Cheap Tupling Transformation

Zhenjiang Hu [*] Hideya Iwasaki [†] Masato Takeichi [‡]

Summary.

Tupling is a well-known transformation tactic to obtain new recursive functions without redundant recursive calls and/or multiple traversals of common data, which is achieved by grouping some recursive functions into a tuple. Although being studied for a long time based on fold/unfold transformation, it suffers from the high cost of keeping track of function calls and has to use clever control to avoid infinite unfolding, which prevent it from being used in a practical compiler of functional languages. In this paper, we propose a cheap tupling based on the theory of constructive algorithmics. We give several simple but effective calculational rules, which not only can be successfully applied to improve a lot of interesting recursive functions but also can be implemented efficiently.

KEYWORDS Program Analysis and Program Transformation
Reasoning about Language Constructs
Executable Specification

1 Introduction

In functional programming, a program *prog* is usually expressed as compositions of transformations over data structures while each transformation is defined by a recursion $\mathcal{R}_i$ traversing over its input data structure, namely

$$prog = \mathcal{R}_1 \circ \cdots \circ \mathcal{R}_n.$$

This compositional style of programming allows clearer and more modular programs, but comes at a price of possibly high runtime overhead resulting mainly from the following two categories:

- unnecessary intermediate data structures passed between the composition of two recursions;

[*] Department of Information Engineering, University of Tokyo (hu@ipl.t.u-tokyo.ac.jp) .
[†] Department of Computer Science, Faculty of Technology, Tokyo University of Agriculture and Technology (iwasaki@ipl.ei.tuat.ac.jp) .
[‡] Department of Mathematical Engineering and Information Physics, Faculty of Engineering, The University of Tokyo (takeichi@u-tokyo.ac.jp) .

- inefficiency in a single recursion, such as redundant recursive calls, multiple traversals of data structures, and unnecessary traversals of intermediate data structures (see Section 2).

Although this paper is mainly concerned with elimination of the inefficiency in the latter case, these two kinds of inefficiency are much related, for which there are two known tactics, namely *Fusion* (or called *deforestation)* [Wad88, Chi92] and *Tupling* [Chi93]. Fusion is to merge nested compositions of recursive functions in order to obtain new recursions without unnecessary intermediate data structures, while tupling is to remove redundant recursive calls and multiple traversals of the same data structure from recursions.

Different approaches are employed to formulate *fusion* and *tupling*. One, extensively studied by Chin [Chi92, Chi93], is based on the so-called *fold/unfold transformation*[BD77]. It, however, suffers from the high cost of keeping track of function calls and has to use clever control to avoid infinite unfolding, which prevents fusion and tupling being embedded in a real practical compiler of functional languages. To overcome this difficulty, quite a lot of studies have been devoted to another approach called *transformation in calculational form* [GLJ93, SF93, TM95, HIT96b] based on the theory of *Constructive Algorithmics* [Fok92]. It makes use of the recursive structure information in some specific forms such as *catamorphisms* (or called *folds*), *anamorphisms* (or called *unfolds*) and *hylomorphisms* and finds how transformation can be performed over them.

The latter approach, being less general than the former, has been successfully applied to the fusion transformation. It is based on a quite simple *Acid Rain* calculational rule [TM95], an extension of the *shortcut deforestation* rule [GLJ93], and can be practically employed in a real compiler of functional languages (e.g., Glasgow Haskell Compiler). However, so far as we know, no attempt has been made to adapt this calculational approach to the tupling transformation for the improvement of recursive functions. We believe it is worth doing for two reasons. First, tupling and fusion are two most related transformation tactics, so they should be studied in the same framework. In fact, the roles of tupling and fusion are complementary; fusion merges compositions of recursions into one which then should be improved again by tupling in order to obtain a final efficient program. Second, with the same reason for the shortcut deforestation [GLJ93, TM95], tupling should be used practically in a real compiler. So far, we haven't seen a real practical compiler which performs tupling transformation.

In this paper, we propose the idea of *cheap tupling*[1] in a calculational way. Our main contributions are as follows.

- We identify the importance of the relationship between tupling transformation and *structural* mutual recursions (not a simple mutual recursion) in Section 4, based on which we propose three simple but effective calculational rules (Theorem 3, 7 and 9) for our cheap tupling transformation. As will be seen, they can be applied to improve a wide class of recursions (though not

[1] We call it *cheap* tupling after the name of *shortcut* deforestation.

all) through the elimination of redundant recursive calls, multiple traversals of the same data structures, and unnecessary traversals of data structures which hasn't been noticed before.

- As discussed above, our cheap tupling follows the approach of transformation in calculational form based on constructive algorithmics. This is in sharp contrast to the previous study [Chi93] based on fold/unfold transformation. Therefore, our cheap tupling preserves the advantages of transformation in calculational form, as we have seen in the discussion of shortcut deforestation in [TM95].

  - It can be applied to the improvement of recursions over any data structures, other than recursions over lists.

  - Each of our transformation rules is an automation of the unfold-simplify-fold method without the intervention of explicit laws. Therefore, our transformations are guaranteed to terminate and would be more practical for being used in a compiler.

- Our cheap tupling is suitable to coexist with the shortcut deforestation (in calculational form). It will be seen that the combination of the two improvement is direct and natural (Section 5). In contrast, the previous study on this combination based on fold/unfold transformation [Chi95] is much more difficult because of complicated control of infinite unfoldings in the case where fusion and tupling are applied simultaneously.

The organization of this paper is as follows. In Section 2, we use several simple examples to show what kind of inefficient recursions we'd like to improve. After giving a brief introduction of the concept of the transformation in calculational form in Section 3, we propose three simple but effective rules for our cheap tupling in Section 4. In Section 5, we give the cheap tupling strategy based our simple rules, and see how cheap tupling can coexist with fusion transformation. Related works and conclusions are discussed in Section 6 and 7 respectively.

2 Examples: Inefficient Recursions

In this Section, we shall use several simple examples to show what kind of inefficiency may occur in a recursion, and how an efficient version can be derived. As will be seen later, the efficiency is always achieved at the cost of clarity and conciseness compared with the original one. We would like to write our programs concisely, but have the compiler automatically make them efficient.

2.1 Multiple Traversals of Data Structures

Consider the function *deepest*, which finds a list of leaves that are farthest away from the root of a given tree, may be defined as follows.

$$\begin{array}{lll} deepest\ (Leaf\ a) & = [a] \\ deepest\ (Node(l,r)) & = deepest(l), & depth(l) > depth(r) \\ & = deepest(l) +\!\!+ deepest(r), & depth(l) = depth(r) \\ & = deepest(r), & \text{otherwise} \\ depth\ (Leaf\ a) & = 1 \\ depth\ (Node(l,r)) & = 1 + max(depth(l), depth(r)) \end{array}$$

The infix binary function $+\!\!+$ concatenates two lists and the function *max* gives the maximum of the two arguments. Function *deepest* uses another recursive function *depth*. Being concise, this definition is quite inefficient because *deepest* and *depth* traverse over the same inputs leading to many repeated computations in calculating the depth of subtrees. We'd like to eliminate this multiple traversal and have the following efficient program.

$$\begin{array}{lll} deepest'\ t & = u \text{ where } (u,v) = dd\ t \\ dd\ (Leaf\ a) & = ([a], 0) \\ dd\ (Node\ l\ r) & = (dpl, 1 + dl), & dl > dr \\ & = (dpl +\!\!+ dpr, 1 + dr), & dl = dr \\ & = (dpr, 1 + dr), & \text{otherwise} \\ & \quad \text{where } (dpl, dl) = dd\ l;\ (dpr, dr) = dd\ r \end{array}$$

2.2 Redundant Recursive Calls

A classical example to illustrate the super-linear speedup achieved when redundant recursive calls are eliminated is the fibonacci function:

$$\begin{array}{lll} fib\ Zero & = Zero \\ fib\ (Succ(Zero)) & = Succ(Zero) \\ fib\ (Succ(Succ(n))) & = plus(fib(Succ(n)), fib(n)) \end{array}$$

The fib is a recursion over the natural number data type:

$$N = Zero \mid Succ(N).$$

This definition gives an inefficient exponential algorithm *fib* because of many redundant recursive calls to *fib*. In previous studies, in order to make it efficient, a creative tupling function $f'(Succ(n)) = (fib(Succ(n)), fib(n))$ has to be defined, and then the transformation based on fold/unfold is applied to improve f' for sharing common computation and remove redundant recursive calls. We would like to show that the redundant recursive calls can be removed by a simple calculation with our approach (Section 4).

2.3 Unnecessary Traversals of Intermediate Results

Given a recursion $\mathcal{R}$, we know that unnecessary intermediate data structures, produced by $\mathcal{R}$, occur when composed with another recursion $\mathcal{R}'$, i.e., $\mathcal{R}' \circ \mathcal{R}$. By

fusion, one can merge the two recursions into one without such unnecessary intermediate data structures.

It would be surprising to see that even in a single recursion, there may remain some unnecessary traversals of intermediate data structures produced by $\mathcal{R}$. As an example, consider the recursive function *foo* recursively defined by

$$\begin{array}{lll} foo\ Nil & = & Nil \\ foo\ (Cons(x, xs)) & = & Cons(x + sum\ p, p) \\ & & \text{where } p = foo\ xs \end{array}$$

where *sum* is used to sum up all elements in a list. Although *foo* is defined in a single recursion, it is inefficient because the intermediate results produced by *foo* are traversed by *sum*, which is actually unnecessary. We could eliminate this unnecessary traversal and obtain the following efficient one.

$$\begin{array}{lll} foo\ xs & = & i, \quad \text{where } (i, s) = f'\ xs \\ f'\ Nil & = & (Nil, 0) \\ f'\ (Cons(x, xs)) & = & (Cons(x + s,\ i),\ x + s + s) \\ & & \text{where } (i, s) = f'\ xs \end{array}$$

3 Transformation in Calculational Form

Before addressing how to calculate efficient recursions, we review previous work on the transformation in calculational form [MFP91, SF93, TM95, HIT96b]. Its theoretical basis can be found in the study of *Constructive Algorithmics*[Fok92, Mal90, MFP91] which will be outlined below. Throughout this paper, our default category C is a $\mathcal{CPO}$, the category of complete partial orders with continuous functions.

3.1 Functors

Endofunctors on category C (functors from C to C) are used to capture both data structure and control structure in a type definition. In this paper, we assume that all the 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.

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

□

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

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

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.

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

Some related operators are:

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

□

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

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

Some related operators are:

$$\begin{aligned} \iota_1\ a &= (1, a) \\ \iota_2\ b &= (2, b) \\ (f \triangledown g)\ (1, x) &= f\, x \\ (f \triangledown g)\ (2, y) &= g\ y. \end{aligned}$$

□

Although the product and the separated sum are defined over 2 parameters, they can be naturally extended for n parameters. For example, the separated sum over n parameters can be defined by

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

3.2 Data Types as Initial Fixed Points of Functors

A data type is a collection of operations (data constructors) denoting how each element of the data type can be constructed in a finite way. Via these data constructors, functions on the type may be defined. So a data type is a particular algebra whose one distinguished property is categorically known as the initiality of the algebra. Let C be a category and F be an endofunctor from C to C.

Definition 5 (F-algebra)
An *F-algebra* is a pair (X, ϕ), where X is an object in C, called the *carrier* of the algebra, and ϕ is a morphism from object $F\,X$ to object X denoted by $\phi : F\,X \rightarrow X$, called the *operation* of the algebra. □

Definition 6 (F-homomorphism)
Given two F-algebras (X, ϕ) and (Y, ψ), the F *homomorphism* from (X, ϕ) to (Y, ψ) is a morphism h from object X to object Y in category C satisfying $h \circ \phi = \psi \circ F\, h$.
□

Definition 7 (Category of F-algebras)
The *category of F-algebras* has as its objects the F-algebras and has as its morphisms all F-homomorphisms between F-algebras. Composition in the category of F-algebra is taken from C, and so are the identities. □

It is known that the initial object in the category of F-algebras exists when F is a polynomial functor[Mal90]. The representative for the initial algebra is denoted by μF. Let $(T, in_F) = \mu F$, μF defines a data type T with the *data constructor* $in_F : F\,T \to T$. Function $out_F : T \to F\,T$ is the inverse of in_F and it destructs its argument and is therefore called *data destructor*. To be concrete, consider the data type of cons lists given by the following definition with elements of type A:

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

It is categorically defined as the initial object of

$$(List\ A,\ Nil \triangledown Cons)$$

in the category of F_{L_A}-algebras[2], where F_{L_A} is the endofunctor defined by $F_{L_A} = !\mathbf{1} + !A \times I$ ($\mathbf{1}$ is the final object in C). Here, the data constructor and the data destructor are as follows.

$$\begin{aligned} in_{F_{L_A}} &= Nil \triangledown Cons \\ out_{F_{L_A}} &= \lambda xs.\ \text{case } xs \text{ of} \\ &\qquad\qquad Nil\ \to (1, ()); \\ &\qquad\qquad Cons\ (a, as) \to (2, (a, as)) \end{aligned}$$

3.3 Cata, Ana, and Hylo: Recursions over Data Types

In constructive algorithmics, data types are categorically defined as initial algebras of functors, and recursive functions from one data type to another are represented as structure-preserving maps between algebras. By doing so, an orderly structure can be imposed on the program and such structure can be exploited to facilitate program transformation.

The most general structure-preserving maps are *Hylomorphisms* [MFP91], a composition of *catamorphisms* and *anamorphisms*. To take into consideration natural transformations between data structures, Takano and Meijer defined hylomorphisms in triplet form[TM95] as follows.

Definition 8 (Hylomorphism in triplet form)
Given two morphisms $\phi : G\,A \to A$, $\psi : B \to F\,B$ and natural transformation $\eta : F \dot{\to} G$, the hylomorphism $[\![\phi, \eta, \psi]\!]_{G,F}$ is defined as the least morphism $f : B \to A$ satisfying the following equation.

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

□

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

Hylomorphisms (Hylo for short) are powerful in description in that practically every recursion of interest (e.g., primitive recursions) can be specified by them [BdM94, HIT96b]. They are considered to be an ideal intermediate recursive form for calculating efficient functional programs.

Definition 9 (Catamorphism, Anamorphism, Map)
Let $(T_F, in_F) = \mu F$, $(T_G, in_G) = \mu G$.

$$\begin{array}{lcl} (\!|\_|\!)_F & : & \forall A.\ (F\ A \to A) \to T_F \to A \\ (\!|\phi|\!)_F & = & [\![\phi, id, out_F]\!]_{F,F} \end{array}$$

$$\begin{array}{lcl} [\!(\_)\!]_F & : & \forall A.\ (A \to F\ A) \to A \to T_F \\ [\!(\psi)\!]_F & = & [\![in_F, id, \psi]\!]_{F,F} \end{array}$$

$$\begin{array}{lcl} \langle\!|\_|\!\rangle_{G,F} & : & (F \dot{\to} G) \to T_F \to T_G \\ \langle\!|\eta|\!\rangle_{G,F} & = & [\![in_G, \eta, out_F]\!]_{G,F} \end{array}$$

□

Catamorphisms (Cata for short) $(\!|\_|\!)$ are generalized foldr operators (or reduces) that substitute the constructor of a data type with other operation of the same signature. Dually, anamorphisms (ana for short) $[\!(\_)\!]$ are generalized unfold operators (or generations). Maps $\langle\!|\_|\!\rangle$ apply a natural transformation on the data structure.

Hylomorphisms enjoy many useful transformation laws[3]. One useful law is called *Hylo shift law*:

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

showing that a natural transformation can be shifted inside a hylomorphism.

For fusion, hylomorphisms possess the general laws called the *hylo fusion laws*, and the specific calculational rules for shortcut deforestation as in the *Acid Rain* Theorem.

Theorem 1 (Hylo Fusion)

$$\begin{array}{ll} \text{Left Fusion Law:} & f \circ \phi = \phi' \circ F\ f \implies f \circ [\![\phi, \eta, \psi]\!]_{F,G} = [\![\phi', \eta, \psi]\!]_{F,G} \\ \text{Right Fusion Law:} & \psi \circ g = G\ g \circ \psi' \implies [\![\phi, \eta, \psi]\!]_{F,G} \circ g = [\![\phi, \eta, \psi']\!]_{F,G} \end{array}$$

□

Theorem 2 (Acid Rain)

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

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

□

[3] We don't list the transformation laws for catamorphisms and anamorphisms, because they can be deduced from those for hylomorphisms.

4 Cheap Tupling Rules in Calculational Form

Tupling achieves efficient recursive functions through elimination of redundant recursive calls and multiple traversals of common inputs. In this section, we propose three simple but effective calculational rules for our cheap tupling.

We shall start by examing the relationship between tupling and mutual recursive definitions and propose our *basic* calculational rule (Theorem 3) on how to perform tupling transformation on mutual recursions. Based on it, we give another two calculational rules for removing redundant recursive calls and unnecessary intermediate data structures respectively. We shall also demonstrate how the rules work practically.

Mutual Recursions and Tupling

There has been a folklore that mutual recursive definitions can be turned into a single non-mutual recursive definition if the functions mutually defined are tupled. Take as an example the following mutual recursive definitions:

$$\begin{aligned} f &= C_1[g,\ f] \\ g &= C_2[f,\ g] \end{aligned}$$

where C_i denotes a context. It says that f calls g and f, and g calls f and g too. One can turn this mutual recursion into non-mutual one by tupling the functions f and g to another function h:

$$h = f \vartriangle g.$$

It follows that

$$\begin{aligned} f &= \pi_1 \circ h \\ g &= \pi_2 \circ h \end{aligned}$$

and that h becomes a non-mutual recursion as follows.

$$h = C_1[\pi_2 \circ h,\ \pi_1 \circ h] \vartriangle C_2[\pi_1 \circ h,\ \pi_2 \circ h]$$

This transformation is interesting in theory in the sense that one need not consider the transformation for mutual recursive functions because they can definitely be turned into a single non-mutual one. It is, however, unsatisfactory in practice as it is not clear how to make h efficient. Generally, the new definition h costs more than the direct implementation of mutual recursions if no further simplification is applied. This is why many compilers, such as Glasgow Haskell, implement mutual recursions directly without such transformation.

Things become expected when the recursive structure information of f and g are known. Such structural information can be exploited to make the tupled function efficient, as shown in the following theorem.

Theorem 3 (Tupling)

$$\frac{f \circ in_F = \phi \circ F(f \vartriangle g),\ \ g \circ in_F = \psi \circ F(f \vartriangle g)}{f \vartriangle g = ([\phi \vartriangle \psi])_F}$$

Proof: We prove it by the following calculation.

$$\begin{array}{ll} & (f \vartriangle g) \circ in_F \\ = & \quad \{ \quad \vartriangle \quad \} \\ & f \circ in_F \vartriangle g \circ in_F \\ = & \quad \{ \text{ Assumptions for } f \text{ and } g \quad \} \\ & \phi \circ F(f \vartriangle g) \vartriangle \psi \circ F(f \vartriangle g) \\ = & \quad \{ \quad \vartriangle \quad \} \\ & (\phi \vartriangle \psi) \circ F(f \vartriangle g) \end{array}$$

It soon follows that

$$f \vartriangle g = ([\phi \vartriangle \psi])_F$$

according to the definition of catamorphisms (Definition 9). □

The Tupling Theorem is quite simple, and some similar studies can be found in [Tak87, Fok92]. What interest us is its significant use in calculating recursions to efficient ones, which has not yet received its worthy consideration. For this purpose, we generalize the Tupling Theorem from one step of unfolding of input to n step as follows.

Corollary 4 Given the following mutual recursive definitions for f and g:

$$\begin{array}{l} f \circ in_F = \phi \circ (F(f \vartriangle g) \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^n(f \vartriangle g) \circ out_F^{n-1}) \\ g \circ in_F = \psi \circ (F(f \vartriangle g) \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^n(f \vartriangle g) \circ out_F^{n-1}) \end{array}$$

then f and g can be tupled as

$$(f \vartriangle g) \circ in_F = (\phi \vartriangle \psi) \circ (F(f \vartriangle g) \vartriangle F^2(f \vartriangle g) \circ out_F \vartriangle \cdots \vartriangle F^n(f \vartriangle g) \circ out_F^{n-1})$$

where[4] $F^n = F^{n-1} \circ F$ and $out_F^n = F^{n-1} out_F \circ g out_F^{n-1}$. □

In the following, we shall show how the simple calculational rules can be effectively used for our cheap tupling to handle the inefficient recursions in Section 2.

4.1 Eliminating Multiple Data Traversals

The Tupling Theorem reads that, if f and g are recursive functions *traversing over the same data structures* in a certain uniform way, then tupling them will definitely give a catamorphism without multiple traversals over the same data structures by both f and g. Therefore, direct use of the tupling theorem can help to eliminate multiple data traversals in a recursion.

To see how the Tupling Theorem works, let's recall the definition of *deepest* given in Section 2. Since *deepest* and *depth* are mutually defined and traverse over the same input tree, we can apply the Tupling Theorem to calculate them into an

[4] Note out_F^i can be considered as unfolding i steps of input data.

efficient one. First, we rewrite them to be our required form.

$$\begin{array}{lll} deepest \circ in_{F_T} & = \phi \circ F_T(deepest \vartriangle depth) & \\ & \text{where} & \\ & \quad \phi = \phi_1 \triangledown \phi_2 & \\ & \quad \phi_1\ a = [a] & \\ & \quad \phi_2\ ((tl, hl), (tr, hr)) & = tl, \quad \text{if } hl > hr \\ & & = tl +\!\!+ tr, \quad \text{if } hl = hr \\ & & = tr, \quad \text{otherwise} \\ depth \circ in_{F_T} & = \psi \circ F_T(deepest \vartriangle depth) & \\ & \text{where} & \\ & \quad \psi = \psi_1 \triangledown \psi_2 & \\ & \quad \psi_1\ a = 1 & \\ & \quad \psi_2\ ((tl, hl), (tr, hr)) = 1 + max(hl, hr) & \end{array}$$

where $F_T = !Int\ +\ I \times I$ (the functor defining the binary tree type) and $in_{F_T} = Leaf \triangledown Node$. Now, according to the Tupling Theorem, we get the following efficient linear recursion:

$$deepest = \pi_1 \circ (deepest \vartriangle depth) = \pi_1 \circ ([\phi \vartriangle \psi])_{F_T}$$

which can be in-lined to the one as given in Section 2.

It should be noted that the above two processing steps, namely rewriting into the required form and applying the Tupling Theorem, can be done automatically at a low cost; the first step is basically an abstraction of recursive calls for the definition of ϕ (similar to the study in [HIT96b]) while the second step is just a simple calculation. One should compare with the previous expensive approach on the basis of fold/unfold transformation [Chi92], where it is required to keep alert on any sub-expression which could be folded and to define many new functions in order to remember occurred sub-expressions during its transformation process.

In the Tupling Theorem, f and g are defined mutually. One special interesting case is when they are independent catamorphisms. They can be tupled as well, as stated in the following corollary.

Corollary 5

$$([\phi])_F \vartriangle ([\psi])_F = ([\phi \circ F\pi_1 \vartriangle \psi \circ F\pi_2])_F$$

Proof. Directly from the Tupling Theorem and the following two equations.

$$\begin{array}{l} ([\phi])_F \circ in_F = \phi \circ F\pi_1 \circ F(([\phi])_F \vartriangle ([\psi])_F) \\ ([\psi])_F \circ in_F = \psi \circ F\pi_2 \circ F(([\phi])_F \vartriangle ([\psi])_F) \end{array}$$

□

This corollary can reduce two traversals of the input (by the two catamorphisms respectively) into one.

4.2 Eliminating Redundant Recursive Calls

It is impractical to eliminate all redundant recursive calls in recursions as done by the most general approach called *memoization*[Mic68]. Therefore some restrictions

on recursions are necessary. For instance, Chin [Chi93] restricted his method on **T0** class of recursions; Hughes [Hug85] argued that it is more practical to eliminate the redundant recursive calls that are applied to exactly the identical arguments – that is, arguments stored in the same place in memory.

The restriction we impose on the recursive definitions, much related to Hughes' restriction, is that the parameters of the recursive calls to the defined function should be the sub-structure of the input data without any processing. This restriction helps to verify the identity of two parameters at the stage of compilation (rather than seeing if the arguments are stored in the same place in memory at execution time as Hughes did). For example, we can treat the recursive definition like

$$foo(Cons(x_1, Cons(x_2, xs))) = x_1 + foo(Cons(x_2, xs)) + foo(xs)$$

but not

$$foo(Cons(x_1, Cons(x_2, xs))) = x_1 + foo\underline{(Cons(2 * x_2, xs))} + foo\underline{(Cons(x_1, xs))}$$

because the two underlined parameters are not sub-structures of $Cons(x_1, Cons(x_2, xs))$.

Another restriction, just for the sake of simple presentation, is that the input is only traversed by a single function. For example, our restriction excludes the following case

$$foo(Cons(x_1, Cons(x_2, xs))) = x_1 + foo(Cons(x_2, xs)) + foo(xs) + g(x2 : xs)$$

because both foo and g traverse over the same input. As a matter of fact, this restriction is unnecessary because we are always be able to tuple foo and g to have a new function meeting this restriction.

Proposition 6 Let $h :: T \to R$ be a recursively defined function over T, where T is defined by functor F, i.e., $(T, in_F) = \mu F$. If (1) every parameter of all recursive calls to h is the sub-structure of the input; and (2) there is no other function traversing over the part or the whole of h's input, then h can be transformed into the following hylomorphism:

$$h = [\![\phi, id, out_F \vartriangle out_F^2 \vartriangle \cdots \vartriangle out_F^n]\!]_{G,G}$$

where $G = F \times F^2 \times \cdots \times F^n$.

Proof Sketch: According to the restriction of parameters of recursive calls to h, we can see that each recursive call to h can be embedded in the term of $F^i h \circ out_F^i$ for an integer i (Recall that out_F^i can be considered as unfolding i steps of input data). In addition, since no other functions traversing h's input, it implies that the input is only traversed by h. Thus, there must exist a ϕ so that h can be expressed as

$$h = \phi \circ (Fh \circ out_F \vartriangle F^2 h \circ out_F^2 \vartriangle \cdots \vartriangle F^n h \circ out_F^n)$$

which is exactly as we expected. □

Now, our theorem for removing redundant recursive calls in the above restricted recursive functions is given below.

Theorem 7 Let $h :: T \to R$ be the hylomorphism given in Proposition 6. Then, h can be defined by the following mutual recursive definitions.

$$\begin{array}{ll} h \circ in_F & = \phi \circ \eta \circ F(h \vartriangle g_1 \vartriangle \cdots \vartriangle g_{n-1}) \\ g_1 \circ in_F & = Fh \\ g_2 \circ in_F & = Fg_1 \\ & \vdots \\ g_{n-1} \circ in_F & = Fg_{n-2} \end{array}$$

where η is a natural transformation defined by

$$\begin{array}{l} \eta :: F(X_1 \times \cdots \times X_n) \to (FX_1 \times \cdots \times FX_n) \\ \eta = F\pi_1 \vartriangle \cdots \vartriangle F\pi_n \end{array}$$

Proof: This can be proved by the following calculation.

$$\begin{array}{ll} & h \circ in_F = \phi \circ \eta \circ F(h \vartriangle g_1 \vartriangle \cdots \vartriangle g_{n-1}) \\ \equiv & \quad \{ \text{ unfolding } g_i \text{ one time } \} \\ & h \circ in_F = \phi \circ \eta \circ F(h \vartriangle Fh \circ out_F \vartriangle \cdots \vartriangle Fg_{n-2} \circ out_F) \\ \equiv & \quad \{ \text{ repeat the above unfolding to remove } g_i \} \\ & h \circ in_F = \phi \circ \eta \circ F(h \vartriangle Fh \circ out_F \vartriangle \cdots \vartriangle F^{n-1}h \circ out_F^{n-1}) \\ \equiv & \quad \{ \eta \} \\ & h \circ in_F = \phi \circ (Fh \vartriangle F(Fh \circ out_F) \vartriangle \cdots \vartriangle F(F^{n-1}h \circ out_F^{n-1})) \\ \equiv & \quad \{ \text{ functor } F,\ in_F^{-1} = out_F,\ \vartriangle \text{ and } \circ, \text{ def. of } G \} \\ & h = \phi \circ Gh \circ (out_F \vartriangle Fout_F \circ out_F \vartriangle \cdots \vartriangle Fout_F^{n-1} \circ out_F) \\ \equiv & \quad \{ \ Fout_F^i \circ out_F = out_F^{i+1}, \text{ as proved later. } \} \\ & h = \phi \circ Gh \circ (out_F \vartriangle out_F^2 \vartriangle \cdots \vartriangle out_F^n) \\ \equiv & \quad \{ \text{ def. of hylo } \} \\ & h = [\![\phi, id, out_F \vartriangle out_F^2 \vartriangle \cdots \vartriangle out_F^n]\!]_{G,G} \end{array}$$

To complete the proof, we prove that $Fout_F^i \circ out_F = out_F^{i+1}$ by induction on i. For the base case of $i = 0$, it is obviously true. For the inductive case, we calculate it as follows.

$$\begin{array}{ll} & Fout_F^i \circ out_F \\ = & \quad \{ \text{ def. of } out_F^i \ \} \\ & F(F^{i-1}out_F \circ out_F^{i-1}) \circ out_F \\ = & \quad \{ \text{ functor } F \} \\ & F^i out_F \circ Fout_F^{i-1} \circ out_F \\ = & \quad \{ \text{ induction } \} \\ & F^i out_F \circ out_F^i \\ = & \quad \{ \text{ def. of } out_F^{i+1} \ \} \\ & out_F^{i+1} \end{array}$$

□

Theorem 7 says that, if every parameter of a recursive call to function h is only the sub-structure of the input and no other functions traverse over the same input of h, than h can be successfully transformed to a single mutual recursive definition which could be further improved by the Tupling Theorem, as stated in the following corollary.

Corollary 8 Under the same assumption in Theorem 7, we have

$$h \vartriangle g_1 \vartriangle \cdots \vartriangle g_{n-1} = (\![\phi \circ \eta \vartriangle F\pi_1 \vartriangle \cdots \vartriangle F\pi_{n-1}]\!)_F$$

□

Before showing how the theorem works practically, let's define some useful natural transformations as follows.

$$\begin{array}{lll} dist & :: & (X \times (Y + Z)) \to (X \times Y + X \times Z) \\ dist(x, (1, y)) & = & (1, (x, y)) \\ dist(x, (2, z)) & = & (2, (x, z)) \end{array}$$

$$\begin{array}{lll} zipSum & :: & (X_1 + Y_1) \times (X_2 + Y_2) \\ zipSum((1, x_1), (1, x_2)) & = & (1, (x_1, x_2)) \\ zipSum((2, y_1), (2, y_2)) & = & (2, (y_1, y_2)) \end{array}$$

Now recall the definition of $fib :: N \to N$ in Section 2. According to Proposition 6, we can transform it to the following hylomorphism.

$$fib = [\![\phi, id, out_{F_N} \vartriangle out^2_{F_N}]\!]_{G,G}$$

Here

$$\begin{array}{l} \phi = (Zero \triangledown ((Succ(Zero) \triangledown plus) \circ dist)) \circ zipSum \\ G = F_N \times F_N^2 \end{array}$$

In addition, we remind readers that

$$\begin{array}{ll} F_N & = !\mathbf{1} + I \\ F_N^2 & = !\mathbf{1} + (!\mathbf{1} + I) \\ out_{F_N} & = \lambda x.\ \text{case } x \text{ of } Zero \to (1, ());\ Succ(n) \to (2, n) \\ out^2_{F_N} & = \lambda x.\ \text{case } x \text{ of } Zero \to (1, ());\ Succ(n) \to (2, case\ n\ of\ Zero \to (1, ());\ Succ(n') \to (2, n')) \end{array}$$

By Theorem 7 and Corollary 8, we soon have that

$$fib = \pi_1 \circ (\![\phi \circ (F_N\, \pi_1 \vartriangle F_N\, \pi_2) \vartriangle F_N\, \pi_1]\!)_{F_N}.$$

Simple simplification gives

$$fib = \pi_1 \circ (\![(Zero \triangledown ((Succ(Zero) \triangledown\ plus) \circ dist)) \vartriangle F_N\, \pi_1]\!)_{F_N}$$

which could be in-lined to the following linear program.

$$\begin{array}{lll} fib\ n & = & x, \quad \text{where } (x, y) = f'\ n \\ f'\ Zero & = & (Zero, (1, ())) \\ f'\ (Succ(n)) & = & (\text{case i of } 1 \to Succ(Zero); 2 \to plus(x, y), (2, x)) \\ & & \text{where } (x, (\mathsf{i}, y)) = f'\ n \end{array}$$

4.3 Removing Unnecessary Traversals of Intermediate Results

It is commonplace in a recursive definition where some intermediate results are traversed by another recursive function. We can generally formulate it by the equation:

$$f \circ in_F = \phi \circ F(f \vartriangle (g \circ f)). \tag{1}$$

where the intermediate results of f is not only used for the final results (the first f in the RHS) but also traversed by another recursive function g (the second f in RHS).

In fact, the traversals of intermediate results by g can be calculated away under a certain condition leading to a more efficient recursion. This is shown in the following theorem.

Theorem 9 Given is the recursive function f defined by Equation (1). If there exists ψ satisfying

$$g \circ \phi = \psi \circ F((g \vartriangle id) \times id) \tag{2}$$

then

$$f = \pi_1 \circ ([\phi \vartriangle (\psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2))])_F$$

Proof: It Suffices to prove that

$$f \vartriangle (g \circ f) = ([\phi \vartriangle (\psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2))])_F.$$

According to the Tupling Theorem, it is only need to prove

$$(g \circ f) \circ in_F = (\psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2)) \circ F(f \vartriangle (g \circ f)).$$

To this end, we do calculation as follows.

$$\begin{array}{ll} & g \circ f \circ in_F = \psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2) \circ F(f \vartriangle (g \circ f)) \\ \equiv & \quad \{ \text{ Equation (1) } \} \\ & g \circ \phi \circ F(f \vartriangle (g \circ f)) = \psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2) \circ F(f \vartriangle (g \circ f)) \\ \equiv & \quad \{ \text{ Equation (2) } \} \\ & \psi \circ F((g \vartriangle id) \times id) \circ F(f \vartriangle (g \circ f)) = \psi \circ F((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2) \circ F(f \vartriangle (g \circ f)) \\ \equiv & \quad \{ \text{ Functor } F, \text{ simplification } \} \\ & \psi \circ F((g \circ f \vartriangle f) \vartriangle (g \circ f)) = \psi \circ F((g \circ f \vartriangle f) \vartriangle (g \circ f)) \\ \equiv & \quad \{ \text{ obvious } \} \\ & \text{True} \end{array}$$

□

The idea behind Theorem 9 is simple. In order to avoid unnecessary traversals of intermediate results by g, it is sufficient to find a condition for $g \circ f$ being a single recursion. This condition is expressed by Equation (2) reading that g should not recursively applied to the results produced by itself.

With the above theorem, we can improve *foo* given in Section 2. First, we rewrite the recursive definition into the form of Equation (1).

$$\begin{array}{rl} foo \circ in_{F_{L_A}} & = \phi \circ F_{L_A}(foo \vartriangle (sum \circ foo)) \\ & \quad \text{where } \phi = Nil \triangledown (\lambda(x, (p_i, p_s)).\, Cons(x + p_s, p_i)) \end{array}$$

Next, we check whether *sum* can be expressed in the form of Equation 2.

$$
\begin{array}{ll}
 & sum \circ \phi \\
= & \quad \{ \ \phi \ \} \\
 & sum \circ (Nil \triangledown (\lambda(x,(p_s,p_i)).\ Cons(x+p_s,p_i))) \\
= & \quad \{ \ \circ \text{ and } \triangledown \ \} \\
 & sum \circ Nil \triangledown sum \circ (\lambda(x,(p_s,p_i)).\ Cons(x+p_s,p_i)) \\
= & \quad \{ \ sum \ \} \\
 & 0 \triangledown (\lambda(x,(p_s,p_i)).(x+p_s+sum(p_i))) \\
= & \quad \{ \ \text{Define } \psi = \lambda(x,((p_i,p),p_s)).(x+p_s+p_i) \ \} \\
 & (0 \triangledown \psi) \circ F_{L_A}((sum \vartriangle id) \times id)
\end{array}
$$

Finally, according to Theorem 9, it soon follows that

$$foo = \pi_1 \circ ([\phi \vartriangle (\psi \circ F_{L_A}((\pi_2 \vartriangle \pi_1) \vartriangle \pi_2))])_{F_{L_A}}$$

which can be easily in-lined to the efficient program as given in Section 2.

5 Cheap Tupling Strategy

So far we have given the three simple calculational rules and demonstrated how they are used to improve recursions. In summary, the general strategy in the application of each calculational rule is a two-step calculation, namely,

- Rewriting the given recursion into a suitable form required by the theorem;
- Performing simple calculation according to the theorem.

Since these two steps can be *automatically* implemented at a rather low cost compared with the previous tupling methods [Chi93], it becomes direct and possible to embed the rule in a real compiler of functional languages.

Now let's turn to see how to give full play of these three rules to handle more complicated recursions. Rather than discussing in a formal way, we show our idea by the following example. Suppose that we want to improve function f defined by[5]

$$
\begin{array}{ll}
f\ (x_1 : x_2 : xs) & = x_1 + f\ (x_2 : xs) + f\ (xs) + g\ (x2 : xs) \\
g\ (x_1 : x_2 : x_3 : xs) & = x_1 + g\ (x_2 : x_3 : xs) + g\ (x_3 : xs) + g\ (xs) + f\ (x_2 : x_3 : xs)
\end{array}
$$

where f calls another recursive function g. Here we omit the equations for the base cases for both f and g, e.g., $f\ [\,] = \cdots$, $f\ [x] = \cdots$, $f\ [x_1, x_2] = \cdots$, etc. Obviously, there are many redundant recursive calls to f and multiple traversals of the input (by f and g), and hence this definition is inefficient.

First, we eliminate multiple traversals in the recursive definition by grouping the functions *traversing over the same data structure*. We usually start with this calculation as it could turn mutual recursive definitions into a single non-mutual one, making room for the later removal of redundant recursive calls by Corollary

[5] For notational convenience, we use infix operator : for list Cons operator, i.e., $x : xs$ denotes $Cons(x : xs)$, and use $[\,]$ to denote *Nil*.

8 and unnecessary traversals by Theorem 9. To this end, we define $h = f \vartriangle g$. By Corollary 4, we can have

$$\begin{array}{ll} f & = \pi_1 \circ h \\ h\,(x_1 : x_2 : x_3 : xs) & = x_1 + x_1 + d_1\,(h\,(x_2 : x_3 : xs)) + d_2\,(h\,(x_3 : xs)) + d_3\,(h\,xs) \end{array}$$

where d_1, d_2 and d_3 are functions defined by $d_1(x, y) = x + x + y + y$; $d_2(x, y) = x + y$; $d_3(x, y) = y$. Next, we turn to improve h in order to improve f. We can eliminate redundant recursive calls to h by Corollary 8, and obtain the result something like

$$h = \pi_1 \circ ([\phi])_{F_{L_A}}$$

where ϕ is another function. Finally, we use Theorem 9 to eliminate possible unnecessary traversals in the catamorphism $([\phi])_{F_{L_A}}$.

In summary, our cheap tupling is proceeded in the following way. Given a set of functions defined by recursions. For each function, we calculate it to an efficient recursive definition in the order of eliminating multiple traversals, removing redundant recursive calls and getting ride of unnecessary data traversals.

It is worth noting that our cheap tupling can be well coexisted with fusion under the transformation in calculational form. As a matter of fact, they assist each other to obtain better optimization:

- Fusion merges several recursive functions into one, which makes our cheap tupling algorithm easy to find how to tuple them.
- Tupling improves the recursion by constructing an efficient catamorphism (a special instance of hylomorphism) for it. It does not lose any possibility for fusion in case it is composed with another recursion (see Section 3)

We shall not give the detailed discussion to convince the reader. A relevant study can be found in [HIT96a] where tupling and fusion are used together to derive list homomorphisms (i.e., catamorphisms over append lists). Recently, Chin [Chi95] discussed this issues on the fold/unfold transformation basis. But it is quite complicated to choose carefully the order of tupling and fusion since they may hinder each other.

6 Related Work and Discussions

It has been argued that the use of generic control structures which capture patterns of recursions in a uniform way is of great significance in program transformation and optimization[MFP91, Fok92, SF93, TM95]. Our work is much related to these studies. In particular, our work was greatly inspired by the success of applying this approach to fusion transformation as studied in [SF93, GLJ93, TM95].

We made the first attempt to apply this calculational approach to the tupling transformation as well. Previous work, as intensively studied by Chin [Chi93], tries to tuple *arbitrary* functions by *fold/unfold* transformations. In spite of its generality, it has to keep track of all function calls and devise clever control to avoid infinite unfolding, resulting in high runtime cost which prevents it from being employed in a real compiler system.

We follows the experience of work of fusion in calculational form [TM95] where a simple calculational rule is used. We identify three patterns of inefficient recursions, and construct three calculational rules to improve them based on the structural knowledge in these patterns. Though being simple and less general than Chin's, our cheap tupling transformation, as demonstrated, can be applied to a wide class of functions. Moreover, it can be efficiently implemented in a practical compiler and can be naturally coexisted with fusion under the basis of transformation in calculational form. Chin [Chi95] discussed the intergration of fusion and tupling under the basis of fold/unfold transformation, whose algorithm is much more complicated than ours because of the complicated control of infinite unfoldings for when fusion and tupling are used in one system.

The idea in Theorem 3 for the tupling of mutual recursive definitions is not new at all. It basically the same as Takeichi's *generalization algorithm* [Tak87] and Fokkinga's *mutumorphisms* [Fok92]. Takeichi showed how to define a higher order function common to all functions mutually defined so that multiple traversals of the same data structures in the mutual recursive definition can be eliminated. Fokkinga proposed the idea of mutumorphism and develop the laws for the fusion of mutual recursions with other functions. We generalize their idea (Corollary 4), and investigate deeply how it can be used further for the elimination of redundant recursive calls and unnecessary traversals of data (Corollary 8 and Theorem 9).

Our idea (Theorem 7) for avoiding redundant recursive calls is much simpler than the existed techniques such as, memoization [Mic68, Hug85], tabulation [Bir80, Coh83] and tupling [Chi92]. Though being less general, it can be done by a simple calculation rather than by complicated program analysis or applied at run time. In fact, we are much influenced by Hughes' idea of *lazy memoization* [Hug85] in which it is only required to reuse the previously computed results of recursive calls applied to arguments *identical* (not equal in value) to previous ones – that is, arguments stored in the same place in memory. We impose more restriction so that and we can remove redundant recursive calls by calculation at compilation time rather than at execution time.

Our cheap tupling is "cheap" in the sense that it can be implemented very *cheaply* and hence more practical than previous studies. But this comes at price of less generality. Although we have demonstrated that combining of three rules can deal with more complicated cases (Section 5), there are still many recursions that our tupling cannot be applied. How to enlarge our application scope is one of our future work.

7 Conclusions

We propose the first cheap tupling in calculational form for obtaining efficient recursions without multiple traversals of the same data structures, redundant recursive calls, and unnecessary traversals of intermediate data structures. Our cheap tupling algorithm can be naturally combined with fusion so that efficient functional programs can be obtained by program calculation. It has shown its promising in many examples, big or small. We are currently implementing a calculational sys-

tem based on the idea of fusion and tupling in calculational form. We are going to give a performance evaluation of the whole functional programs in the Haskell's benchmark in order to see how much we could get generally.

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.

[Bir80] R. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, 1980.

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

[Chi93] W. Chin. Towards an automated tupling strategy. In *Proc. Conference on Partial Evaluation and Program Manipulation*, pages 119–132, Copenhagen, June 1993. ACM Press.

[Chi95] W. Chin. Fusion and tupling transformations: Synergies and conflits. In *Proc. Fuji International Workshop on Functional and Logic Programming*, pages 106–125, Susono, Japan, July 1995. World Scientific Publisher.

[Coh83] N.H. Cohen. Eliminating redundant recursive calls. *ACM Transaction on Programming Languages and Systems*, 5(3):265–299, July 1983.

[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.P. Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms via tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418, Cracow, September 1996. Springer-Verlag.

[HIT96b] 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.

[Hug85] J. Hughes. Lazy memo-functions. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (LNCS 201), pages 129–149, Nancy, France, September 1985. Springer-Verlag, Berlin.

[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.

[Mic68] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[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.

[Tak87] M. Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.

[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.