $(\lfloor \varphi \rfloor)_F = (\lfloor \varphi, out_F \rfloor)_F$  and  $(\lceil \psi \rceil)_F = (\lceil in_F, \psi \rceil)_F$ .

Hylomorphism  $(\lfloor \varphi, \psi \rfloor)_F$  is a recursive function whose call graph is isomorphic to the data type  $\mu F$ . It is known most practical functions can be represented as hylomorphisms ([BM94]). Meertens proved in [Mee92] that every primitive recursive function on an algebraic type can be represented as a hylomorphism on some algebraic type.

Hylomorphisms enjoy many useful laws for program calculation. The law called *HyloShift*,

$$\eta : F \rightarrow G \Rightarrow (\lfloor \varphi \circ \eta, \psi \rfloor)_F = (\lfloor \varphi, \eta \circ \psi \rfloor)_G$$

shows that natural transformations can be shifted between the two parameters of hylomorphisms. Based on this property we will introduce a new notation for hylomorphisms in section 3.3.

### 3 Rules for Deforestation

#### 3.1 Shortcut Deforestation

The core of the shortcut deforestation algorithm proposed in [GLPJ93] is the single rule of cancellation for a pair of foldr and build:

$$\begin{aligned} g &: \forall \beta. (A \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta \\ &\Rightarrow \text{foldr } k z (\text{build } g) = g k z \end{aligned}$$

where the function `build` is defined as

$$\text{build } g = g \text{ Cons Nil.}$$

By using `foldr` and `build` to standardize the structure of consuming/producing functions of lists, their transformation is the repetitive applications of this rule. By expanding the definition of `build`, we get

$$\begin{aligned} g &: \forall \beta. (A \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta \\ &\Rightarrow \text{foldr } k z (g \text{ Cons Nil}) = g k z \end{aligned}$$

This rule can be restated in terms of catamorphisms:

$$\begin{aligned} g &: \forall B. (\mathbb{L}_A B \rightarrow B) \rightarrow B \\ &\Rightarrow (\varphi)_{\mathbb{L}_A} (g \text{ in}_{\mathbb{L}_A}) = g \varphi \end{aligned}$$

This law nicely captures the intuition behind a catamorphism, namely replace the constructors  $\text{in}_{\mathbb{L}_A}$ , the initial  $\mathbb{L}_A$ -algebra, by function  $\varphi$ , an arbitrary  $\mathbb{L}_A$ -algebra.

Now we have enough clues to generalize this rule for other algebraic types.

#### 3.2 Acid Rain Theorem

The analysis in the previous section suggests that the following theorem holds in general ([Mei94]).

##### Theorem 3.1 (Acid Rain)

$$g : \forall A. (F A \rightarrow A) \rightarrow A \Rightarrow (\varphi)_F (g \text{ in}_F) = g \varphi$$

**Proof** The free theorem ([Wad89]) associated with the type of  $g$  is

$$f \circ \psi = \varphi \circ F f \Rightarrow f (g \psi) = g \varphi$$

In case  $g$  is defined using recursion,  $f$  needs to be strict as well. By taking  $f := (\varphi)_F$ ,  $\psi := \text{in}_F$ , this rule is instantiated to

$$(\varphi)_F \circ \text{in}_F = \varphi \circ F (\varphi)_F \Rightarrow (\varphi)_F (g \text{ in}_F) = g \varphi$$

This premise trivially holds because  $(\varphi)_F$  satisfies its defining fixed point equation (and is strict as well).  $\square$

For the applications we have in mind it is more convenient to rephrase the Acid Rain theorem on the function level:

##### Theorem 3.2 (Acid Rain : Catamorphism)

$$\begin{aligned} g &: \forall A. (F A \rightarrow A) \rightarrow B \rightarrow A \\ &\Rightarrow (\varphi)_F \circ (g \text{ in}_F) = g \varphi \end{aligned}$$

Here  $B$  is some fixed type that does not depend on  $A$ .

One of the benefits of working on the function level is that we can take the dual of this rule.

##### Theorem 3.3 (Acid Rain : Anamorphism)

$$\begin{aligned} h &: \forall A. (A \rightarrow F A) \rightarrow A \rightarrow B \\ &\Rightarrow (h \text{ out}_F) \circ (\psi)_F = h \psi \end{aligned}$$

It is not difficult to prove these theorems as a free theorem in the same way as the first one. We omit the proof here.

The two acid rain theorems show how we can generalize shortcut deforestation to any algebraic data types, and moreover provide yet another deforesting transformation for values produced by unfolding. Although these two rules are general enough to capture every case where we can deforest intermediate data structures of arbitrary type, it is not easy to design an automatic deforestation algorithm based on them. It is not obvious how to find the places (redexes) in the program where these rules are applicable, and in which order we should apply these rules when the redexes are overlapping. As they are stated in the function level and cover the dual cases (anamorphisms), there are more chance to have overlapping redexes.

To tackle this problem we need some syntactic clue in the program for searching the candidates for  $g$  or  $h$  of these rules. The ideal representation of program must facilitate us finding these candidate polymorphic functions together with catamorphisms and anamorphisms. Natural choice would be hylomorphisms, which include catamorphisms and anamorphisms as special cases. And most practical functions can be represented as hylomorphisms ([BM94]).

#### 3.3 Hylomorphisms as Triplets

We adopt hylomorphisms as the basic components to represent the structure of programs. As the preparation for designing our deforestation algorithm, we restate the above two Acid Rain rules as a single theorem about hylomorphisms.

Many functions are catamorphic on their input type and anamorphic on their output type at the same time. For example, the function `length` that returns the length of a given list is a catamorphism on list  $A$  and an anamorphism on `nat`:

$$\begin{aligned} \text{length} &= (\text{Zero} \nabla (\text{Succ} \circ \text{err}))_{\mathbb{L}_A} \\ &= \llbracket (\text{id}_{\perp} + \text{tl}) \circ \text{isnil?} \rrbracket_N \end{aligned}$$

For any natural transformation  $\eta : F \rightarrow G$ , `HyloShift` implies

$$\begin{aligned} (\text{in}_G \circ \eta)_F &= \llbracket \text{in}_G \circ \eta, \text{out}_F \rrbracket_F \\ &= \llbracket \text{in}_G, \eta \circ \text{out}_F \rrbracket_G = \llbracket \eta \circ \text{out}_F \rrbracket_G \end{aligned}$$

If we do not want to miss the chance that this kind of hylomorphism consists the redex for the Acid Rain rules, the possibilities of `HyloShift` rule application have to be considered always.

To avoid this cumbersomeness by preparing a neutral representation for them, we introduce a new notation of hylomorphisms, where the natural transformation is explicitly factored out as an extra second parameter.

**Definition 3.1 (Hylomorphism in triplet form)**

*Hylomorphism*  $\llbracket \varphi, \eta, \psi \rrbracket_{G,F}$  is defined as follows:

$$\llbracket \dashv, \dashv, \dashv \rrbracket_{G,F} : \forall A, B. (GA \rightarrow A) \times (F \rightarrow G) \times (B \rightarrow FB) \rightarrow (B \rightarrow A)$$

$$\llbracket \varphi, \eta, \psi \rrbracket_{G,F} = \mu(\lambda f. \varphi \circ \eta \circ Ff \circ \psi)$$

We sometimes omit the suffix  $G, F$  when it is clear from the context.

With this notation the hylomorphisms which are essentially built up from some natural transformation can be represented as it is. Now the example explained above has the proper neutral representation as a hylomorphism:

$$\llbracket in_G \circ \eta \rrbracket_F = \llbracket in_G, \eta, out_F \rrbracket_{G,F} = \llbracket \eta \circ out_F \rrbracket_G$$

The function *length* is represented as follows:

$$length = \llbracket in_N, id+exr, out_{L_A} \rrbracket_{N,L_A}$$

The type functor  $\mathbb{T}$  defined in section 2.3 can always be representable as a hylomorphism of this kind:

$$\mathbb{T}f = \llbracket in, F(f, id), out \rrbracket.$$

With this notation, it becomes much easier to judge whether a hylomorphism is either a catamorphism or an anamorphism. If the third parameter of the hylomorphism is  $out_F$ , it is a  $F$ -catamorphism, and if its first parameter is  $in_G$ , it is a  $G$ -anamorphism.

The HyloShift rule becomes

$$\llbracket \varphi, \eta, \psi \rrbracket_{G,F} = \llbracket \varphi \circ \eta, id, \psi \rrbracket_{F,F} = \llbracket \varphi, id, \eta \circ \psi \rrbracket_{G,G}$$

**3.4 Rules for Hylomorphism Fusion**

The Acid Rain Theorems for catamorphism and anamorphism can be restated in terms of the new notation for hylomorphisms.

**Theorem 3.4 (Cata-HyloFusion)**

$$\tau : \forall A. (FA \rightarrow A) \rightarrow FA \rightarrow A \Rightarrow$$

$$\llbracket \varphi, \eta_1, out_F \rrbracket_{G,F} \circ \llbracket \tau in_F, \eta_2, \psi \rrbracket_{F,L} = \llbracket \tau (\varphi \circ \eta_1), \eta_2, \psi \rrbracket_{F,L}$$

**Proof** The first component of the left-hand side is just a catamorphism  $\llbracket \varphi \circ \eta_1 \rrbracket_F$ . The second component hylomorphism has a type  $B \rightarrow A$  where  $A$  and  $B$  are the carriers of  $\tau in_F$  and  $\psi$ , correspondingly. Consider the following lambda term:

$$g = \lambda f. \llbracket \tau f, \eta_2, \psi \rrbracket_{F,L}$$

As  $\tau$  is polymorphic,  $g$  becomes also polymorphic and has a type:

$$g : \forall A. (FA \rightarrow A) \rightarrow B \rightarrow A$$

This type exactly match the type requirement for  $g$  in Theorem 3.2, and the simple instantiation proves this theorem.  $\square$

Taking the dual of this theorem, we get the following theorem.

**Theorem 3.5 (Hylo-AnaFusion)**

$$\sigma : \forall A. (A \rightarrow FA) \rightarrow A \rightarrow FA \Rightarrow$$

$$\llbracket \varphi, \eta_1, \sigma out_F \rrbracket_{G,F} \circ \llbracket in_F, \eta_2, \psi \rrbracket_{F,L} = \llbracket \varphi, \eta_1, \sigma (\eta_2 \circ \psi) \rrbracket_{G,F}$$

These two rules provides the theoretical basis of our deforestation algorithm.

Note that HyloSplit<sup>1</sup> rule is just a special case of these rules, by taking  $\tau := id$  or  $\sigma := id$ :

$$\llbracket \varphi, \eta_1, out_F \rrbracket_{G,F} \circ \llbracket in_F, \eta_2, \psi \rrbracket_{F,L} = \llbracket \varphi, \eta_1 \circ \eta_2, \psi \rrbracket_{G,F}$$

**4 Transformation based on Hylomorphism Fusion**

Our transformation algorithm is completely based on the two HyloFusion rules above, and repetitively applies these rules until there is no redex left in the program. Both rules replace the composition of two hylomorphisms with one hylomorphism, so termination is vacuous. The application of the rule is always safe in the sense that every application removes some intermediate data structure which has been passed through the eliminated composition.

The essence of our transformation algorithm is the reduction strategy which controls the order of redexes to be picked up. How to decide the reduction order of overlapping redexes is the main part of the algorithm.

**4.1 The Language**

In principle, our transformation method is applicable to any functional program as long as it includes some compositions of hylomorphisms. But to get the most out of our method, we assume here the programs are entirely written as compositions of hylomorphisms. Of course programs may include some lambda expressions inside and outside of hylomorphisms, but it is not allowed to write explicit recursion. Every recursion has to be standardized using hylomorphisms.

Basic functors and the related combinators can be freely used to combine hylomorphisms or to define the parameters for hylomorphisms.

**4.2 Two Examples of Transformation**

Let us consider the following three standard functions on the list data structure. They are defined as hylomorphisms:

$$length = \llbracket in_N, id+exr, out_{L_B} \rrbracket$$

$$map f = \llbracket in_{L_B}, L(f, id), out_{L_A} \rrbracket$$

$$(++ ys) = \llbracket \tau in_{L_A}, id, out_{L_A} \rrbracket$$

$$\text{where } \tau = \lambda n \nabla c. (\llbracket n \nabla c, id, out_{L_A} \rrbracket ys) \nabla c$$

Here we assume that ' $\lambda n \nabla c \dots$ ' is a pattern that matches any ' $f \nabla h$ '.

To define  $(++ ys)$  in the proper abstract level, the constructors (*Nil* and *Cons*) in  $ys$  are replaced by  $n$  and  $c$  systematically, using another hylomorphism in the definition of  $\tau$ . This exactly corresponds to the definition of  $++$  in [GLPJ93].

Then the composition  $length \circ (map f) \circ (++) ys$  is transformed as follows:

$$\begin{aligned}
& length \circ (map f) \circ (++) ys \\
= & \{ \text{definition of } length, map \text{ and } ++ \} \\
& \llbracket in_N, id+exr, out_{L_B} \rrbracket \\
& \circ \llbracket in_{L_B}, L(f, id), out_{L_A} \rrbracket \circ \llbracket \tau in_{L_A}, id, out_{L_A} \rrbracket \\
= & \{ \text{HyloSplit}^{\perp 1} \} \\
& \llbracket in_N, (id+exr) \circ L(f, id), out_{L_A} \rrbracket \circ \llbracket \tau in_{L_A}, id, out_{L_A} \rrbracket \\
= & \{ \text{Cata-HyloFusion} \} \\
& \llbracket \tau(in_N \circ (id+exr) \circ L(f, id)), id, out_{L_A} \rrbracket \\
= & \{ \text{definition of } in_N \text{ and } L \} \\
& \llbracket \tau((Zero \nabla Succ) \circ (id+exr) \circ (\perp + f \times id)), id, out_{L_A} \rrbracket \\
= & \{ \text{properties of basic functors} \} \\
& \llbracket \tau(Zero \nabla (Succ \circ exr)), id, out_{L_A} \rrbracket \\
= & \{ \text{definition of } \tau \} \\
& \llbracket [Zero \nabla (Succ \circ exr), id, out_{L_A}] \nabla (Succ \circ exr), \\
& \quad id, out_{L_A} \rrbracket
\end{aligned}$$

By inlining the definition of hylomorphism, we get the familiar recursive definition:

$$\begin{aligned}
length \circ (map f) \circ (++) ys &= h \\
\text{where } h \text{ Nil} &= g \text{ ys} \\
\text{where } g \text{ Nil} &= 0 \\
g \text{ Cons}(x, xs) &= 1 + (g \text{ xs}) \\
h \text{ Cons}(x, xs) &= 1 + (h \text{ xs})
\end{aligned}$$

Note that the intermediate list structure produced by  $map f$  and  $(++) ys$  is no longer generated.

Our second example includes the reverse function  $rev$  on the list.  $rev$  is discussed in [SF93] as an example which is not potentially normalizable. Sheard and Fegaras devise the second-order fold to treat it properly. The naive quadratic definition of  $rev$  can be represented as a hylomorphism:

$$rev = \llbracket \sigma in_{L_A}, id, out_{L_A} \rrbracket$$

$$\text{where } \sigma = \lambda n \nabla c. (n \nabla (\lambda(x, r). \llbracket c(x, n) \nabla c, id, out_{L_A} \rrbracket r))$$

Then the composition  $length \circ rev$  is transformed as follows:

$$\begin{aligned}
& length \circ rev \\
= & \{ \text{definition of } length \text{ and } rev \} \\
& \llbracket in_N, id+exr, out_{L_A} \rrbracket \circ \llbracket \sigma in_{L_A}, id, out_{L_A} \rrbracket \\
= & \{ \text{Cata-HyloFusion} \} \\
& \llbracket \sigma(in_N \circ (id+exr)), id, out_{L_A} \rrbracket \\
= & \{ \text{definition of } in_N \text{ and properties of functors} \} \\
& \llbracket \sigma(Zero \nabla (Succ \circ exr)), id, out_{L_A} \rrbracket \\
= & \{ \text{definition of } \sigma \} \\
& \llbracket Zero \nabla \\
& \quad (\lambda(x, r). \llbracket (SuccZero) \nabla (Succ \circ exr), id, out_{L_A} \rrbracket r), \\
& \quad id, out_{L_A} \rrbracket
\end{aligned}$$

By inlining the definition of hylomorphism, we get the following recursive definition:

$$\begin{aligned}
length \circ rev &= h \\
\text{where } h \text{ Nil} &= 0 \\
h \text{ Cons}(x, xs) &= g \text{ xs} \\
\text{where } g \text{ Nil} &= 1 \\
g \text{ Cons}(y, ys) &= 1 + (g \text{ ys})
\end{aligned}$$

Note that the intermediate list generated by  $rev$  has been eliminated.

It is clear from this example that  $rev$  need not any exceptional treatment in our method. By working in function level uniformly, our method wins this advantage over the normalization algorithm in [SF93].

### 4.3 Transformation Algorithm

The reduction strategy to control the order of application of the rules (Cata-HyloFusion and Hylo-AnaFusion) defines our transformation algorithm.

Note that Cata-HyloFusion does not change the compositional interface to the right: the third parameter of the right hylomorphism remains unchanged as the third parameter of the resultant hylomorphism. Dually, Hylo-AnaFusion does not change the compositional interface to the left.

We call the redexes of each rules *Cata-Hylo redex* and *Hylo-Ana redex* correspondingly. Because there are two kinds of redexes, four different cases of overlapping redexes exist:

1. Two Cata-Hylo redexes overlap:

$$\llbracket \varphi, \eta_1, out_F \rrbracket \circ \llbracket \tau_1 in_F, \eta_2, out_G \rrbracket \circ \llbracket \tau_2 in_G, \eta_3, \psi \rrbracket$$

In this case the reduction of the left redex does not destroy the right one.

2. Two Hylo-Ana redexes overlap:

$$\llbracket \varphi, \eta_1, \sigma_1 out_F \rrbracket \circ \llbracket in_F, \eta_2, \sigma_2 out_G \rrbracket \circ \llbracket in_G, \eta_3, \psi \rrbracket$$

In this case the reduction of the right redex does not destroy the left one.

3. A Cata-Hylo redex (in the left) overlaps with a Hylo-Ana redex (in the right):

$$\llbracket \varphi, \eta_1, out_F \rrbracket \circ \llbracket \tau in_F, \eta_2, \sigma out_G \rrbracket \circ \llbracket in_G, \eta_3, \psi \rrbracket$$

In this case the reduction of either redex does not destroy the other redex.

4. A Hylo-Ana redex (in the left) overlaps with a Cata-Hylo redex (in the right):

$$\llbracket \varphi, \eta_1, \sigma out_F \rrbracket \circ \llbracket in_F, \eta_2, out_G \rrbracket \circ \llbracket \tau in_G, \eta_3, \psi \rrbracket$$

In this case the reduction of either redex does destroy the other redex.

This observation tells us that the series of overlapping redexes of the same kind (case 1 and 2 above) are sensitive to the reduction order.

**Definition 4.1 (Redex chain)** *A Cata-Hylo (Hylo-Ana) redex chain is the series of Cata-Hylo (Hylo-Ana) redexes with overlaps.*

The following reduction strategy defines our algorithm:

#### Definition 4.2 (Reduction order)

1. Find all maximal Cata-Hylo redex chains, and reduce each chain from left to right.
2. Find all maximal Hylo-Ana redex chains, and reduce each chain from right to left.
3. Simplify the inside of each hylomorphisms using reduction rules for basic functors and related combinators.
4. If there exists any redex for HyloFusion rules, return to step 1 and continue reduction.

The reduction rules used in step 3 is listed in the next section.

#### 4.4 Reduction Rules for Basic Functors

Following rules are used to reduce the functor during the transformation. These equations describe some of the properties of basic functors and related combinators.

$$\begin{aligned}
exl \circ (f \times g) &= f \circ exl \\
exr \circ (f \times g) &= g \circ exr \\
exl \circ (f \triangle g) &= f \\
exr \circ (f \triangle g) &= g \\
exl \triangle exr &= id \\
(f \times g) \circ (h \triangle j) &= (f \circ h) \triangle (g \circ j) \\
(f \times g) \circ (h \times j) &= (f \circ h) \times (g \circ j) \\
(f \triangle g) \circ h &= (f \circ h) \triangle (g \circ h) \\
(f+g) \circ inl &= inl \circ f \\
(f+g) \circ inr &= inr \circ g \\
(f \nabla g) \circ inl &= f \\
(f \nabla g) \circ inr &= g \\
inl \nabla inr &= id \\
(f \nabla g) \circ (h+j) &= (f \circ h) \nabla (g \circ j) \\
(f+g) \circ (h+j) &= (f \circ h) + (g \circ j) \\
f \circ (g \nabla h) &= (f \circ h) \nabla (g \circ h) \quad (\text{for strict } f)
\end{aligned}$$

#### 4.5 More Examples of Transformation

To demonstrate the power of our transformation, let's consider the following functions:

$$\begin{aligned}
zip &= \llbracket in_{L_A \times B}, (id+abide) \circ IsNilOr, out_{L_A} \times out_{L_B} \rrbracket \\
\text{where } abide &= (exl \times exl) \triangle (exr \times exr) \\
IsNilOr ((1, x), (1, y)) &= (1, (x, y)) \\
IsNilOr ((i, x), (j, y)) &= (0, (x, y)) \\
iterate f &= \llbracket in_{L_A}, id, inr \circ (id \triangle f) \rrbracket
\end{aligned}$$

Then the composition  $zip \circ ((iterate f) \times zip)$  is transformed as follows:

$$\begin{aligned}
&zip \circ ((iterate f) \times zip) \\
&= \{\text{definition of } zip \text{ and } iterate\} \\
&\llbracket in_{L_A \times (B \times C)}, (id+abide) \circ IsNilOr, out_{L_A} \times out_{L_B \times C} \rrbracket \\
&\quad \circ (\llbracket in_{L_A}, id, inr \circ (id \triangle f) \rrbracket \\
&\quad \quad \times \llbracket in_{L_B \times C}, (id+abide) \circ IsNilOr, out_{L_B} \times out_{L_C} \rrbracket) \\
&= \{\text{property of } \times\} \\
&\llbracket in_{L_A \times (B \times C)}, (id+abide) \circ IsNilOr, out_{L_A} \times out_{L_B \times C} \rrbracket \\
&\quad \circ \llbracket in_{L_A} \times in_{L_B \times C}, id \times ((id+abide) \circ IsNilOr), \\
&\quad \quad (inr \circ (id \triangle f)) \times (out_{L_B} \times out_{L_C}) \rrbracket \\
&= \{\text{HyloSplit}^{+1}\} \\
&\llbracket in_{L_A \times (B \times C)}, \\
&\quad (id+abide) \circ IsNilOr \circ (id \times ((id+abide) \circ IsNilOr)), \\
&\quad (inr \circ (id \triangle f)) \times (out_{L_B} \times out_{L_C}) \rrbracket
\end{aligned}$$

By inlining the definition of hylomorphisms, we get the familiar recursive definition:

$$\begin{aligned}
zip \circ ((iterate f) \times zip) &= h \\
\text{where } h(x, (Nil, zs)) &= Nil \\
h(x, (ys, Nil)) &= Nil \\
h(x, (ys, zs)) &= Cons((x, (hd ys, hd zs)), \\
&\quad h(f x, (tl ys, tl zs)))
\end{aligned}$$

In [GLPJ93] *zip* has been discussed to explain the most serious limitation of their method. It is clear from this example that our transformation has successfully lift their limitation: the both input lists to *zip* have been deforested.

## 5 Related Work

Deforestation was first proposed by Wadler in [Wad88] as an automatic transformation to remove unnecessary intermediate data structure. The class of programs his algorithm can treat is characterized as *treeless* program which is a subset of first-order programs. Based on the observation that some intermediate data structures of basic types (e.g. integers, characters, etc.) need not to be removed, Wadler developed the *blazing* technique to handle such terms. He also discusses to apply his method to some higher-order programs whose higher-order functions can be treated as macros. Our method works on much wider class of higher-order programs, and it need not expand to first-order forms. It is also easy to control what types of intermediate data structures are to be removed with our method.

The *fusion* transformation proposed by Chin ([Chi92]) generalizes deforestation to make it applicable to all first-order programs. Combining it with his higher-order removal technique, his algorithm can take any first-order and higher-order program as its input. Inspired by Wadler's blazing deforestation, Chin devised the double annotation scheme for safe fusion to recognize and skip over terms to which his techniques do not apply. Because his method basically annotates non-treeless subterms, the improvement to Wadler's method comes from the power of higher-order removal. Our method accepts the example of higher-order removal in his paper and successfully transforms it to

the same first-order program. Moreover the example program `size` (defined as `length ∘ flatten` on binary tree), which cannot be handled with Chin's method without assuming an extra law on `length` and `append`, naturally be deforested by our method without any extra laws.

In [FSZ94] Fegaras, Sheard and Zhou extend their normalization algorithm in [SF93] to the more general fold programs which recurse over multiple inductive structures simultaneously, such as `zip` or `nth`. Because our method always works on the function level and explicitly manipulates the functors, it is easy to give symmetric definitions to those functions like `zip`.

Sørensen ([Sør94]), applies a tree grammar-based data-flow analysis to put annotations on programs that guarantee termination of deforestation for the wider class of first-order programs than Chin's method. The grammar is used to approximate the set of terms that the deforestation algorithm encounters, and successfully locates the increasing (accumulating) parameters which could be the source of infinite unfolding.

Sørensen, Glück and Jones ([SGJ94]) pick up four different transformational methods (Partial Evaluation, Deforestation, Supercompilation and Generalized Partial Computation [FNT91, Tak91]) and discuss the difference of transformational power of each method. Because each method is defined for the different language in syntactic way, it is not easy to compare without losing the insights of each method. Calculation-based transformations provides the better device for such comparative study.

## Acknowledgments

We are grateful to Lambert Meertens, Kieran Cleaghan and Fer-Jan de Vries for providing encouragement and valuable feedback for this research. Many thanks also to Leonidas Fegaras, Zhenjiang Hu, Hideya Iwasaki and Masato Takeichi for their comments on the draft of this paper.

## References

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24(1):44–67, 1977.

[BM94] Richard S. Bird and Oege Moor. Relational program derivation and context-free language recognition. In A.W. Roscoe, editor, *A Classical Mind*, pages 17–35. Prentice Hall, 1994.

[Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In *ACM Conference on Lisp and Functional Programming, San Francisco, Ca.*, pages 11–20. ACM, June 1992.

[FNT91] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90:61–79, 1991.

[Fok92] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, February 1992.

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

[GLPJ93] A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *FPCA '93, Copenhagen*. ACM, June 1993.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, University of Utrecht, February 1993.

[Mee86] L. Meertens. Algorithmics – towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[Mei92] E. Meijer. *Calculating Compilers*. PhD thesis, University of Nijmegen, February 1992.

[Mei94] E. Meijer. Acid rain theorem. Submitted for publication, 1994.

[MFP91] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA '91, Boston, Ma.*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, August 1991.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *FPCA '93, Copenhagen*, pages 233–242. ACM, June 1993.

[SGJ94] M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and gpc. In *ESOP '94*, volume 788 of *LNCS*. Springer-Verlag, April 1994.

[Sør94] M.H. Sørensen. A grammar-based data-flow analysis to stop deforestation. In *CAAP '94*, volume 787 of *LNCS*. Springer-Verlag, April 1994.

[Tak91] A. Takano. Generalized partial computation for a lazy functional language. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut.*, pages 1–11. ACM, 1991.

[TNT82] V.F. Turchin, R.M. Nirenberg, and D.V. Turchin. Experiments with a supercompiler. In *ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania*, pages 47–55. ACM, 1982.

[Tur86] V.F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, July 1986.

[Wad88] P. Wadler. Deforestation. In *ESOP '88, Nancy, France*, volume 300 of *LNCS*. Springer-Verlag, March 1988.

[Wad89] P. Wadler. Theorems for free! In *FPCA '89, London*, ACM, September 1989.