A Calculational Fusion System HYLO

Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi

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 with no 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 transformation.

Keywords

Program Transformation, Fusion, Constructive Algorithmics, 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 [PP93]. 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. It has been gaining more and more interest even in functional programming community [Wad88, Chi92, GLJ93, TM95, HIT96c].

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 [Wad88]: 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, easier to write, and potentially more 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) and degrade 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 [Wad88, Chi92] makes the composition of recursive functions into a simple function by *fold-unfold* transformations [BD77]. 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 function calls occurred so far 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 prevent fusion from being practically implemented [GLJ93].

Our interest is in the calculational fusion [SF93, GLJ93, TM95, LS95, HIT96c, HIT96d], 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 exploration of recursive structure for each component function so that fusion can be performed directly by applying a simple transformation law: the *Acid Rain Theorem* [GLJ93, TM95]. Its theoretical basis can be found in the study of *Constructive Algorithmics*[Fok92, Mal90, MFP91]. Returning to our example program *ssf*, let us explain briefly how to fuse the following composition with this approach, while the detailed discussions will be given later:

$$sum \circ map\ square \circ upto.$$

1. *Deriving Hylomorphisms from Recursive Definitions*
   First of all, we need to rewrite every component recursive function in terms of a general 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 [BdM94], and we have proposed an algorithm for deriving hylomorphisms from recursive definitions of functions [HIT96c]. 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 Scheme*
   For a composition of two hylomorphisms, there is a useful law called *Acid Rain Theorem* (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 over data constructors are 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 each 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:

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

   Then applying Step 2 to the second hylomorphism gives

$$[\![\phi_1, \eta_1, \sigma_1\, out]\!] \circ [\![\tau_2'\, in, \eta_2'', \psi_2'']\!].$$

   Again, applying the Acid Rain Theorem yields a single hylomorphism $[\![\phi_1, \eta_1, \psi']\!]$ without any composition.
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.

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

   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 [GLJ93, TM95], but so far as we are aware of, there are few practical fusion systems based on this approach. The difficulties in the design and implementation of a real fusion system lie in two respects:

- *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 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 (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 particularly, we should focus on specification *language design*, which should not only be convenient to specify problems but also be general and powerful enough, and *algorithm design*, which should be as much applicable as possible to 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 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 approach. 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 a compiler 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 look over the HYLO system in Section 3. Implementation algorithms are presented in Section 4 with transformation examples, and experiment results are given in Sectionsec:experiment. Section 6 gives remarks on the HYLO system and related works.

2 PRELIMINARY

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. For example, a function f over *List* is often defined in a pattern matching way as

$$\begin{array}{lcl} f\ Nil & = & \cdots \\ f\ Cons(x, xs) & = & \cdots \end{array}$$

whose definition body can be expressed in any way. Recently, many studies [MFP91, SF93] showed that recursion should be structured with its definition body being expressed in a specific form. It is motivated for many of the same reasons that the "structured programming" methodology for imperative programs was motivated, but it has an even larger payback: the structure can be exploited to facilitate the proof of program properties and to calculate program transformation.

Hylomorphism is one such specific recursive form which can describe almost all recursive functions of interest [BdM94, HIT96c]. 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 F-structure from the input by ψ; applying f to all recursive components in F-structure by $F\, f$; manipulating F-structure into 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

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

Hylomorphism enjoys many useful calculational laws (Section 2.3), facilitating program transformation. To be more precise, we shall review previous works on program calculation [MFP91, Fok92, TM95]. Rather than being involved in a complicated discussion in terms of category theory, we shall impose emphasis on the results which will be used later and the notational conventions to which we hope that the readers could get used.

2.1 Functors

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

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

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

□

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

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

where id stands for the identity function. □

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

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

Some related operators are:

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

□

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

as follows.

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

Some related operators are:

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

□

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

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

2.2 Data Types as Initial Fixed Points of Functors

A data type is a collection of data constructors 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 [Fok92]. 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 $\mathbf{1}$ denotes the final object, corresponding to $()$*. In fact, the definition of F_{L_A} can be automatically derived from the original definition of $List\ A$ [SF93]. 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 binary trees, usually declared by

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

is captured by the following endofunctor:

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

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

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

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

Hylomorphism, a general recursive form covering the well-known *catamorphism* and *anamorphism* as its special cases, is defined in triplet form[TM95] as follows.

Definition 5 (Hylomorphism in triplet form)
Let F and G be two functors. Given $\phi : G\,A \to A$, $\psi : B \to F\,B$ and natural transformation $\eta : F \dot{\to} G$, the hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F} : B \to A$ is defined as the least fixed point of the following equation.

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

Hylomorphisms (Hylo for short) are powerful in description in that practically every recursion of interest (e.g., primitive recursions) can be specified by them [HIT96c]. 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 it is clear from the context.

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

Definition 6 (Catamorphism $([-])$, Anamorphism $[\!(-)\!]$)

$$\begin{array}{rcl} ([\phi])_F & = & [\![\phi, id, out_F]\!]_{F,F} \\ [\!(\psi)\!]_F & = & [\![in_F, id, \psi]\!]_{F,F} \end{array} \qquad \square$$

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 that a natural transformation can be shifted around inside a hylomorphism. It is very useful for transformation inside a hylomorphism. Another useful law is the *Acid Rain* Theorem [TM95], showing how to fuse a composition of two hylomorphisms into a single one.

Theorem 1 (Acid Rain)

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

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

$\square$

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 [Jon94], and can be easily ported to other systems such as Haskell [HAB+95]. The HYLO system receives a program written in Gofer and

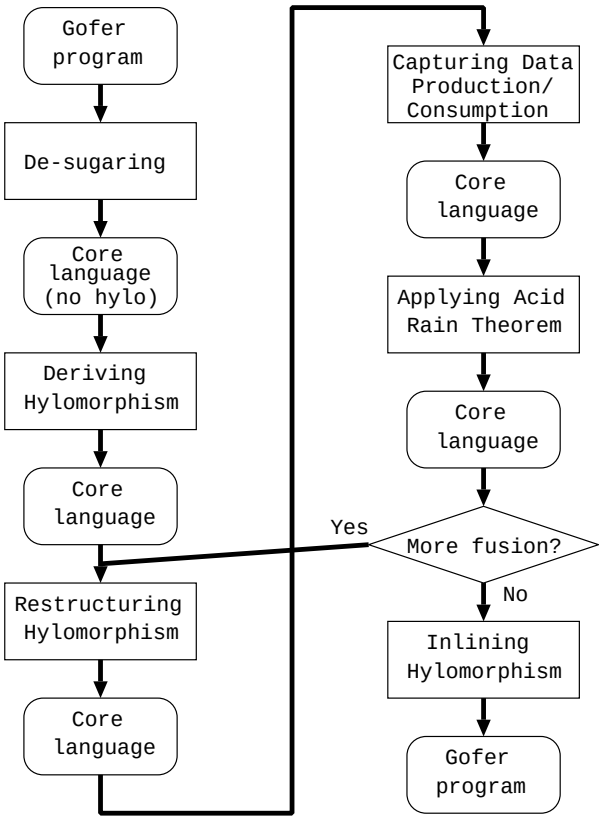


Figure 1 Overview of HYLO System

returns an improved one as its result with intermediate data structures being 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 we will use hereafter in the transformation steps.

The core language is shown in Figure 2. It is similar to that used by the GHC [Tea96] system. Main difference between these two languages 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 be easily flattened.

The next step of our transformation process is to receive a program written in the core language and derive hylomorphisms from all recursive function definitions. As we explained in Section 2.3, 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 admitted to shift some parts of the components of $[\![\phi, \eta, \psi]\!]$ each other. Our algorithm extracts computations from ϕ and ψ and moves them

Prog	::=	*Def*; *Prog* | *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 $p_1 \rightarrow t_1; \cdots; p_n \rightarrow t_n$	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)
	|	$[\![t_\phi, t_\eta, t_\psi]\!]_{F_0,F_1}$	Hylo representation (F_0, F_1 are functors)
	|	(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

to the η component. This process simplifies 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 specific forms τin_F and σout_F for the Acid Rain Theorem to be applicable. The algorithms are presented in Sections 4.3.

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

Finally, hylomorphism representation will be 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

We propose several important algorithms for the implementation of HYLO system in this section. 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 extension of those already discussed in [HIT96c].

4.1 Deriving Hylomorphism

We extend the algorithm in [HIT96c] 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 in [HIT96c]. We shall omit any detailed description here, but give an informal explanation for understanding the derivation process. Below shows how we can turn the typical recursive definition of function f in pattern matching style:

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

into a hylomorphism.

$$\begin{array}{lll}
f & = & \{\text{Definition of } f\} \\
 & & \lambda v_s.\text{case } t_0 \text{ of } p_1 \rightarrow t_1; \cdots; p_n \rightarrow t_n \\
 & = & \{\text{Trick 1: Replacing } t_i \text{ with } g_i\ t'_i\} \\
 & & \lambda v_s.\text{case } t_0 \text{ of } p_1 \rightarrow g_1\ t'_1; \cdots; p_n \rightarrow g_n\ t'_n \\
 & = & \{\text{Using separated sum}\} \\
 & & (g_1 \triangledown \cdots \triangledown g_n) \circ (\lambda v_s.\text{case } t_0 \text{ of } p_1 \rightarrow (\mathsf{1},\ t'_1); \cdots; p_n \rightarrow (\mathsf{n},\ t'_n)) \\
 & = & \{\text{Trick 2: Replacing } g_i \text{ with } \phi_i \circ F_i\ f\} \\
 & & (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ ((F_1 + \cdots + F_n)\ f) \circ (\lambda v_s.\text{case } t_0 \text{ of } p_1 \rightarrow (\mathsf{1},\ t'_1); \cdots; p_n \rightarrow (\mathsf{n},\ t'_n)) \\
 & = & \{\text{Auxiliary definitions } \phi, F, \text{and } \psi\} \\
 & & \phi \circ F\ f \circ \psi \\
 & = & \{\text{Definition of hylomorphism}\} \\
 & & [\![\phi, id, \psi]\!]_{F,F} \\
 & & \text{where } F = F_1 + \cdots + F_n \\
 & & \qquad \phi = \phi_1 \triangledown \cdots \triangledown \phi_n \\
 & & \qquad \psi = \lambda v_s.\text{case } t_0 \text{ of } p_1 \rightarrow (\mathsf{1},\ t'_1); \cdots; p_n \rightarrow (\mathsf{n},\ t'_n)
\end{array}$$

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, 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 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 renaming process has been done, so all bounded variables have been assigned unique names.

We demonstrate how this algorithm work on the example program *ssf* in Section 1.

$$
\begin{array}{l}
\mathcal{A}[\![\lambda v_{s_1} \cdots \lambda v_{s_m}.\ \text{case } t_0 \text{ of } p_1 \to t_1; \cdots; p_n \to t_n]\!]\ f = \lambda v_{s_1} \cdots \lambda v_{s_{m-1}}.\ [\![\phi_1 \triangledown \cdots \triangledown \phi_n, id, \psi]\!]_{F,F} \\
\quad \text{where} \\
\qquad (\{v_{i_1}, \ldots, v_{i_{k_i}}\}, \{(v'_{i_1}, t_{i_1}), \cdots, (v'_{i_{l_i}}, t_{i_{l_i}})\}, t'_i) = \mathcal{D}[\![t_i]\!]\ \{\}\ f\ (i = 1, \ldots, n) \\
\qquad \phi_i = \lambda(v_{i_1}, \ldots, v_{i_{k_i}}, v'_{i_1}, \ldots, v'_{i_{l_i}}).\ t'_i \\
\qquad t''_i = (v_{i_1}, \ldots, v_{i_{k_i}}, t_{i_1}, \ldots, t_{i_{l_i}}) \\
\qquad \psi = \lambda v_1 \cdots v_m.\text{case } t_0 \text{ of } p_1 \to (\mathsf{1}, t''_1); \cdots; p_n \to (\mathsf{n}, t''_n) \\
\qquad F_i = !\mathbf{1}, \text{ if } k_i = l_i = 0 \\
\qquad\quad = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{i_{k_i}}) \times I_1 \times \cdots \times I_{l_i}\ \ (I_1 = \cdots = I_{l_i} = I), \text{ otherwise} \\
\qquad F = F_1 + \cdots + F_n \\
\qquad \Gamma(v) = \text{return } v\text{'s type}
\end{array}
$$

$$
\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 the 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,\ \ (s_i, c_i, t'_i) = \mathcal{D}[\![t_i]\!]\ s_l\ f\ (i = 1, \ldots, 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

ssf uses functions sum, map, and $upto$ which are defined in pattern matching way as:

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

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

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

We have thus derived the following hylomorphism for *sum*:

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

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

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

where

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

4.2 Restructuring Hylomorphism

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 for ϕ and ψ to have the computation that η can do. The aim of restructuring is to extract as much computation as possible from ϕ and ψ and move them to η. This simplifies ϕ and ψ and makes the derivation of the data production and consumption scheme τ and σ easier in the next stage. A basic restructuring algorithm for the ϕ component is given in [HIT96c] 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 is omitted in [HIT96c].

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

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

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{array}{rcl}
\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\qquad \text{else } (w_1 \cup \cdots \cup w_n, (t'_1, \ldots, t'_n)) \\
 & & \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') \\
 & & \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\qquad \text{else } (w_0 \cup w_1, \text{let } v = t'_1 \text{ in } t'_0) \\
 & & \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\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) \\
 & & \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\qquad \text{else } (w_0 \cup \cdots \cup w_n, t'_0\ t'_1 \cdots, t'_n) \\
 & & \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\qquad \text{else } (w_1 \cup \cdots \cup w_n, C\ t'_1 \cdots t'_n) \\
 & & \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\qquad \text{else } (w_0 \cup w_1, t'_0\ t'_1)) \\
 & & \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\qquad \text{else } (w_0 \cup w_1 \cup w_2, [\![t'_0, t'_1, t'_2]\!]_{F_0,F_1})) \\
 & & \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')) \\
 & & \text{where } (w, t') = \mathcal{E}[\![t]\!]\, s_r \\
 & & \\
Var_{s_r}(t) & = & t \text{ is a variable } \ \wedge\ \ t \notin s_r
\end{array}$$

Figure 4 Restructuring Hylomorphism — ϕ part

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

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

Figure 5 Restructuring Hylomorphism — ψ part

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 free variables. The computations related to these subterms are then moved to the η component.

As an example, let's 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

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

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) and makes easier to apply the Acid Rain Theorem.

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

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

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 I functor). New terms t_i' work only as suppliers of free variables, and actual computation is performed by 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 $[\![\phi, \eta, \psi]\!]_{G,F} : A \rightarrow B$ 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 [HIT96c].

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

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

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

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 about derivation of τ and making brief remarks on the dual derivation of σ.

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 (a), we need ϕ to 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 a data of type μF. Our idea of deriving τ out of 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 generality 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 $F'_i = !\Gamma(v_{i_1}) \times \cdots \times !\Gamma(v_{m_i}) \times I_1 \times \cdots \times I_{l_i}$, 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 reduces 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 limit much descriptive power. We borrow a similar idea of *Canonical Terms* in [SF93] and impose the 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 it simplifies ϕ a lot by moving out many computations not relating to

[†]In fact, after restructuring, our restrictive term will not include the last form (a function application)

the construction of the resultant data of type μF. 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* structure can be embedded in hylomorphism, 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 [SF93].

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:

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

We proceed this lambda abstraction, substituting 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 substitution by applying catamorphism $([c_1 \triangledown \cdots \triangledown c_n])_F$ to the expression to replace each constructor of μF by 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 $\mathcal{F}'$ are recursively applied to its arguments.
3. When t_i is a hylo application $[\![\phi', \eta, out_F]\!]_{F',F}$, apply our main algorithm recursively to $\mathcal{F}$ (not $\mathcal{F}'$) to ϕ'.
4. When t_i is a function application $f\ t_1\ \cdots\ t_n$, we apply $([c_1 \triangledown \cdots \triangledown c_n])$.

The overall algorithm for deriving τ is summarized in Figure 6, where the main algorithm is $\mathcal{F}$ which in turn calls Algorithm $\mathcal{F}'$.

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

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

It 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

because the terms of this form will be extracted and moved to η component. We include it to this part self-contained, which might be helpful for those readers not yet familiar with our restructing algorithm.

‡So $\mathcal{F}'$ does not apply to a tuple term.

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

$$\begin{array}{lcl} \phi & : & F'\mu F \to \mu F \\ \phi & \Rightarrow & \tau in_F \quad \text{where } \tau = \mathcal{F}[\![\phi]\!] \end{array}$$

where

$$\begin{array}{lcl} \mathcal{F}[\![\phi_1 \triangledown \cdots \triangledown \phi_k]\!] & = & \lambda cs.\ (\mathcal{F}[\![\phi_1]\!]\ cs \triangledown \cdots \triangledown \mathcal{F}[\![\phi_k]\!]\ cs) \\ \mathcal{F}[\![\lambda(v_{i_1}, \cdots, v_{i_{m_i}}\ v'_{i_1}, \cdots, v'_{i_{l_i}}).\, t_i]\!] & = & \lambda(c_1 \triangledown \cdots \triangledown c_n).\lambda(v_{i_1}, \cdots, v_{i_{m_i}}\ v'_{i_1}, \cdots, v'_{i_{l_i}}).\, \mathcal{F}'[\![t_i]\!] \end{array}$$

$$\begin{array}{lcll} \mathcal{F}'[\![v]\!] & = & ([c_1 \triangledown \cdots \triangledown c_n])_F\ v & (*\ v \text{ is not a recursive variable } *) \\ \mathcal{F}'[\![v']\!] & = & v' & (*\ v' \text{ is a recursive variable } *) \\ \mathcal{F}'[\![C_i\ t'_1\ \cdots\ t'_{n'}]\!] & = & c_i(F_i\ \mathcal{F}'\ (t'_1, \cdots, t'_{n'})) & \\ \mathcal{F}'[\![[\![\phi', \eta, out_F]\!]_{F',F}\ v']\!] & = & [\![\mathcal{F}[\![\phi']\!]\ (c_1 \triangledown \cdots \triangledown c_n), \eta, out_F]\!]_{F',F}\ v' & \\ \mathcal{F}'[\![f\ t'_1\ \cdots\ t'_n]\!] & = & ([c_1 \triangledown \cdots \triangledown c_n])\ (f\ t'_1\ \cdots\ t'_n) & \end{array}$$

Figure 6 Deriving τ

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 catamorphism.

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

$$\begin{array}{ll} \text{(i)} & \phi = \mathcal{F}[\![\phi]\!]\ in_F \\ \text{(ii)} & \mathcal{F}[\![\phi]\!] : \forall A.(FA \to A) \to (F'A \to A) \end{array}$$

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 if we can proof that, for any $\phi : F'(\mu F) \to \mu F$ we have

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

according to Theorem 2. 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

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

which can be restructured into

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

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.

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

Algorithm for Deriving σ

Another case of applying the Acid Rain Theorem is to fuse $[\![_, _, \psi]\!]_{_, F'} \circ [\![in_F, _, _]\!]_{F, _}$, which needs to derive σ from the ψ component satisfying $\sigma out_F = \psi$. 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 $\psi : \mu F \to F' \mu F$ of the following restrictive form:

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

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) a data of type μF successively by out_F as follows.

$$\begin{array}{lcl} d_1 & = & out_F\ vs \\ d_2 & = & F\ out_F\ d_1 \\ & \vdots & \\ d_m & = & F^{m-1}\ out_F\ d_{m-1} \end{array}$$

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 $vs : \mu F$ 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 vs 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 an unempty list of integers is defined and transformed into a hylomorphism as:

$$\begin{array}{lcl} maximum & = & \lambda xs.\text{case } xs \text{ of} \\ & & \qquad Nil \to error \\ & & \qquad Cons\ (a, Nil) \to a \\ & & \qquad Cons\ (a, as) \to max(a, maximum\ as) \\ & = & [\![\phi, id, \psi]\!]_{F', F'} \end{array}$$

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

where

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

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

Figure 7 Deriving σ

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

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

$$\begin{array}{rcl}
\mathcal{G}'[\![Nil]\!] & = & (_,\ [(1, ())]) \\
\mathcal{G}'[\![Cons\,(a,\ Nil)]\!] & = & (_,\ q) \\
& & \text{where } (a,\ [\,]) = \mathcal{G}'[\![a]\!] \quad (_, [(1, ())]) = \mathcal{G}'[\![Nil]\!]
\end{array}$$

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

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 looks like

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

4.4 Applying Acid Rain Theorem

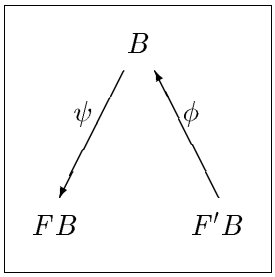
In this section, we describe how to apply the Acid Rain Theorem for fusion. Theorem 1 indicates that it must have special forms $[\![_, _, \tau in_F]\!] \circ [\![out_F, _, _]\!]$ or $[\![_, _, in_F]\!] \circ [\![\sigma out_F, _, _]\!]$ to apply this theorem. Now we are confronted with some technical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

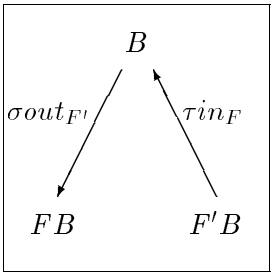
Definition 7 (Potential Fusible Composition) A term is called a *potential fusible composition* if (i) it is either in the form of $t_1 \circ t_2$ or in the form of $(t_1\ (t_2\ t))$; and (ii) the domain of t_1 is a data structure defined by an endofunctor. □

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 $h_1 \circ h_2$:

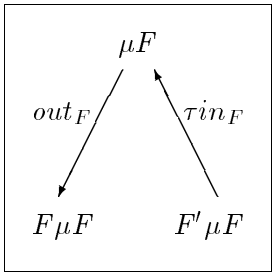
$$
\begin{array}{ll}
h_1 : B \rightarrow C & h_2 : A \rightarrow B \\
h_1 = [\![_, _, \psi]\!]_{_,F} & h_2 = [\![\phi, _, _]\!]_{F',_}
\end{array}
$$



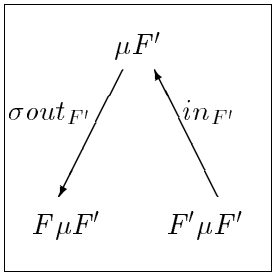
(a) ϕ and ψ in general case



(b) Fusion Impossible



(c) $B = \mu F$



(d) $B = \mu F'$

Figure 8 Relation between two hylomorphisms

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 have 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 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 composition which passes the above checking that the Acid Rain Theorem can be applied.

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 the original function ssf to the following.

$$\begin{aligned} ssf &= sum \circ map\ square \circ upto \\ &= [\![0 \triangledown (+), id, out_F]\!]_{F,F} \circ [\![in_F, id + \lambda(v, v').(square\ v, v'), out_F]\!]_{F,F} \circ [\![in_F, id, \psi]\!]_{F,F} \\ &= [\![0 \triangledown (+), id + \lambda(v, v').(square\ v, v'), out_F]\!]_{F,F} \circ [\![in_F, id, \psi]\!]_{F,F} \end{aligned}$$

$$
\begin{aligned}
&= [\![0 \triangledown (+), id + \lambda(v, v').(square\ v, v'), \psi]\!]_{F,F} \\
&= \lambda(m, n).\text{case}\ (m > n)\ \text{of} \\
&\qquad True \rightarrow 0 \\
&\qquad False \rightarrow square\ m + ssf\ (m + 1, n)
\end{aligned}
$$

4.5 Fusing inside Hylomorphisms

So far we have shown how to fuse between two hylomorphisms. We, however, have not mentioned the fusion inside a hylomorphism. 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 η part to ψ part as:

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

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

Our idea is to proceed 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 Hylomorphism

Inlining hylomorphism is the inverse process of hylomorphism derivation. All the hylomorphisms left in a program are transformed back to recursive definitions by the following step and the resultant program becomes a legal Gofer program.

$$
\begin{aligned}
f \quad = \quad & \{\text{Hylomorphism definition}\} \\
& [\![\phi, \eta, \psi]\!]_{G,F} \\
& \text{where } F = F_1 + \cdots + F_n \\
& \qquad \phi = \phi_1 \triangledown \cdots \triangledown \phi_n \\
& \qquad \eta = \eta_1 + \cdots + \eta_n \\
& \qquad \psi = \lambda v_s.\text{case}\ t_0\ \text{of}\ p_1 \rightarrow (\mathsf{1},\ t_1); \cdots ; p_n \rightarrow (\mathsf{n},\ t_n) \\
= \quad & \{\text{Inlining hylomorphism}\} \\
& \lambda v_s.\text{case}\ t_0\ \text{of}\ p_1 \rightarrow (\phi_1 \circ (\eta_1 \circ F_1\ f))\ t_1; \cdots ; p_n \rightarrow (\phi_n \circ (\eta_n \circ F_n\ f))\ t_n
\end{aligned}
$$

Note a simple symbolic simplifier is necessary here to remove unnecessary compositions.

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	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

Figure 9 Experiment results (*unlines* applies to a text with 100 words)

5 PERFORMANCE EVALUATION

All algorithms reported in this paper have been implemented. To evaluate the effectiveness of 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 string and returns a new string by concatenating all strings separated by the CR character. *queen* is the well-known queen program.

These programs were evaluated using Gofer (Version 2.30a) interpreter. The experiment results are given in Figure 9, 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 queen problem in Figure 10. Its non-triviality shows the power of our HYLO system. It should be noted that our improvement for *queens* came out automatically without any special treatment of list comprehension as in [GLJ93], i.e., we just remove list comprehension by a naive transformation scheme in terms of *concat*, *map*, *filter* [Jon87]. This example was also used in [GLJ93] where a cheap fusion was applied (but the improved program was not given at all). The main difference lies in that the cheap fusion needs one to turn all recursions in terms of *build/cata* form by hand before fusion, which turns out to be unnecessary in our HYLO system.

6 DISCUSSIONS AND CONCLUDING REMARKS

It has been argued that programming with the use of generic control structures which capture patterns of recursions in a uniform way is very significant in program transformation and optimization[GLJ93, MFP91, SF93, LS95, TM95]. Our work is much related to these studies. In particular, our work was greatly motivated by Sheard and Fegaras' [SF93] and Takano and Meijer's [TM95]. Sheard and Fegaras implemented a fusion algorithm called *normalization algorithm* which can work with 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

```
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 10 Initial and Transformed *queens*' Programs

[GLJ93] 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.

This work extends our previous work [HIT96c] 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 function τ and σ implementing the transformation laws given in [HIT96c]. Third, also the most important, we are implementing HYLO, stepping towards 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. The HYLO is hoped to be embedded in the practical Haskell system [HAB+95].

All transformation algorithms introduced in this paper have been implemented. It is completely mechanical and does not rely on heuristics 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 [HIT96d], mutual recursions, and etc. in order to enable more fusion. In a 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 [HIT96b], accumulating[HIT96a] as well as other optimization tactics.

ACKNOWLEDGEMENT

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

REFERENCES

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

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

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

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

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

[HAB+95] Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Jon Fairbairn, Joseph Fasel, Andy Gordon, María M. Guzmán, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Dick Kieburtz, Rishiyur Nikhil, Will Partain, John Peterson, Simon Peyton Jones, and Philip Wadler. *Report on the Programming Language Haskell, Version 1.3*, 1995.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. Caculating accumulations. Technical Report METR 96–03, Faculty of Engineering, University of Tokyo, March 1996.

[HIT96b] Z. Hu, H. Iwasaki, and M. Takeichi. Cheap tupling in calculational form. Technical Report METR 96–08, Faculty of Engineering, University of Tokyo, October 1996. Poster abstract appears in PLILP'96, LNCS 1140, pages 471–472.

[HIT96c] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 73–82, Philadelphia, PA, May 1996. ACM Press.

[HIT96d] Z. Hu, H. Iwasaki, and M. Takeichi. An extension of the Acid Rain Theorem. In *2nd Fuji International Workshop on Functional and Logic Programming*, Shonan, Japan, November 1996. World Scientific.

[Jon87] S.B. Jones, editor. *Investigation of performance achievable with highly concurrent interpretation of functional programs*. ESPRIT 302 final report, October 1987.

[Jon94] Mark P. Jones. Gofer 2.30a release notes. 1994.

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

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

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

[PP93] A. Pettorossi and M. Proietti. Rules and strategies for program transformation. In *IFIP TC2/WG2.1 State-of-the-Art Report*, pages 263–303. LNCS 755, 1993.

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

[Tea96] The AQUA Team. *Glasgow Haskell Compiler User's Guide ver0.29*, 1996.

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

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