A Calculational Fusion System HYLO

Y. Onoue[†], *Z. Hu*[†], *H. Iwasaki*[‡], *and M. Takeichi*[†]
†*Department of Information Engineering, University of Tokyo*
Email: {onoue,hu,takeichi}@ipl.t.u-tokyo.ac.jp
‡*Department of Computer Science*
Tokyo University of Agriculture and Technology
Email: iwasaki@ipl.ei.tuat.ac.jp

Abstract

Fusion, one of the most useful transformation tactics for deriving efficient programs, is the process whereby separate pieces of programs are fused into a single one, leading to an efficient program without intermediate data structures produced. In this paper, we report our on-going investigation on the design and implementation of an automatic transformation system HYLO which performs fusion transformation in a more systematic and more general way than any other systems. The distinguished point of our system is its *calculational* feature based on simple application of transformation laws rather than traditional search-based transformations.

Keywords

Program Calculation, Fusion, Bird-Meertens Formalisms

1 INTRODUCTION

Program transformation has been advocated as the linchpin of a programming paradigm in which the derivation of efficient programs from naive specification of problems is a formal and mechanically supported process (Pettorossi & Proietti 1993). It does not attempt to directly produce a program that is correct, understandable and efficient. Instead, it starts with a program (specification) which is as clear and understandable as possible ignoring any question of efficiency, and successively transforms it to more and more efficient versions based on a set of transformation *tactics*.

Fusion Transformation (or *Fusion* for short), one of such most useful transformation tactics for deriving efficient programs, is the process whereby separate pieces of programs are fused into a single one, typically transforming a multi-pass program into a single pass and thus leading to an efficient program without intermediate data structures. Recently, it has been gaining more and more interest in the functional programming community (Wadler 1988, Chin

©IFIP 1996. Published by Chapman & Hall

1992, Gill, Launchbury & Jones 1993, Takano & Meijer 1995, Hu, Iwasaki & Takeichi 1996*b*).

The purpose of this paper is to report our on-going investigation on the design and implementation of an automatic system HYLO which implements fusion transformation in a more systematic and general way than any other transformation systems. To explain our idea, consider the following example program (Wadler 1988): computing the sum of the squares of the integers from 1 to n.

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

It is defined as a composition of three recursive functions: *upto* generates a list of numbers from 1 to n, *map square* forms the second list by squaring each number in the first list, and *sum* calculates the sum of the numbers in the second list.

This program has the advantage of clarity and higher level of modularity. It constructs the program ssf by gluing three components: "generate the numbers," "square the numbers," and "calculate the sum," which are relatively simple, easy to write, and potentially reusable. However, it relies 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 (even by lazy evaluation), degrading execution time and space dreadfully. To obtain an efficient program, it is our hope that three component functions can be merged into a single one eliminating intermediate data structures. This is exactly what fusion aims to do.

There are mainly two kinds of approaches to fusion transformation: *search-based fusion* and *calculational fusion*. The traditional search-based fusion (Wadler 1988, Chin 1992) turns the composition of recursive functions into a simple function by *fold-unfold* transformations (Burstall & Darlington 1977). For our example, it unfolds recursive definitions of functions *upto*, *map square* and *sum* to some extent, manipulates the expression, and identifies suitable places where subexpressions are to be folded into corresponding function applications. It is called "search-based" because it basically has to keep track of all occurring function calls and introduce function definitions to be searched in the folding step. The process of keeping track of function calls and controlling the steps cleverly to avoid infinite unfolding introduces substantial cost and complexity, which prevents fusion from being practically implemented (Gill et al. 1993).

Our interest is in the calculational fusion (Sheard & Fegaras 1993, Gill et al. 1993, Takano & Meijer 1995, Launchbury & Sheard 1995, Hu et al. 1996*b*, Hu, Iwasaki & Takeichi 1996*c*), which is rather new and has been attracting more and more attention. Different from the search-based fusion whose emphasis is on the transformation process, its emphasis is on the ex-

ploration of recursive constructs in each component function so that fusion can be performed directly by applying a simple transformation law: the *Acid Rain Theorem* (Gill et al. 1993, Takano & Meijer 1995). Returning to our example program ssf, let us explain briefly how to fuse the composition part, $sum \circ map\,square \circ upto$, with this approach while leaving the detailed discussions for later.

1. *Deriving Hylomorphisms from Recursive Definitions*
   First of all, we need to rewrite every component recursive function in terms of a specific recursive form called *hylomorphism* (Section 2) represented by a triplet ϕ, η and ψ grouped with special brackets as $[\![\phi, \eta, \psi]\!]$. In fact, almost all recursive functions of interest can be captured by hylomorphisms (Bird & de Moor 1994), and we have proposed an algorithm for deriving hylomorphisms from recursive definitions of functions (Hu et al. 1996*b*). After deriving hylomorphisms for *sum*, *map square* and *upto*, we have the following compositional expression in which each hylomorphism corresponds to the component function:

   $$[\![\phi_1, \eta_1, \psi_1]\!] \circ [\![\phi_2, \eta_2, \psi_2]\!] \circ [\![\phi_3, \eta_3, \psi_3]\!].$$

2. *Capturing Data Production and Consumption Schemes*
   For a composition of two hylomorphisms, there is a useful law called *Acid Rain Theorem* (see Section 2) which says that a composition of two hylomorphisms can be fused into a single one under certain conditions. The theorem expects ϕ and ψ in $[\![\phi, \eta, \psi]\!]$ to be specified as $\tau\, in$ and $\sigma\, out$ respectively. Here, *in* denotes data constructors and *out* data destructors. Functions τ and σ are polymorphic and used to capture the scheme of production and consumption of data structures. For the application of the Acid Rain Theorem, we have to derive τ_i and σ_i for some ϕ_i and ψ_i respectively, giving the following expression:

   $$[\![\phi_1, \eta_1, \sigma_1\, out]\!] \circ [\![\tau_2\, in, \eta_2, \sigma_2\, out']\!] \circ [\![\tau_3\, in', \eta_3, \psi_3]\!]$$

3. *Applying Acid Rain Theorem*
   After the preparation by Steps 1 and 2, we can apply the Acid Rain Theorem for fusion. By fusing the latter composition of our example, the program becomes:

   $$[\![\phi_1, \eta_1, \sigma_1\, out]\!] \circ [\![\phi_2', \eta_2', \psi_2']\!].$$

   Repeating Step 2 and 3 to the above composition could yield a single hylomorphism

   $$[\![\phi_1', \eta_1', \psi_1']\!].$$

4. *Inlining Resulting Hylomorphism*
   Finally, we can inline the resulting hylomorphism into our familiar recursive definition as shown below, removing inefficiency due to hylomorphic structure.

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

   It is obviously more efficient than the original because all intermediate lists have been successfully eliminated.

The calculational approach has been argued to be more practical (Gill et al. 1993, Takano & Meijer 1995), but to the best of our knowledge, there are no practical fusion systems based on this approach. The difficulties in the design and implementation of a real fusion system lie in two aspects:

- *Development of Constructive Algorithms for Implementing Transformation Laws.* Program calculation, i.e., calculation with programs, is a kind of program transformation which proceeds by means of program manipulation based on a rich collection of transformation laws. These laws, however, are basically developed as a guidance to calculate with programs by hands (not by machine) and most of them are not constructive. Therefore, to implement an automatic calculational fusion system, we must develop *constructive algorithms* for implementing those non-constructive transformation laws (see Section 4.3).
- *Implementation Issues for Practical Use.* In the design and implementation of a practical fusion system, we have to take account of a lot of practical issues. In particular, we should focus on specification *language design*, which should not only be convenient to specify problems but also be general and sufficiently powerful, and *algorithm design*, which should be as much applicable as possible to of a considerable scale programs rather than just some toy examples.

In this paper, we report our on-going investigation on the design and implementation of an automatic system HYLO which performs fusion transformation based on calculational method. Our main contributions are summarized as follows. First, we made the first attempt to apply the idea of a calculational approach to the implementation of a transformation system, not just being limited to theoretical interest, which is in sharp contrast to previous search-based approaches. This work extends our previous work on developing algorithms for implementation of transformation laws. Second, we develop a

set of effective algorithms implementing the constructive fusion laws. Third, our HYLO system is a general system which can be applied to a wide class of programs; particularly, it is expected to be used in compilers of functional languages such as Haskell.

The organization of this paper is as follows. After introducing some notations and theoretical results on hylomorphisms in Section 2, we give an overview of the HYLO system in Section 3. Implementation algorithms are presented in Section 4 with transformation examples, and experimental results are given in Section 5. Section 6 gives remarks on the HYLO system and related work.

2 PRELIMINARIES

Recursion plays an important role in function definitions but it does not impose much structure on the form of the definition that may be expressed. Recently, many studies (Meijer, Fokkinga & Paterson 1991, Sheard & Fegaras 1993) have shown that recursion should be structured with some specific forms. The hylomorphism is one such specific recursive form which can describe almost all recursive functions of interest (Meijer et al. 1991, Bird & de Moor 1994, Hu et al. 1996*b*). Informally, by hylomorphism, a function should be defined in the following recursive way:

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

We can read the right hand side in this way: generating some F-structure from the input by ψ; applying f to all recursive components in the F-structure by $F\, f$; manipulating the F-structure into some G-structure by η; and finaly folding the G-structure by ϕ to give the result. Since f can be uniquely determined whenever ϕ, η, ψ, G and F are determined, we usually denote f by $f = [\![\phi, \eta, \psi]\!]_{G,F}$.

Hylomorphisms enjoy many useful calculational laws (see Section 2.3), facilitating program transformation. To be more precise, we shall review previous works on program calculation (Meijer et al. 1991, Fokkinga 1992, Takano & Meijer 1995).

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*.

- The **identity** functor I on type X and its operation on functions are defined as follows.

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

- The **constant** functor $!A$ on type X and its operation on functions are defined as follows.

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

  where id stands for the identity function.

- The **product** $X \times Y$ of two types X and Y and its operation to functions are defined as follows.

  $$\begin{array}{lcl} X \times Y & = & \{(x,y) \mid x \in X,\, y \in Y\} \\ (f \times g)\ (x,y) & = & (f\ x,\, g\ y) \\ \pi_1\ (a,b) & = & a \\ \pi_2\ (a,b) & = & b \\ (f \mathbin{\triangle} g)\ a & = & (f\ a,\, g\ a) \end{array}$$

- The **separated sum** $X + Y$ of two types X and Y and its operation to functions are defined as follows.

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

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

2.2 Data Types as Initial Fixed Points of Functors

A data type is a collection of data constructors specifying 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 (Fokkinga 1992). 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 polynomial functor to capture the recursive structure of the data type:

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

where **1** denotes the final object, corresponding to (). Strictly speaking, Nil should be written as the 0-ary constructor $Nil\,()$. In this paper, the form of $t\,()$ will be simply denoted as t.

In fact, the definition of F_{L_A} can be automatically derived from the original definition of $List\ A$ (Sheard & Fegaras 1993). Besides, we define $in_{F_{L_A}}$, the *data constructor* of $List\ A$, by

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

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

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

As another example, the data type of the binary trees declared by

$$Tree\ a\ =\ Leaf \mid Node\ (a, Tree\ a, Tree\ a),$$

is captured by the following endofunctor:

$$F_T\ =\ !\mathbf{1}\ +\ !a \times I \times I.$$

In general, functor F determines a data type as its least fixed point, so we denote μF as the data type determined by F.

2.3 Hylomorphisms

The hylomorphism, a general recursive form covering the well-known *catamorphism* and *anamorphism* as its special cases, is defined in triplet form (Takano & Meijer 1995) as follows.

Definition 1 (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 $\eta : F \dot{\to} G$, 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 (Hu et al. 1996*b*). They are considered to be an ideal recursive form for calculating efficient functional programs. Note that we sometimes omit the subscripts G and F when they are clear from the context.

Hylomorphisms are quite general in that many useful forms are their special cases.

Definition 2 (Catamorphism $(\!|-|\!)$, Anamorphism $[\!(-)\!]$)

$$\begin{array}{lcl} (\!|\phi|\!)_F & = & [\![\phi, id, out_F]\!]_{F,F} \\ [\!(\psi)\!]_F & = & [\![in_F, id, \psi]\!]_{F,F} \end{array}$$ □

Hylomorphisms enjoy many useful transformation laws. One 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 the very useful property that a natural transformation can be shifted around inside a hylomorphism. Another useful law is the *Acid Rain Theorem* (Takano & Meijer 1995), showing how to fuse a composition of two hylomorphisms into a single one.

Theorem 1 (Acid Rain)

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

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

3 OVERVIEW OF HYLO SYSTEM

Figure 1 shows a schematic overview of our HYLO transformation system. This system is written in a functional language Gofer (Jones 1994), and can easily be ported to other systems such as Haskell. The HYLO system receives a program written in Gofer and returns an improved one as its result where the production of intermediate data structures has been removed by means of fusion transformation.

The front-end of the system does several standard preprocesses such as parsing, renaming, type checking and de-sugaring before producing a program in the core language that will be used hereafter in the transformation steps.

The core language is shown in Figure 2. It is similar to that used by the GHC (Team 1996) system. The main difference between these two languages

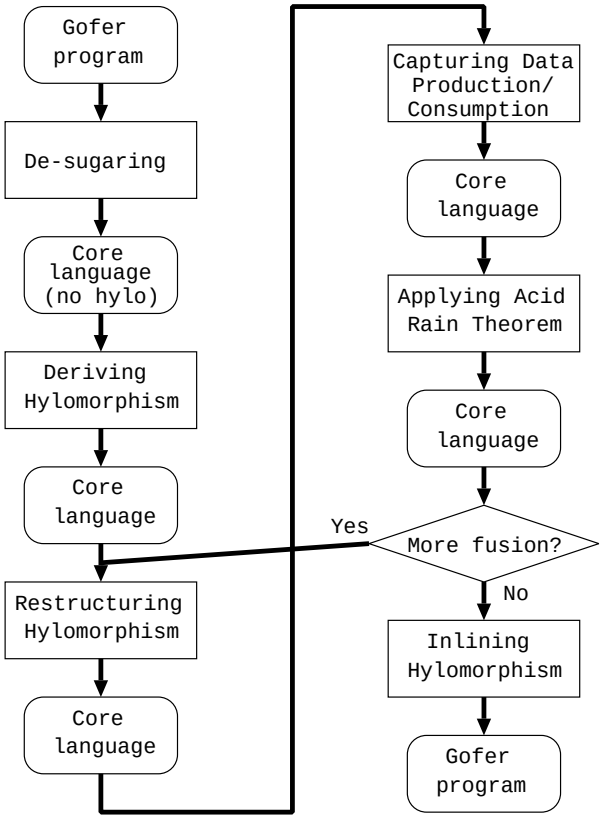


Figure 1 Overview of HYLO System

is that ours has an extra kind of expression for hylomorphism, which plays an important role in fusion transformation. In order to simplify our presentation, we restrict ourselves to single-recursive data types and functions without mutual recursions, since the standard tupling technique can transform mutual recursive definitions to non-mutual ones. Furthermore, we do not allow nested case expressions because they can easily be flattened.

The next step of our transformation process is to receive a program written in the core language and derive hylomorphisms from all the recursive function definitions. As we explained in Section 2.3, a hylomorphism is a representation of a specific form of recursive structure and is manipulable by applying the Acid Rain Theorem. In Section 4.1, we describe the algorithm for deriving hylomorphisms in detail.

We then restructure hylomorphisms to obtain proper hylo-structures suitable for fusion. As the Hylo Shift Law shows, we are allowed to shift some parts of the components of $[\![\phi, \eta, \psi]\!]$ among each other. Our algorithm extracts computations from ϕ and ψ and moves them to the η component. This

$Prog$	::=	$Def;\ Prog \mid Def$	Program
Def	::=	$v = t$	Definition (non-mutual)
t	::=	v	Variable
	|	l	Literal (Int,Float,Char,...)
	|	$(t_1, \ldots, t_n)$	Term tuple
	|	$\lambda v_s.t$	Lambda expression
	|	let $v = t_1$ in t_0	Let expression (non-recursive)
	|	case t_0 of $p_1 \to t_1; \cdots; p_n \to t_n$	Case expression
	|	$v\ t_1 \cdots t_n$	Function application (saturated)
	|	$C\ t_1 \cdots t_n$	Constructor application (saturated)
	|	$t_0\ t_1$	Application (general)
	|	$[\![t_\phi, t_\eta, t_\psi]\!]$	Hylo representation
	|	(n, t)	Expression with tag
v_s	::=	$v \mid (v_1, \cdots, v_n)$	Argument bounded by λ
p	::=	v	Variable
	|	$(p_1, \ldots, p_n)$	Tuple
	|	$C\ p_1 \cdots p_n$	Constructor pattern

Figure 2 HYLO Core Language

process simplifies the ϕ and ψ components to make the next step of transformation easier. In Section 4.2, we fully explain the algorithm for restructuring hylomorphisms.

What we should do next is to prepare for applying the Acid Rain Theorem in order to proceed fusion transformation. We may consider that ϕ and ψ express data production and consumption respectively. For our goal, ϕ and ψ should be converted to the specific forms of τin_F and σout_F for the Acid Rain Theorem to be applicable. The algorithms are presented in Section 4.3.

Now we are ready to apply the Acid Rain Theorem to the result, fusing two hylomorphisms into a single one. This process terminates even if we apply the theorem repeatedly. If there are no fusible hylomorphisms left in the program, we proceed to the next step for inlining. Otherwise, we go back to the restructuring step. In Section 4.4, we describe how the fusion is put into action.

Finally, the hylomorphism representation is inlined into our familiar recursive definitions to get a Gofer program. This inlining process plays the role of the back-end of the system. If we replace the front-end and back-end with those of the practical compiler system, we have a compiler with hylo-fusion optimization.

4 IMPLEMENTATION ALGORITHMS

In this section, we propose several important algorithms for the implementation of the HYLO system. Particularly we shall focus on the algorithms for

deriving polymorphic functions for capturing data consumption and construction. The algorithms for deriving and restructuring hylomorphisms are some extensions of those already discussed in (Hu et al. 1996*b*).

4.1 Deriving Hylomorphisms

We extend the algorithm in (Hu et al. 1996*b*) to derive hylomorphisms from recursive definitions, enabling it to be applied to programs in the core language, more general than the previous one with several new expression constructs. Our algorithm, as summarized in Figure 3, basically follows the same thought as in (Hu et al. 1996*b*). We shall omit any detailed description here, but give an informal explanation for understanding the derivation process. The calculation below shows how we can turn the typical recursive definition of function f in pattern matching style:

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

into a hylomorphism.

$$\begin{array}{lll}
f & = & \{\text{Definition of } f\} \\
 & & \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to t_1; \cdots; p_n \to t_n \\
 & = & \{\text{Trick 1: Replacing } t_i \text{ with } g_i\ t_i'\} \\
 & & \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to g_1\ t_1'; \cdots; p_n \to g_n\ t_n' \\
 & = & \{\text{Using separated sum}\} \\
 & & (g_1 \triangledown \cdots \triangledown g_n) \circ (\lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1,\ t_1'); \cdots; p_n \to (\mathsf{n},\ t_n')) \\
 & = & \{\text{Trick 2: Replacing } g_i \text{ with } \phi_i \circ F_i\ f\} \\
 & & (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ ((F_1 + \cdots + F_n)\ f) \circ \\
 & & \quad (\lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1,\ t_1'); \cdots; p_n \to (\mathsf{n},\ t_n')) \\
 & = & \{\text{Auxiliary definitions } \phi, F, \text{and } \psi\} \\
 & & \phi \circ F\ f \circ \psi \\
 & = & \{\text{Definition of hylomorphism}\} \\
 & & [\![\phi, id, \psi]\!]_{F,F} \\
 & & \text{where } F = F_1 + \cdots + F_n \\
 & & \qquad \phi = \phi_1 \triangledown \cdots \triangledown \phi_n \\
 & & \qquad \psi = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (1,\ t_1'); \cdots; p_n \to (\mathsf{n},\ t_n')
\end{array}$$

Tricks 1 and 2 suggest that if we can find ϕ_i, F_i, t_i' satisfying $t_i = (\phi_i \circ F_i\ f)\ t_i'$ for each t_i, a hylomorphism can be derived from the definition. This derivation heavily depends on Algorithm $\mathcal{D}$ which processes on each term t_i and returns a triple: a set of free variables in t_i but not in any recursive call to f, a set of pairs of a fresh variable and an argument of f in t_i, and a term from t_i

$$
\begin{array}{l}
\mathcal{A}[\![\lambda v_{s_1} \cdots \lambda v_{s_m}.\ \text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!]\, f \\
\quad = \lambda v_{s_1} \cdots \lambda v_{s_{m-1}}.\, [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F} \\
\qquad \textbf{where } (\{v_{i_1}, \ldots, v_{i_{k_i}}\}, \{(v'_{i_1}, t_{i_1}), \cdots, (v'_{i_{l_i}}, t_{i_{l_i}})\}, t'_i) = \mathcal{D}[\![t_i]\!]\, \{\}\, f \ \ (i = 1, \ldots, n) \\
\qquad\quad \phi_i = \lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ t'_i \\
\qquad\quad t''_i = (v_{i_1}, \ldots, v_{i_{k_i}}, t_{i_1}, \ldots, t_{i_{l_i}}) \\
\qquad\quad \psi = \lambda v_{s_m}.\text{case } t_0 \text{ of } p_1 \to (1, t''_1); \cdots; p_n \to (\mathsf{n}, t''_n) \\
\qquad\quad F_i = !\mathbf{1}, \text{ if } k_i = l_i = 0 \\
\qquad\qquad = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} \ \ (I_1 = \cdots = I_{l_i} = I), \text{otherwise} \\
\qquad\quad F = F_1 + \cdots + F_n \\
\qquad\quad \Gamma(v) = \text{return } v\text{'s type}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}[\![v]\!]\, s_l\, f & = & \text{if } v \in \mathit{global_vars} \cup s_l \text{ then } (\{\}, \{\}, v) \text{ else } (\{v\}, \{\}, v) \\
\mathcal{D}[\![l]\!]\, s_l\, f & = & (\{\}, \{\}, l) \\
\mathcal{D}[\![(t_1, \ldots, t_n)]\!]\, s_l\, f & = & (s_1 \cup \cdots \cup s_n, c_1 \cup \cdots \cup c_n, (t'_1, \ldots, t'_n)) \\
 & & \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f \ \ (i = 1, \ldots, n) \\
\mathcal{D}[\![\lambda v_s.t]\!]\, s_l\, f & = & (s, c, \lambda v_s.t') \\
 & & \text{where } (s, c, t') = \mathcal{D}[\![t]\!]\, (s_l \cup Var(v_s))\, f \\
\mathcal{D}[\![\text{let } v = t_1 \text{ in } t_0]\!]\, s_l\, f & = & (s_0 \cup s_1, c_0 \cup c_1, \text{let } v = t'_1 \text{ in } t'_0) \\
 & & \text{where } (s_1, c_1, t'_1) = \mathcal{D}[\![t_1]\!]\, s_l\, f, (s_0, c_0, t'_0) = \mathcal{D}[\![t_0]\!]\, (s_l \cup \{v\})\, f \\
\mathcal{D}[\![\text{case } t_0 \text{ of } p_1 \to t_1 & = & (s_0 \cup \cdots \cup s_n, c_0 \cup \cdots \cup c_n, \text{case } t'_0 \text{ of } p_1 \to t'_1; \cdots; p_n \to t'_n) \\
\quad ; \cdots; p_n \to t_n]\!]\, s_l\, f & & \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!](s_l \cup Var(p_i))f \ (i = 0, \ldots, n), p_0 = () \\
\mathcal{D}[\![v\ t_1 \cdots t_n]\!]\, s_l\, f & = & \text{if } v = f \text{ then} \\
 & & \quad \text{if } n = m \ \wedge \ t_i = v_{s_i} \ (1 \le \forall i \le m-1) \\
 & & \quad \text{then } (\{\}, \{(u, t_m)\}, u) \\
 & & \quad \text{else } \mathit{error}\ (f \text{ should have saturated args and induct on last}) \\
 & & \text{else } (s_0 \cup \cdots \cup s_n, c_0 \cup \cdots \cup c_n, t'_0\ t'_1 \cdots t'_n) \\
 & & \text{where } (s_0, c_0, t'_0) = \mathcal{D}[\![v]\!]\, s_l\, f \\
 & & \qquad (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f, (i = 1, ..., n) \\
 & & \qquad u \text{ is a fresh variable} \\
\mathcal{D}[\![C\ t_1 \cdots t_n]\!]\, s_l\, f & = & (s_1 \cup \cdots \cup s_n, c_1 \cup \cdots \cup c_n, C\ t'_1 \cdots t'_n) \\
 & & \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f \ \ (i = 1, \ldots, n) \\
\mathcal{D}[\![t_0\ t_1]\!]\, s_l\, f & = & (s_0 \cup s_1, c_0 \cup c_1, t'_0\ t'_1)) \\
 & & \text{where } (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\, s_l\, f \ \ (i = 1, 2) \\
 & & \\
Var(x) & = & \text{set of variables in } x
\end{array}
$$

Figure 3 Deriving Hylomorphism

with recursive calls to f being replaced by the corresponding fresh variables. It is worth noting that we require, without loss of generality, that function f inducts over its last argument, while the other arguments remain unchanged in every recursive call to f. Note also that Algorithm $\mathcal{A}$ is applied to programs after the renaming process has been done, so all bound variables have been assigned unique names.

We demonstrate how this algorithm works on the example program ssf in Section 1. ssf uses functions sum, map, and $upto$ which are defined in pattern matching way as:

$$\begin{array}{lcl} sum & = & \lambda xs.\ \text{case}\ xs\ \text{of}\ Nil \to 0 \\ & & \qquad\qquad Cons\ (a, as) \to plus\ (a, sum\ as) \\ map & = & \lambda g.\ \lambda xs.\text{case}\ xs\ \text{of}\ Nil \to Nil \\ & & \qquad\qquad Cons\ (a, as) \to Cons\ (g\ a,\ map\ g\ as) \\ upto & = & \lambda(m, n).\ \text{case}\ (m > n)\ \text{of}\ True \to Nil \\ & & \qquad\qquad False \to Cons\ (m, upto\ (m + 1, n)). \end{array}$$

We pick up sum from these definitions, and apply Algorithm $\mathcal{A}$ which in turn calls Algorithm $\mathcal{D}$. We have

$$\begin{array}{rcl} \mathcal{D}[\![0]\!]\ \{\}\ sum & = & (\{\}, \{\}, 0) \\ \mathcal{D}[\![plus\ (a, sum\ as)]\!]\ \{\}\ sum & = & (\{a\}, \{(v_1', sum\ as)\}, plus\ (a, v_1')) \end{array}$$

with

$$\begin{array}{ll} t_1' = (), & \phi_1 = \lambda().\ 0 = 0 \\ t_2' = (a, as), & \phi_2 = \lambda(a, v_1').\ plus\ (a, v_1') = plus \end{array}$$

and

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

So we have derived the following hylomorphism for sum:

$$sum \quad = \quad [\![0 \triangledown plus, id, out_F]\!]_{F,F}.$$

Similarly we can derive hylomorphisms for map and $upto$ as follows.

$$\begin{array}{lcl} map & = & \lambda g.\ [\![\phi, id, out_F]\!]_{F,F} \\ & & \text{where}\ \phi = \lambda().\ Nil \triangledown \lambda(v, v').\ Cons\ (g\ v, v') \\ upto & = & [\![in_F, id, \psi]\!]_{F,F} \\ & & \text{where}\ \psi = \lambda(m, n).\ \text{case}\ (m > n)\ \text{of} \\ & & \qquad\qquad True \to (1, ()) \\ & & \qquad\qquad False \to (2, (m, (m + 1, n))) \end{array}$$

where

$$\begin{array}{lcl} F & = & !\mathbf{1} + !Int \times I \ (= F_{L_{Int}}) \\ in_F & = & Nil \triangledown Cons \\ out_F & = & \lambda xs.\ \text{case}\ xs\ \text{of}\ Nil \to (1, ()) \\ & & \qquad\qquad\qquad Cons\ (a, as) \to (2, (a, as)) \end{array}$$

4.2 Restructuring Hylomorphisms

Generally, the behavior of a hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ could be understood as follows: ψ generates a recursive structure, η transforms one structure into another, and ϕ manipulates on the resulting recursive structure. It is possible that the ϕ and ψ contains the computation which η can do. The aim of restructuring is to extract as much computation as possible from ϕ and ψ and move it to η. This simplifies ϕ and ψ and makes the derivation of the data production and consumption schemes τ and σ easier in the next stage. A basic restructuring algorithm for the ϕ component is given in (Hu et al. 1996*b*) and an extended version of the algorithm is shown in Figure 4. The restructuring algorithm for the ψ component is shown in Figure 5, which was omitted in (Hu et al. 1996*b*).

Algorithm $\mathcal{S}_\phi$ to restructure $\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$ applies an auxiliary algorithm $\mathcal{E}$ to the bodies t_i of lambda expressions ϕ_i. Algorithm $\mathcal{E}$ detects maximal subterms in each t_i without recursive variables and generates a new term t_i' with these subterms substituted by new variables, and $\mathcal{S}_\phi$ makes it as a body of a new lambda expression ϕ_i'. In more detail, the algorithm $\mathcal{E}$ accepts as input a term t_i and a set of *recursive* variables and returns as result a pair: a set of maximal subterms containing no recursive variables with corresponding fresh variables, and a new term from t_i with these subterms substituted by corresponding fresh variables. The computations related to these subterms are then moved to the η component.

As an example, we will restructure the following hylomorphism obtained in the previous section.

$$map\ g = [\![Nil \triangledown \lambda(a, v_1').\ Cons\ (g\ a, v_1'), id, out]\!]$$

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

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

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

where

$G_1 + \cdots + G_n = G$

$!\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i} = G_i$

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

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

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

$G'_i = !\Gamma(u_{i_1}) \times \cdots \times !\Gamma(u_{i_{m_i}}) \times I_1 \times \cdots \times I_{l_i}$

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

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

Assume that u is a fresh variable in the following:

$$\begin{aligned}
\mathcal{E}[\![v]\!]\,s_r &= \text{if } Var_{s_r}(v) \text{ then } (\{(u,v)\}, u) \text{ else } (\{\}, v)\\
\mathcal{E}[\![l]\!]\,s_r &= (\{(u,l)\}, u)\\
\mathcal{E}[\![(t_1, \ldots, t_n)]\!]\,s_r &= \text{if } \forall i.Var_{s_r}(t'_i) \text{ then } (\{(u,(t_1,\ldots,t_n))\}, u)\\
&\qquad \text{else } (w_1 \cup \cdots \cup w_n, (t'_1, \ldots, t'_n))\\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\,s_r \ (i = 1, \ldots, n),\\
\mathcal{E}[\![\lambda v.t]\!]\,s_r &= \text{if } Var_{s_r}(t') \text{ then } (\{(u,\ \lambda v.t)\}, u) \text{ else } (w, \lambda v.t')\\
&\quad \text{where } (w, t') = \mathcal{E}[\![t]\!]\,s_r\\
\mathcal{E}[\![\text{let } v = t_1 \text{ in } t_0]\!]\,s_r &= \text{if } Var_{s_r}(t'_0) \wedge Var_{s_r}(t'_1) \text{ then } (\{(u, \text{let } v = t_1 \text{ in } t_0)\}, u)\\
&\qquad \text{else } (w_0 \cup w_1, \text{let } v = t'_1 \text{ in } t'_0)\\
&\quad \text{where } (w_1, t'_1) = \mathcal{E}[\![t_1]\!]\,s_r,\ \ (w_0, t'_0) = \mathcal{E}[\![t_0]\!]\,s_r\\
\mathcal{E}[\![\text{case } t_0 \text{ of } p_1 \to t_1 &= \text{if } \forall i.Var_{s_r}(t'_i) \text{ then } (\{(u, \text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n)\}, u)\\
\quad ; \cdots ; p_n \to t_n]\!]\,s_r &\qquad \text{else } (w_0 \cup \cdots \cup w_n, \text{case } t'_0 \text{ of } p_1 \to t'_1; \cdots; p_n \to t'_n)\\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\,s_r \ (i = 0, \ldots, n),\\
\mathcal{E}[\![v\ t_1 \cdots t_n]\!]\,s_r &= \text{if } \forall i.Var_{s_r}(t'_i) \text{ then } (\{(u, v\ t_1 \cdots t_n)\}, u)\\
&\qquad \text{else } (w_0 \cup \cdots \cup w_n, t'_0\ t'_1 \cdots, t'_n)\\
&\quad \text{where } (w_0, t'_0) = \mathcal{E}[\![v]\!]\,s_r,\ \ (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\,s_r \ (i = 1, \ldots, n),\\
\mathcal{E}[\![C\ t_1 \cdots t_n]\!]\,s_r &= \text{if } \forall i.Var_{s_r}(t'_i) \text{ then } (\{(u, C\ t_1 \cdots t_n)\}, u)\\
&\qquad \text{else } (w_1 \cup \cdots \cup w_n, C\ t'_1 \cdots t'_n)\\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\,s_r \ (i = 1, \ldots, n),\\
\mathcal{E}[\![t_0\ t_1]\!]\,s_r &= \text{if } Var_{s_r}(t'_0) \wedge Var_{s_r}(t'_1) \text{ then } (\{(u, t_0\ t_1)\}, u)\\
&\qquad \text{else } (w_0 \cup w_1, t'_0\ t'_1))\\
&\quad \text{where } (w_0, t'_0) = \mathcal{E}[\![t_0]\!]\,s_r,\ \ (w_1, t'_1) = \mathcal{E}[\![t_1]\!]\,s_r\\
\mathcal{E}[\![[\![t_0, t_1, t_2]\!]_{F_0,F_1}]\!]\,s_r &= \text{if } \forall i.Var_{s_r}(t'_i) \text{ then } (\{(u, [\![t_0, t_1, t_2]\!]_{F_0,F_1})\}, u)\\
&\qquad \text{else } (w_0 \cup w_1 \cup w_2, [\![t'_0, t'_1, t'_2]\!]_{F_0,F_1}))\\
&\quad \text{where } (w_i, t'_i) = \mathcal{E}[\![t_i]\!]\,s_r \ (i = 0, 1, 2),\\
\mathcal{E}[\![(\mathsf{n}, t)]\!]\,s_r &= \text{if } Var_{s_r}(t') \text{ then } (\{(u, (\mathsf{n}, t))\}, u) \text{ else } (w, (\mathsf{n}, t'))\\
&\quad \text{where } (w, t') = \mathcal{E}[\![t]\!]\,s_r\\
\\
Var_{s_r}(t) &= t \text{ is a variable } \ \wedge \ \ t \notin s_r
\end{aligned}$$

Figure 4 Restructuring Hylomorphisms — ϕ part

$$
\begin{array}{l}
\mathcal{S}_\psi[\![[\![\phi,\eta,\psi]\!]_{G,F}]\!] = [\![\phi,\eta\circ(\eta_{\psi_1}+\cdots+\eta_{\psi_n}),\psi']\!]_{G,F'} \\
\quad \text{where} \\
\qquad \lambda\ v_s.\text{case}\ t_0\ \text{of}\ p_1 \to (\mathsf{1},t_1);\cdots;p_n \to (\mathsf{n},t_n) = \psi \\
\qquad F_1+\cdots+F_n = F \\
\qquad !\Gamma(t_{i_1})\times\cdots\times !\Gamma(t_{i_{k_i}})\times I_1\times\cdots\times I_{l_i} = F_i \\
\qquad (t_{i_1},\ldots,t_{i_{k_i}},tt_{i_1},\ldots,tt_{i_{l_i}}) = t_i \\
\qquad \{v_{i_1},\ldots,v_{i_{m_i}}\} = \mathcal{FV}[\![(t_{i_1},\ldots,t_{i_{k_i}})]\!]\ global_vars \\
\qquad \eta_{\psi_i} = \lambda(v_{i_1},\ldots,v_{i_{m_i}},v'_{i_1},\ldots,v'_{i_{l_i}}).(t_{i_1},\ldots,t_{i_{k_i}},v'_{i_1},\ldots,v'_{i_{l_i}}) \\
\qquad t'_i = (v_{i_1},\ldots,v_{i_{m_i}},tt_{i_1},\ldots,tt_{i_{l_i}}) \\
\qquad F'_i = !\Gamma(v_{i_1})\times\cdots\times !\Gamma(v_{i_{m_i}})\times I_1\times\cdots\times I_{l_i} \\
\qquad F' = F'_1+\cdots+F'_n \\
\qquad \psi' = \lambda\ v_s.\text{case}\ t_0\ \text{of}\ p_1 \to (\mathsf{1},t'_1);\cdots;p_n \to (\mathsf{n},t'_n)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{FV}[\![v]\!]\,s_g &=& \text{if } v\in s_g \text{ then } \{\} \text{ else } \{v\} \\
\mathcal{FV}[\![l]\!]\,s_g &=& \{\} \\
\mathcal{FV}[\![(t_1,\ldots,t_n)]\!]\,s_g &=& s_1\cup\cdots\cup s_n \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\,s_g\ (i=1,\ldots,n) \\
\mathcal{FV}[\![\lambda v.t]\!]\,s_g &=& \mathcal{FV}[\![t]\!]\,(s_g\cup\{v\}) \\
\mathcal{FV}[\![\text{let } v=t_1 \text{ in } t_0]\!]\,s_g &=& s_0\cup s_1 \\
 & & \text{where } s_1 = \mathcal{FV}[\![t_1]\!]\,s_g,\ \ s_t = \mathcal{FV}[\![t_0]\!]\,(s_g\cup\{v\}) \\
\mathcal{FV}[\![\text{case } t_0 \text{ of } p_1\to t_1 & = & s_0\cup\cdots\cup s_n \\
;\cdots;p_n\to t_n]\!]\,s_g & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\,(s_g\cup Pat(p_i))\ (i=0,\ldots,n),\ \ p_0=() \\
\mathcal{FV}[\![v\ t_1\cdots t_n]\!]\,s_g &=& s_0\cup\cdots\cup s_n \\
 & & \text{where } s_0 = \mathcal{FV}[\![v]\!]\,s_g,\ \ s_i = \mathcal{FV}[\![t_i]\!]\,s_g\ (i=1,\ldots,n) \\
\mathcal{FV}[\![C\ t_1\cdots t_n]\!]\,s_g &=& s_1\cup\cdots\cup s_n \\
 & & \text{where } s_i = \mathcal{FV}[\![t_i]\!]\,s_g\ (i=1,\ldots,n) \\
\mathcal{FV}[\![t_0\ t_1]\!]\,s_g &=& s_0\cup s_1 \\
 & & \text{where } s_0 = \mathcal{FV}[\![t_0]\!]\,s_g,\ \ s_1 = \mathcal{FV}[\![t_1]\!]\,s_g \\
\mathcal{FV}[\![[\![t_0,t_1,t_2]\!]_{F_0,F_1}]\!]\,s_g &=& s_0\cup s_1\cup s_2 \\
 & & \text{where } s_0 = \mathcal{FV}[\![t_0]\!]\,s_g,\ \ s_1 = \mathcal{FV}[\![t_1]\!]\,s_g,\ \ s_2 = \mathcal{FV}[\![t_2]\!]\,s_g \\
\mathcal{FV}[\![(\mathsf{n},t)]\!]\,s_g &=& \mathcal{FV}[\![t]\!]\,s_g
\end{array}
$$

Figure 5 Restructuring Hylomorphisms — ψ part

and finally give the following structural hylomorphism:

$$map\ g = [\![Nil \triangledown \lambda(u_{2_1},v'_1).\ Cons(u_{2_1},v'_1), id + \lambda(a,v'_1).(g\,a,v'_1), out]\!]$$

This transformation simplifies ϕ of $(map\ g)$ into $Nil \triangledown \lambda(u_{2_1},v'_1).\ Cons(u_{2_1},v'_1)$ (i.e., in) making it easier to apply the Acid Rain Theorem.

Dually, Algorithm $\mathcal{S}_\psi$ restructures ψ in a typical general form:

$$\psi = \lambda v_s.\ \text{case}\ t_0\ \text{of}\ p_1 \to (\mathsf{1},t_1);\cdots;p_n \to (\mathsf{n},t_n).$$

It extracts free variables from $t_{i_1},\cdots,t_{i_{k_i}}$, constituents of t_i's, which are not

related to recursive parts (corresponding to the identity functor I). New terms t'_i work only as suppliers of free variables, and actual computation is performed by the generated η_{ψ_i}. Shifting η_{ψ_i} to the η component removes many computations from ψ.

4.3 Capturing Schemes of Data Production and Consumption

In order to apply the Acid Rain Theorem for fusion, it is expected that ϕ and ψ in a given hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F} : A \to B$ are described as $\tau\, in_{F_B}$ and $\sigma\, out_{F_A}$ respectively. Here, τ and σ are polymorphic functions and F_A and F_B are functors defining types A and B respectively. Our laws for deriving such τ and σ are as follows, whose proof can be found in (Hu et al. 1996*b*).

Theorem 2 (Deriving Polymorphic Functions)
Under the above conditions, τ and σ are defined by the following two laws.

$$\frac{\forall \alpha.\ \ (\![\alpha]\!)_{F_B} \circ \phi = \phi' \circ G\,(\![\alpha]\!)_{F_B}}{\tau = \lambda \alpha.\ \phi'}, \qquad \frac{\forall \beta.\ \ \psi \circ [\![\beta]\!]_{F_A} = F\,[\![\beta]\!]_{F_A} \circ \psi'}{\sigma = \lambda \beta.\ \psi'} \qquad \square$$

This theorem, though being general, does not show clearly how to find ϕ' or ψ' in a constructive way (i.e., by an algorithm). Therefore, to implement an automatic calculational fusion system, we must develop *constructive algorithms* for implementing those non-constructive transformation laws. In this section, we shall propose such basic algorithms; discussing in detail the derivation of τ and making brief remarks on the dual derivation of σ.

(a) Algorithm for Deriving τ

When a composition of two hylomorphisms $[\![_, _, out_F]\!]_{_,F} \circ [\![\phi, _, _]\!]_{F',_}$ is to be fused by the Acid Rain Theorem, Theorem 1 (a), ϕ must be described by a polymorphic function τ, a function abstracting over data constructors, satisfying $\tau\, in_F = \phi$. By typing, we know that ϕ should be $F'\mu F \to \mu F$, meaning that ϕ receives a F'-structure data whose recursive components are of type μF, and then assembles some data of type μF. Our idea of deriving τ from ϕ is thus to detect the data constructors of μF and abstract them.

To be precise, let $C_1, \cdots, C_n$ be the data constructors of μF, namely $in_F = C_1 \triangledown \cdots \triangledown C_n$, and let $c_1, \cdots, c_n$ be lambda variables used in abstracting $C_1, \cdots, C_n$. We shall derive τ by lambda abstraction of $C_1, \cdots, C_n$ in ϕ and then rewrite ϕ by τ applied to in_F, i.e., $\phi = \tau\, in_F$, where $\tau = \lambda(c_1 \triangledown \cdots \triangledown c_n).\ \phi'$ meeting Theorem 2.

For the sake of simple presentation, we shall assume without loss of gen-

erality that ϕ for which we are going to derive τ is in the following specific form:

$$\phi_1 \triangledown \cdots \triangledown \phi_k : (F'_1 + \cdots + F'_k)\mu F \to \mu F,$$

and each $\phi_i : F'_i \mu F \to \mu F$ is a lambda abstraction:

$$\phi_i = \lambda(v_{i_1}, \cdots, v_{i_{m_i}}\ v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t_i \tag{1}$$

in which $F'_i = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{m_i}}) \times I_1 \times \cdots \times I_{l_i}$, i.e., identity functors after constant functors. Here, the lambda variables corresponding to the identity functor I are usually called *recursive variables* and are explicitly attached with a prime ($'$). Clearly, if we can derive τ_i for each ϕ_i, it soon follows the τ for ϕ will be $\tau = \lambda cs.\ (\tau_1\ cs\ \triangledown\ \cdots\ \triangledown\ \tau_k\ cs)$. Therefore, our attention turns to the derivation of τ_i from ϕ_i defined by Equation (1). As we know that not every ϕ guarantees existence of τ, some restriction on ϕ becomes necessary. We have two requirements for choosing proper restriction: one is that it should be automatically checkable whether a given ϕ meets the restriction; another is that it should not seriously limit the descriptive power. We borrow a similar idea of *Canonical Terms* in (Sheard & Fegaras 1993) and impose this restriction on ϕ_i in which the term t_i has one of the following forms:

1. *a variable*: v, bounded in the lambda abstraction;
2. *a constructor application*: $C\ t'_1\ \cdots\ t'_n$, where C is a data constructor and each t'_i is a term;
3. *a hylo application*: $[\![\phi'_1 \triangledown \cdots \triangledown \phi'_{n'}, \eta, out]\!]\ v'$, where ϕ'_i is in our restrictive form and v' is a recursive variable;
4. *a function application*: $f\ t_1\ \cdots\ t_n$, where f is a global function and each term t_i has no reference to any recursive variable.

It is worth noting that our ϕ_i, though looking rather restrictive, is the "core" to which a more general ϕ_i can be transformed. First, the restructuring algorithm in Section 4.2 is a great help, because we simplifies ϕ a lot by moving out many computations not relating to the construction of the resultant data of type μF. In fact, after restructuring, our restrictive term should not include the last form (a function application) because the terms of this form will be extracted and moved to the η component. Second, some constructs like *let*, *case* and application of non-recursive functions need not to be considered because they are removable by preprocessing; local declarations can be lifted to be global, *case* structures can be embedded in hylomorphisms, and application of non-recursive functions can be removed by unfolding. Third, not surprising but convincing, all potentially normalization programs, which have reasonable descriptive power, can be automatically turned into this form (Sheard & Fegaras 1993).

Assume: $F = F_1 + \cdots + F_n$, $in_F = C_1 \triangledown \cdots \triangledown C_n$ and $c_1, \cdots, c_n$ are fresh variables.

$\phi \quad : \quad F'\mu F \to \mu F$
$\phi \quad \Rightarrow \quad \tau in_F \quad$ where $\tau = \mathcal{F}[\![\phi]\!]$

where

$$\begin{aligned}
\mathcal{F}[\![\phi_1 \triangledown \cdots \triangledown \phi_k]\!] &= \lambda cs.\ (\mathcal{F}[\![\phi_1]\!]\ cs \triangledown \cdots \triangledown \mathcal{F}[\![\phi_k]\!]\ cs) \\
\mathcal{F}[\![\lambda(v_{i_1}, \cdots, v_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\ t_i]\!] & \\
&= \lambda(c_1 \triangledown \cdots \triangledown c_n).\lambda(v_{i_1}, \cdots, v_{i_{m_i}}, v'_{i_1}, \cdots, v'_{i_{l_i}}).\mathcal{F}'[\![t_i]\!] \\
\\
\mathcal{F}'[\![v]\!] &= (\![c_1 \triangledown \cdots \triangledown c_n]\!)_F\ v \quad (*\ v \text{ is not a recursive variable } *) \\
\mathcal{F}'[\![v']\!] &= v' \qquad\qquad\qquad\quad (*\ v' \text{ is a recursive variable } *) \\
\mathcal{F}'[C_i\ t'_1\ \cdots\ t'_{n'}]\!] &= c_i(F_i\ \mathcal{F}'\ (t'_1, \cdots, t'_{n'})) \\
\mathcal{F}'[\![[\![\phi', \eta, out_F]\!]_{F',F}\ v']\!] &= [\![\mathcal{F}[\![\phi']\!]\ (c_1 \triangledown \cdots \triangledown c_n), \eta, out_F]\!]_{F',F}\ v' \\
\mathcal{F}'[\![f\ t'_1\ \cdots\ t'_{n'}]\!] &= (\![c_1 \triangledown \cdots \triangledown c_n]\!)\ (f\ t'_1\ \cdots\ t'_{n'})
\end{aligned}$$

Figure 6 Deriving τ

With the help of these restrictions, our algorithm becomes simpler and the proof of the correctness of the algorithm becomes easier. Now we propose our algorithm $\mathcal{F}'$ to derive τ_i from ϕ_i:

$$\phi_i = \tau_i in_F, \quad \text{where } \tau_i = \lambda(c_1 \triangledown \cdots \triangledown c_n).\ \lambda(v_{i_1}, \cdots, v_{i_{m_i}}\ v'_{i_1}, \cdots, v'_{i_{l_i}}).\ \mathcal{F}'[\![t_i]\!]$$

This lambda abstraction substitutes appropriate occurrences of each constructor of μF in t_i by its corresponding lambda variable. The algorithm $\mathcal{F}'$ is given in Figure 6, performing one of the following processes corresponding to the kind of t_i. Notice that all expressions which $\mathcal{F}'$ are applied to must have the type of μF.

1. When t_i is a variable, do nothing if it is a recursive variable. Otherwise, as it may be bound to an expression that includes μF constructors, we must proceed by applying the catamorphism $(\![c_1 \triangledown \cdots \triangledown c_n]\!)_F$ to the expression to replace each constructor of μF by the lambda variable c_i.
2. When t_i is a constructor application $C_i\ t'_1 \cdots t'_{n'}$, C_i has to be substituted by the corresponding lambda variable c_i. And the algorithm $\mathcal{F}'$ is recursively applied to its arguments.
3. When t_i is a hylo application $[\![\phi', \eta, out_F]\!]_{F',F}$, our main algorithm $\mathcal{F}$ (not $\mathcal{F}'$) is applied recursively to ϕ'.
4. When t_i is a function application $f\ t_1\ \cdots\ t_n$, we apply $(\![c_1 \triangledown \cdots \triangledown c_n]\!)$.

(b) Correctness of Algorithm $\mathcal{F}$

When abstracting constructors in ϕ, one might wonder why not just simply substitute in ϕ all occurrences of the constructors of μF with corresponding lambda variables. In fact, special care should be taken here. First, there are two kinds of constructors of μF in ϕ, one is to be substituted and the other is not. To see the difference between them, let us consider an example

$$\phi = \lambda().\ Nil \triangledown \lambda(v, v').\ Cons\ (Cons\ (v, Nil), v').$$

We can derive a correct τ for ϕ as

$$\tau = \lambda(c_1 \triangledown c_2).\ (\lambda().\ c_1 \triangledown \lambda(v, v').\ c_2\ (\underline{Cons\ (v, Nil)}, v')).$$

This tells us that we must not substitute the constructors inside an element of the list (as underlined). In the rule for constructor application in Figure 6, this fact is taken into consideration in applying $\mathcal{F}'$ recursively only to suitable arguments directed by functor F_i. The second point to note is that we have to consider the construction of non-recursive arguments of ϕ which are actually used for producing the final result. In this case we have to substitute them in this phase by applying a catamorphism.

Theorem 3 (Correctness of Algorithm $\mathcal{F}$) For the algorithm in Figure 6, we have that

(i) $\phi = \mathcal{F}[\![\phi]\!]\ in_F$
(ii) $\mathcal{F}[\![\phi]\!] : \forall A.(FA \to A) \to (F'A \to A)$

Proof Sketch. The proof of (i) is straightforward. We can obtain ϕ from $\mathcal{F}[\![\phi]\!]\ in_F$ by recovering $c_1, \cdots, c_n$ with $C_1, \cdots, C_n$ respectively and using the fact that $(\![C_1 \triangledown \cdots \triangledown C_n]\!)_F$ is the identity function over type μF.

As to (ii), it will be true according to Theorem 2 if we can prove that, for any $\phi : F'(\mu F) \to \mu F$, we have

$$\forall \alpha.\ \ (\![\alpha]\!) \circ \phi = (\mathcal{F}[\![\phi]\!]\ \alpha) \circ F'\ (\![\alpha]\!).$$

This can be done by induction over the structure of ϕ. □

To make the point clear, we demonstrate an example concerning the abstraction of production of a tree instead of a list. Consider the function foo which takes a list and generates a balanced tree:

$$\begin{array}{lcl} foo & = & \lambda xs.\ \text{case}\ xs\ \text{of} \\ & & \quad Nil \to Leaf \\ & & \quad Cons\ (a, as) \to Node\ (double\ a, squareNodes\ (foo\ as), foo\ as) \end{array}$$

where $squareNodes$ is a hylomorphism:

$$[\![\lambda().Leaf \triangledown \\ \lambda(v, v'_1, v'_2).\, Node(v, v'_1, v'_2), id + \lambda(v, v'_1, v'_2).(square\ v, v'_1, v'_2), out_{F_T}]\!]_{F_T, F_T}$$

and F_T is the functor defining the tree type in Section 2.2. By applying the algorithm for hylomorphism derivation, we have

$$foo = [\![\phi, id, out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\ \quad \text{where } \phi = \lambda().\ Leaf \triangledown \lambda(v, v').\ Node\ (double\ v, squareNodes\ v', v')$$

which can be restructured into

$$foo = [\![\phi, id + \lambda(v, v').(double\ v, v'), out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\ \quad \text{where } \phi = \lambda().\ Leaf \triangledown \lambda(v, v').\ Node\ (v, squareNodes\ v', v').$$

Now applying Algorithm $\mathcal{F}$ to derive τ for $\phi : F_L(\mu F_T) \to \mu F_T$ in the above hylo gives our result.

$$\begin{aligned} foo \quad = \quad & [\![\tau\ in_{F_T}, id + \lambda(v, v').(double\ v, v'), out_{F_{L_A}}]\!]_{F_{L_A}, F_{L_A}} \\ & \text{where } \tau = \lambda(c_1 \triangledown c_2).\ (\lambda().\ c_1 \triangledown \\ & \qquad \lambda(v, v').\ c_2\ (v, [\![\lambda().c_1 \triangledown \lambda(v, v'_1, v'_2).c_2\ (v, v'_1, v'_2), \\ & \qquad id + \lambda(v, v'_1, v'_2).(square\ v, v'_1, v'_2), out_{F_T}]\!]_{F_T, F_T}\ v', v')) \end{aligned}$$

(c) Algorithm for Deriving σ

Another case of applying the Acid Rain Theorem is to fuse the composition of two hylomorphisms like $[\![_, _, \psi]\!]_{_, F'} \circ [\![in_F, _, _]\!]_{F, _}$, where σ must be deribed from ψ such that $\sigma out_F = \psi$. Deriving σ is the *dual* process of deriving τ, but from technical point of view, there are some important points worth noting.

Our algorithm is shown in Figure 7, deriving σ from $\psi : \mu F \to F'\mu F$ in the following restrictive form e:

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

in which the expression after the keyword case has been simplified to a variable. The key point of our algorithm is to transform the "implicit" decomposition of input by pattern matching into an "explicit" one by using out_F which decomposes μF into $F\mu F$. To this end, we define $d_1, \ldots, d_n$, which decomposes (unfolds) data of type μF successively by out_F as follows.

$$\begin{aligned} d_1 \quad &= \quad out_F\ v_s \\ d_2 \quad &= \quad F\ out_F\ d_1 \\ & \vdots \\ d_m \quad &= \quad F^{m-1}\ out_F\ d_{m-1} \end{aligned}$$

$\psi \ : \ \mu F \to F'\mu F$
$\psi = \sigma out_F, \quad$ where $\sigma = \lambda\beta.\mathcal{G}[\![\psi]\!]$

where

$$
\begin{array}{l}
\mathcal{G}[\![\lambda v_s.\text{case } v_s \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!] = \\
\quad \lambda v_s.\text{let } d_1 = \beta\ t_0 \\
\qquad\qquad\ d_2 = F\ \beta\ d_1 \\
\qquad\qquad\qquad\ \vdots \\
\qquad\qquad\ d_m = F^{m-1}\ \beta\ d_{m-1} \\
\qquad \text{in} \\
\qquad \text{case } (d_1, \ldots, d_m) \text{ of} \\
\qquad\quad p'_1 \to t_1 \\
\qquad\qquad\ \vdots \\
\qquad\quad p'_n \to t_n \\
\qquad \text{where } (v_i, q_i) = \mathcal{G}'[\![p_i]\!] \quad (i = 1, \ldots, n) \\
\qquad\qquad\ [q_{i_1}, \ldots, q_{i_{l_i}}] = q_i \\
\qquad\qquad\ m = max\{|q_1|, \ldots, |q_n|\} \\
\qquad\qquad\ p'_i = (q_{i_1}, \ldots, q_{i_{l_i}}, _, \ldots, _) \quad \text{(m-tuple)}
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{G}'[\![v]\!] = & (v,\ []) \\
\mathcal{G}'[\![C_j\ p_1 \cdots p_n]\!] = & (_,\ q) \\
& \text{where } (v_i, q_i) = \mathcal{G}'[\![p_i]\!] \quad (i = 1, \ldots, n) \\
& \qquad q = (\mathsf{j},\ (v_1, \ldots, v_n)) : map\ (\lambda x.(\mathsf{j}, x))\ (zip_n\ q_1 \cdots q_n) \\
\mathcal{G}'[\![(p_1, \ldots, p_n)]\!] = & (_,\ q) \\
& \text{where } (v_i, q_i) = \mathcal{G}'[\![p_i]\!] \quad (i = 1, \ldots, n) \\
& \qquad q = zip_n\ q_1 \cdots q_n \\
& /^*\ \text{This } zip_n \text{ is different from the standard in regard to } ^*/ \\
& /^*\ \text{supplying wildcard } _ \text{ at the tail for shorter lists } ^*/
\end{array}
$$

Figure 7 Deriving σ

Here F^m is defined inductively by $F^m = F^{m-1} \circ F$. Now, we can substitute $(d_1, \ldots, d_m) : F\mu F \times F^2\mu F \times \cdots \times F^m\mu F$ for $v_s : \mu F$ in ψ, and accordingly, as shown in Figure 7, we apply the algorithm $\mathcal{G}'$ to turn the case patterns of $p_1, \ldots, p_n$ corresponding to the original v_s into $p'_1, \ldots, p'_n$ corresponding to $(d_1, \ldots, d_m)$. Finally, we lambda abstract all out_F's in all d_i's and obtain the result σ.

We omit the explanation and proof of the correctness of the algorithms for deriving σ from ψ. Instead, we give an example. A function calculating the maximum of a nonempty list of integers is defined and transformed into a hylomorphism as:

$$
\begin{array}{lll}
maximum & = & \lambda xs.\ \text{case } xs \text{ of } Nil \to error \\
& & \qquad\qquad\qquad\ Cons\ (a, Nil) \to a
\end{array}
$$

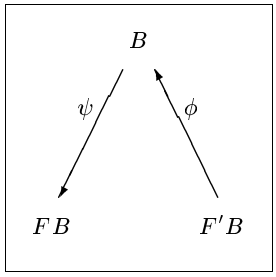
$$
\begin{array}{rcl}
 & & \quad Cons\ (a, as) \to max(a, maximum\ as) \\
 & = & [\![\phi, id, \psi]\!]_{F',F'} \\
 & & \text{where } \phi = error \triangledown id \triangledown max \\
 & & \qquad \psi = \lambda xs.\ \text{case } xs \text{ of } Nil \to (1, ()) \\
 & & \qquad\qquad\qquad\qquad Cons\ (a, Nil) \to (2, (a)) \\
 & & \qquad\qquad\qquad\qquad Cons\ (a, as) \to (3, (a, as)) \\
 & & \qquad F' = !\mathbf{1} + !Int + !Int \times I.
\end{array}
$$

We apply Algorithm $\mathcal{G}'$ to the pattern part of the case expression in ψ:

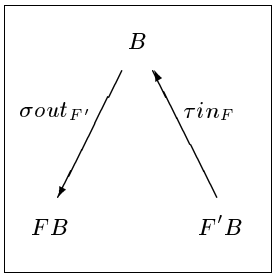
$$
\begin{array}{rcl}
\mathcal{G}'[\![Nil]\!] & = & (_,\ [(1, ())]) \\
\mathcal{G}'[\![Cons\,(a,\ Nil)]\!] & = & (_,\ q) \\
 & & \text{where } (a,\ []) = \mathcal{G}'[\![a]\!] \quad (_,\ [(1, ())]) = \mathcal{G}'[\![Nil]\!] \\
 & & \qquad q = (2,\ (a, _)) : map\ (\lambda x.\ (2, x))\ (zip\ []\ [(1, ())]) \\
 & & \qquad\ = [(2,\ (a, _)), (2, (_, (1, ())))] \\
\mathcal{G}'[\![Cons\,(a,\ as)]\!] & = & (_,\ q) \\
 & & \text{where } (a,\ []) = \mathcal{G}'[\![a]\!] \quad (as,\ []) = \mathcal{G}'[\![as]\!] \\
 & & \qquad q = (2,\ (a, as)) : [] \\
 & & \qquad\ = [(2,\ (a, as))]
\end{array}
$$

The longest list produced by $\mathcal{G}'$ is for the second alternative and its length is two. So we apply out_F twice to generate the case expression for switching alternatives. The σ function with a let expression which binds decomposed xs to d_1 and d_2 is given by

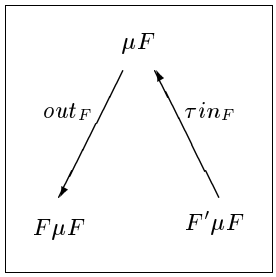
$$
\begin{array}{rcl}
\sigma & = & \lambda\beta.\ \mathcal{G}[\![\lambda xs.\ \text{case } xs \text{ of } Nil \to (1, ()) \\
 & & \qquad\qquad\qquad\qquad Cons\ (a, Nil) \to (2, (a)) \\
 & & \qquad\qquad\qquad\qquad Cons\ (a, as) \to (3, (a, as))]\!] \\
 & = & \lambda\beta.\ \lambda xs.\ \text{let } d_1 = \beta\ xs \\
 & & \qquad\qquad\ d_2 = F'\ \beta\ d_1 \text{ in} \\
 & & \qquad \text{case } (d_1, d_2) \text{ of } ((1, ()), _) \to (1, ()) \\
 & & \qquad\qquad\qquad\qquad ((2, (a, _)), (2, (_, (1, ())))) \to (2, (a)) \\
 & & \qquad\qquad\qquad\qquad ((2, (a, as)), _) \to (3, (a, as)).
\end{array}
$$



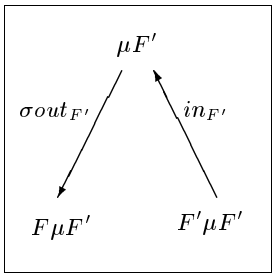
(a) ϕ and ψ in general case



(b) Fusion Impossible



(c) $B = \mu F$



(d) $B = \mu F'$

Figure 8 Relation between two hylomorphisms

4.4 Applying Acid Rain Theorem

In this section, we describe how to apply the Acid Rain Theorem for fusion. Theorem 1 indicates that it must have special forms $[\![_, _, \tau in_F]\!] \circ [\![out_F, _, _]\!]$ or $[\![_, _, in_F]\!] \circ [\![\sigma out_F, _, _]\!]$ to apply this theorem. Now we are confronted with some technical problems. For example, how to find a potential fusible composition in a program? And how to check automatically if the composition can really be fused by the Acid Rain Theorem?

We define our potential fusible compositions as follows, where there really exist intermediate data structures (rather than the data supported directly by machines such as data of types *float* and *Int*).

Definition 3 (Potential Fusible Composition) A term is called a *potential fusible composition* if (i) it is either in the form of $t_1 \circ t_2$ or in the form of $(t_1\ (t_2\ t))$; and (ii) the domain of t_1 is a data structure defined by an endofunctor. □

We start with the main function to which we apply fusion transformation. We then select a potential fusible composition in the definition body, say $t_1 \circ t_2$, and try to fuse it with the Acid Rain Theorem by obtaining suitable

hylomorphisms for both t_1 and t_2. Now consider a fusion of two hylomorphism $h_1 \circ h_2$:

$$\begin{array}{ll} h_1 : B \to C & h_2 : A \to B \\ h_1 = [\![_, _, \psi]\!]_{_,F} & h_2 = [\![\phi, _, _]\!]_{F',_} \end{array}$$

Figure 8 (a) shows the relation between ψ and ϕ, and the data types delivered by them. To apply the theorem, the following procedure is taken. The terms in parentheses indicate the dual condition to be checked at the same time.

1. *Check whether B is μF ($\mu F'$).* Obviously, if B is neither μF nor $\mu F'$ as shown in Figure 8 (b), we cannot obtain the form to which the Acid Rain Theorem is applicable.
2. *Check whether ψ (ϕ) is out_F ($in_{F'}$), in case B is μF ($\mu F'$).* Generally, it is impossible to tell whether two functions are equal or not. But we can judge whether ϕ is in_F by applying ϕ and in_F to all (finite) cases of data type μF and comparing the result values.
3. *Check whether ϕ (ψ) in one hylomorphism can be transformed to τin_F ($\sigma out_{F'}$), if ψ (ϕ) in another is out_F ($in_{F'}$).* By using the algorithms in Figures 6 and 7, we arrive at the state as shown in Figures 8 (c) and (d).

It is the compositions passing these tests that the Acid Rain Theorem can be applied to.

Returning to our running example ssf, we can fuse the derived hylomorphisms successfully. Changing function applications to function composition and noting that

$$[\![\phi_1, \eta_1, out_F]\!]_{F,F} \circ [\![in_F, \eta_2, \psi_2]\!]_{F,F} \Rightarrow [\![\phi_1, \eta_1 \circ \eta_2, \psi_2]\!]_{F,F}$$

lead from the original function ssf to the following.

$$\begin{array}{lcl} ssf & = & sum \circ map\ square \circ upto \\ & = & [\![0 \triangledown (+), id, out_F]\!]_{F,F} \circ [\![in_F, id + \lambda(v, v').\ (square\ v, v'), out_F]\!]_{F,F} \circ \\ & & \quad [\![in_F, id, \psi]\!]_{F,F} \\ & = & [\![0 \triangledown (+), id + \lambda(v, v').\ (square\ v, v'), out_F]\!]_{F,F} \circ [\![in_F, id, \psi]\!]_{F,F} \\ & = & [\![0 \triangledown (+), id + \lambda(v, v').\ (square\ v, v'), \psi]\!]_{F,F} \\ & = & \lambda(m, n).\ \text{case}\ (m > n)\ \text{of}\ True \to 0 \\ & & \qquad\qquad False \to square\ m + ssf\ (m + 1, n) \end{array}$$

4.5 Fusing inside Hylomorphisms

So far we have shown how to fuse between two hylomorphisms. We have not, however, mentioned fusion inside hylomorphisms. At a first glance, it seems that this fusion is very simple; performing fusion transformation for each component in a hylomorphism. But this simple strategy cannot effectively remove intermediate data structures among three components. As a simple example, consider the following hylomorphism (derived from $concat \circ map\ f$):

$$[\![\lambda().()\ \triangledown\ \lambda(u,v).u +\!\!+ v,\ \ id \triangledown \lambda(a,b).(f\ a,b),\ \ out]\!]$$

in which each component cannot be fused any more. But if we shift the η part to the ψ part as:

$$[\![\lambda().()\ \triangledown\ \lambda(u,v).f\,u +\!\!+ v,\ \ id,\ \ out]\!]$$

then the data structures produced by f become eliminatable as they are consumed by $(+\!\!+\ v)$.

Our idea is to perform fusion for a hylomorphism, say $[\![\phi,\eta,\psi]\!]$, in the following way:

(1) Fuse each component of the hylomorphism $[\![\phi \circ \eta, id, \psi]\!]$;
(2) Restructure the hylomorphism obtained in Step 1 and get $[\![\phi',\eta',\psi']\!]$;
(3) Fuse each component of the hylomorphism $[\![\phi', id, \eta' \circ \psi']\!]$;
(4) Restructure the hylomorphsims obtained in Step 3.

4.6 Inlining Hylomorphisms

Inlining hylomorphisms is the inverse process of hylomorphism derivation. All the hylomorphisms left in a program are transformed back to recursive definitions.

5 PERFORMANCE EVALUATION

All algorithms reported in this paper have been implemented. To evaluate the effectiveness of the HYLO system and get an idea of how much improvement could be gained by our approach, we have investigated the performance of three programs, namely *ssf*, *unlines*, and *queens*, both before and after our fusion transformation. *ssf* is our running example. *unlines*, a program from the Gofer prelude, accepts a list of strings and returns a new string by concatenating all strings separated by the CR character. *queens* is the well-known queens program, placing n queens on an $n \times n$ chess board.

Table 1 Experimental results using Gofer († applies to a text with 100 words)

Program	Reduction Steps			Heap Cells		
	before fusion	after fusion	ratio	before fusion	after fusion	ratio
$ssf(1000)$	11,015	6,005	0.54	17,030	10,019	0.59
$unlines^{\dagger}$	4,151	1,759	0.42	8,079	4,393	0.54
$queens(10)$	33,776,593	26,419,252	0.89	65,428,706	50,839,988	0.78

Table 2 Experimental results of *queens* using GHC

	Total Time (secs)		Heap Cells (mbytes)	
	before fusion	after fusion	before fusion	after fusion
by cheap fusion	9.41	5.13	61.27	15.55
by hylo fusion	9.41	4.10	61.27	9.38

We proceed our experiments using Gofer interpreter (Version 2.30a) and Glasgow Haskell compiler (GHC for short, Version 0.29). First, these programs were evaluated using Gofer. The experimental results are given in Table 1, showing improvement both in time (reduction steps) and in space (heap-cells). For example, it indicates that our HYLO system can save about half of both reduction steps and heap cells for *ssf* and *unlines*. To be more concrete, we give both the initial and transformed programs for the queens problem in Figure 9. Its non-triviality shows the power of our HYLO system.

Second, we compiled both the original and transformed queens programs, in order to compare our fusion with the cheap fusion (Gill et al. 1993) which has been already implemented in GHC. We enable and disable the cheap deforestion in GHC by two compile options `-O` and `-fno-foldr-build` respectively. Table 2 gives the experimental results indicating that our fusion gets better results than the cheap fusion; about 20% faster with 60% heap cells. It is also interesting to see that the improvement using GHC is exciting (comparing with that using Gofer); reducing time from 9.41 sec to 4.10 sec and heap cells from 61.27 mbytes to 9.38 mbytes.

6 DISCUSSION AND CONCLUDING REMARKS

Programming with the use of generic control structures which capture patterns of recursions in a uniform way is very significant in program transformation and optimization (Gill et al. 1993, Meijer et al. 1991, Sheard & Fegaras 1993, Launchbury & Sheard 1995, Takano & Meijer 1995). Our work is closely related to these studies. In particular, our work was greatly motivated by the work in (Sheard & Fegaras 1993) and (Takano & Meijer 1995). Sheard and

```
nqueen = 10

queens_initial = (print . sum . concat . queens) nqueen
  where
    queens 0 = [[]]
    queens m = [ p ++ [n] | p <- queens (m-1),
                            n <- [1..nqueen], safe p n ]
    safe p n = all not [ check m n (i,j) | (i,j) <- zip [1..] p]
               where m = length p + 1
    check m n (i,j) = j==n || i+j==m+n || i-j==m-n

queens_transformed = print (
  let
    x681 [] = 0
    x681 ((:) x342 x343)
     = let
         x680 [] = x681 x343
         x680 ((:) x351 x352) = (+) x351 (x680 x352)
       in x680 x342
  in x681
      (let
        x687 x354
         = case ((==) x354 0) of
            True -> (:) [] []
            False ->
             let
              x686 [] = []
              x686 ((:) x375 x376)
               = let
                   x685 x528
                    = case ((<=) x528 nqueen) of
                       True ->
                        case (let
                                x404 = (+) 1 (length x375)
                              in let
                                   x682 ((:) x446 x447,(:) x448 x449)
                                    = (&&) (not ((||) ((==) x448 x528)
                                                      ((||) ((==) ((+) x446 x448)
                                                                  ((+) x404 x528))
                                                            ((==) ((-) x446 x448)
                                                                  ((-) x404 x528)))))
                                          (x682 (x447,x449))
                                   x682 x450 = True
                                  in x682 (let
                                             x683 x458
                                               = (:) x458 (x683 ((+) 1 x458))
                                           in x683 1,x375)) of
                         True -> (:) (let
                                        x684 [] = (:) x528 []
                                        x684 ((:) x508 x509)
                                                 = (:) x508 (x684 x509)
                                      in x684 x375) (x685 ((+) 1 x528))
                         False -> x685 ((+) 1 x528)
                       False -> x686 x376
                in x685 1
             in x686 (x687 ((-) x354 1))
       in x687 nqueen))
```

Figure 9 Initial and Transformed *queens* Programs

Fegaras implemented a fusion algorithm called the *normalization algorithm* which can work on the language containing *folds*, a special case of hylomorphisms, as basic recursive form. Because of the restriction of descriptive power of folds, the power of the fusion system is rather limited. On the other hand, Takano and Meijer adopted hylomorphisms as the basic recursive forms rather than folds. Their work was motivated by Gill, Launchbury and Peyton Jones's one-step fusion algorithm (Gill et al. 1993) relying on functions being written in a highly-stylized *build-cata* forms (i.e., folds with data constructors being parameterized), and implemented another one-step fusion algorithm based on the Acid Rain Theorem. However, it is impractical to force programmers to define their recursive definitions only in terms of the specific hylomorphisms so that the Acid Rain Theorem could be applied directly (Launchbury & Sheard 1995).

This work extends our previous work (Hu et al. 1996*b*) in which the automatic algorithms for the derivation of structural hylomorphisms are proposed. The main extensions are as follows. First, we add new language constructs to the original core language for practical reasons, and modify the original algorithms for deriving and restructuring hylomorphisms. Second, we develop our new algorithms for the derivation of polymorphic functions τ and σ implementing the transformation laws given in (Hu et al. 1996*b*). Third, also the most important, we are implementing HYLO, stepping towards a practically-useful fusion system.

Functional programs that HYLO manipulates can be accepted by the Haskell (core) language. The merit is that the verification of the improvement of functional programs after transformation can be done easily by running them on Haskell system. It is hoped that the HYLO system will be embedded in the practical Haskell system.

All transformation algorithms introduced in this paper have been implemented. It is completely mechanical and does not rely on heuristics about how or when transformation is taken. Although we have to wait for the detailed experimental results to say that this system is effective for practical programs, we are absolutely convinced that our calculational approach to fusion transformation makes a good progress in code optimization of functional programs.

Besides evaluating general performance of HYLO, we are currently working on the extension of our algorithms to deal with recursions traversing over multiple data structures (Hu et al. 1996*c*), mutual recursions etc. in order to enable more fusion. In the near future, we are going to extend HYLO to be a general automatic program calculation system, which can optimize functional programs not only by fusion but also by tupling (Hu, Iwasaki, Takeichi & Takano 1997), and accumulating (Hu, Iwasaki & Takeichi 1996*a*), as well as other optimization tactics.

ACKNOWLEDGEMENTS

This paper owes much to the thoughtful and helpful discussions with Akihiko Takano, Fer-Jan de Vries and other CACA members. Thanks are also due to Oege de Moor and Arne John Glenstrup for reading an earlier draft and making a number of helpful suggestions, and to the referees who provided detailed and helpful comments.

REFERENCES

Bird, R. & de Moor, O. (1994), Relational program derivation and context-free language recognition, *in* A. Roscoe, ed., 'A Classical Mind', Prentice Hall, pp. 17–35.

Burstall, R. & Darlington, J. (1977), 'A transformation system for developing recursive programs', *Journal of the ACM* **24**(1), 44–67.

Chin, W. (1992), Safe fusion of functional expressions, *in* 'Proc. Conference on Lisp and Functional Programming', San Francisco, California.

Fokkinga, M. (1992), Law and Order in Algorithmics, Ph.D thesis, Dept. INF, University of Twente, The Netherlands.

Gill, A., Launchbury, J. & Jones, S. P. (1993), A short cut to deforestation, *in* 'Proc. Conference on Functional Programming Languages and Computer Architecture', Copenhagen, pp. 223–232.

Hu, Z., Iwasaki, H. & Takeichi, M. (1996*a*), Caculating accumulations, Technical Report METR 96–03, Faculty of Engineering, University of Tokyo.

Hu, Z., Iwasaki, H. & Takeichi, M. (1996*b*), Deriving structural hylomorphisms from recursive definitions, *in* 'ACM SIGPLAN International Conference on Functional Programming', ACM Press, Philadelphia, PA, pp. 73–82.

Hu, Z., Iwasaki, H. & Takeichi, M. (1996*c*), An extension of the Acid Rain Theorem, *in* '2nd Fuji International Workshop on Functional and Logic Programming', World Scientific, Shonan, Japan.

Hu, Z., Iwasaki, H., Takeichi, M. & Takano, A. (1997), Tupling calculation eliminates multiple data traversals, *in* 'ACM SIGPLAN International Conference on Functional Programming', ACM Press, Amsterdam, The Netherlands. to appear.

Jones, M. P. (1994), Gofer 2.30a release notes.

Launchbury, J. & Sheard, T. (1995), Warm fusion: Deriving build-catas from recursive definitions, *in* 'Proc. Conference on Functional Programming Languages and Computer Architecture', La Jolla, California, pp. 314–323.

Meijer, E., Fokkinga, M. & Paterson, R. (1991), Functional programming with bananas, lenses, envelopes and barbed wire, *in* 'Proc. Conference on Functional Programming Languages and Computer Architecture (LNCS 523)', Cambridge, Massachuetts, pp. 124–144.

Pettorossi, A. & Proietti, M. (1993), Rules and strategies for program transformation, *in* 'IFIP TC2/WG2.1 State-of-the-Art Report', LNCS 755, pp. 263–303.

Sheard, T. & Fegaras, L. (1993), A fold for all seasons, *in* 'Proc. Conference on Functional Programming Languages and Computer Architecture', Copenhagen, pp. 233–242.

Takano, A. & Meijer, E. (1995), Shortcut deforestation in calculational form, *in* 'Proc. Conference on Functional Programming Languages and Computer Architecture', La Jolla, California, pp. 306–313.

Team, T. A. (1996), *Glasgow Haskell Compiler User's Guide ver0.29*.

Wadler, P. (1988), Deforestation: Transforming programs to eliminate trees, *in* 'Proc. ESOP (LNCS 300)', pp. 344–358.

7 BIOGRAPHY

Yoshiyuki Onoue is a Ph.D. student at the University of Tokyo. His research area is the transformation of functional programs, in particular the algorithm and implementation for partial evaluation and fusion transformation.

Zhenjiang Hu is a research associate at the University of Tokyo. He received his BS and MS in computer science from Shanghai Jiao Tong University in 1988 and 1990 respectively, and his Ph.D in information engineering from the University of Tokyo in 1996. His current research concerns functional programming, calculational program transformation systems and program evolution.

Hideya Iwasaki is an associate professor of Faculty of Technology, Tokyo University of Agriculture and Technology. He received the M.E. degree in 1985, the Dr. Eng. Degree in 1988 from the University of Tokyo. His research interests are list processing languages, functional languages, parallel processing, and constructive algorithmics.

Masato Takeichi is a professor in Mathematical Engineering and Information Engineering at the University of Tokyo since 1993. After graduation from the University of Tokyo, he joined the faculty at the University of Electro-Communications in Tokyo before he went back to work at the Univesity of Tokyo in 1987. His research concerns the design and implementation of functional programming languages, and calculational program transfomation systems.