Making Recursions Manipulable by Constructing Medio-types

Zhenjiang Hu * Hideya Iwasaki † Masato Takeichi ‡

Summary.

A catamorphism, generic version of our familiar foldr on lists, is quite a simple recursive scheme associated with data type definitions. It plays a very important role in program calculation, since for it there exists a general transformation rule known as Promotion Theorem. However, its structure is so tight that many recursions cannot be specified by catamorphisms because of their irregular reference to compound data structures. In this paper, aiming at manipulating functions defined by more general recursive schemes, we propose a method to factorize these functions into a composition of a catamorphism and a type reformer based on the construction of a medio-type – suitable intermediate data type extracted from recursion. Consequently, our method extends the applicability of transformational techniques specially geared to catamorphisms. Some examples are also given to illustrate our idea.

Topic: Program Transformation

1 Introduction

Recursive definitions, as found in most functional programming languages, provide a powerful mechanism for specifying programs. By programming with a small, fixed set of recursive patterns derivable from *algebraic type* definitions, an orderly structure can be imposed upon functional programs. Such structures are exploited to facilitate the proof of program properties, and even to calculate program transformation [18, 19].

A *Catamorphism*, generic version of our familiar *foldr* on lists, is quite a simple recursive patterns associated with data type definitions. Catamorphisms are significant in program calculation [1, 3, 12, 14], since for them there exists a general transformation rule known as *Promotion Theorem* stating that the composition of a function with a catamorphism is another catamorphism under the so-called *promotable condition*. Sheard [19] has succeeded in describing a normalization

* Dept. of Information Engineering, Graduate School of Engineering, University of Tokyo. Email: hu@ipl.t.u-tokyo.ac.jp

† Educational Computer Centre, University of Tokyo. Email: iwasaki@rds.ecc.u-tokyo.ac.jp

‡ Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo. Email: takeichi@takeichi.t.u-tokyo.ac.jp

algorithms to automatically calculate improvements of programs expressed in a language using catamorphism as the exclusive control operator.

Though general and simple, catamorphisms are lacking in descriptive power. Some functions, even being naturally derivable from algebraic types, cannot be specified by catamorphisms, and hence transformation rules for these functions have been investigated one by one. For instance, Meertens[16] gave transformation rules for *paramorphisms*; Fegaras[8] showed how to improve recursions inducting simultaneously over multiple algebraic data types. A question arises naturally: is there a systematic way to manipulate these recursive patterns derivable from data structures?

To answer this question, it may be helpful to make it clear why some recursions cannot be specified by catamorphisms. From our experience, we found that one of the reasons is simply that some components of data structures are referenced in an irregular way which breaks the condition of catamorphisms. To explain it, consider a simple recursion:

$$\begin{array}{lll} peak\ [\,] & = 0\,; & \\ peak\ [x] & = x\,; & \\ peak\ (x_1 : x_2 : xs) & = x_1 & x_1 > x_2 \\ & = peak\ (x_2 : xs) & otherwise. \end{array}$$

The function *peak* takes a cons list, and returns its first peak element, e.g., $peak\ [1,3,4,2,5] = 4$. It, being quite simple, cannot be specified as a catamorphism over cons lists. In other words, a binary operator, say $\oplus$, does not exist such that $peak\ (x : xs) = x \oplus (peak\ xs)$ holds, because the recursion needs to look one element ahead to continue. It implies that the data structure *cons lists* does not include enough information for *peak* to be specified as a catamorphism over it. This hints us to construct a new data structure with the needed element as a component. To do so, we define a new type, namely $M_L\ a$, as follows.

$$M_L\ a\ =\ C_1 \mid C_2\ a \mid C_3\ a\ a\ (M_L\ a)$$

We use the function *trans* to describe the transformation from the original data structure to the new one, which hereafter is called *type reformer*. It is defined as follows.

$$\begin{array}{ll} trans & :: [a] \to M_L\ a \\ trans\ [\,] & = C_1; \\ trans\ [x] & = C_2\ x; \\ trans\ (x_1 : x_2 : xs) & = C_3\ x_1\ x_2\ (trans\ (x_2 : xs)). \end{array}$$

In the definition of *trans*, x_2 has been included as a component of the structure constructed by C_3. What we have done by this transformation is that *peak* becomes to be factorized into two parts: a catamorphism over the new type $M_L\ a$ and a type reformer[1], i.e.,

$$\begin{array}{l} peak = (\!|\, 0, id, g \,|\!)\ .trans \\ \quad \text{where}\ \ g\ x_1\ x_2\ r = x_1 \quad if\ x_1 > x_2 \\ \qquad\qquad\qquad\quad = r \quad otherwise \end{array}$$

[1] We do not intend to puzzle readers here by using the notation $(\!|\, - \,|\!)$ for catamorphisms which will be defined in Section 2, but to give a concrete explanation.

Generally, as long as a recursion has the form of

$$(\!|\, \phi_1, \cdots, \phi_n \,|\!)\,.\epsilon$$

where ϵ is a type reformer, it becomes manipulable in the sense that the composition of another function h and this form can be manipulated by the Promotion Theorem once geared for catamorphisms. That is, if h is promotable to $(\!|\, \phi_1, \cdots, \phi_n \,|\!)$, h is also promotable to this recursion. We call M_L *medio-type* since it serves as a medium for making recursions manipulable.

To summarize, rather than investigating transformation rules for various recursive patterns, we try to factorize them into a composition of a catamorphism and a type reformer so that transformation rules for catamorphisms become also applicable for these recursions.

This paper is organized as follows. We briefly introduce some basic concepts in Section 2. In section 3, we formulate type reformers as a natural transformation between map functors of two data structures. Section 4 proposes a general algorithm for extracting medio-types from recursions. Two applications are given in Section 5, and conclusions and future work are described in Section 6.

2 Types, Functors and Catamorphisms

We adopt *Bird-Meertens Formalism* (BMF) as our algebraic framework [1, 3, 13, 14], which relies on the algebraic properties on data structures to provide the basis of program transformation.

We denote the application of a function f to its argument a by $f\ a$, and denote the functional composition with an infix dot (.) as $(f\,.\,g)\ x = f\ (g\ x)$.

We sometimes use the symbols like $\oplus$ to denote infix binary operators. These operators can be turned into unary functions by *sectioning* or partial application as follows.

$$(a\oplus)\ b = a \oplus b = (\oplus b)\ a$$

A data type is constructed as the *least solution* of recursive type equations. For example, the type of cons lists with elements of type a is usually given by the following type equation

$$L\ a = [\,] \mid a : (L\ a).$$

To capture both data structure and control structure, type $\mathcal{T}$ is defined by a type functor $F_{\mathcal{T}} = F_1 + \cdots + F_n$ and a type constructor $\tau = \tau_1 + \cdots + \tau_n$ such that

$$\tau_i :: F_i\mathcal{T} \to \mathcal{T} \qquad (i = 1, \cdots, n).$$

Note that a *type functor*[11] is a function from types to types that has a corresponding action on functions which respects identity and composition properties.

The cons list type, for example, is defined by a type functor $F_L = F_1 + F_2$ and a type constructor $\tau = \tau_1 + \tau_2$ where

F_L for objects:	$F_1\ X = \mathbf{1}$,	$F_2\ X = a \times X$
F_L for functions:	$F_1\ f = id$,	$F_2\ f = id \times f$
τ :	$\tau_1 = [\,]$,	$\tau_2 =\ (:)$

in which **1** stands for some distinguished one-element set, id for the identity function, $\times$ for product and $+$ for sum. For notational convenience[18], we may define the above F_L as

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

where ! is used to define constant functor, i.e.,

$$\begin{aligned} !a\ X &= a \\ !a\ f &= id \end{aligned}$$

and I denotes identity functor, i.e.,

$$\begin{aligned} I\ X &= X \\ I\ f &= f. \end{aligned}$$

Besides, we follow de Moor [7] to use μF_L to denote τ and μF_i to τ_i, since the name for type constructor is not so important in program calculation[2].

Central to this paper is the notion of catamorphisms which form an important class of functions over a given data type. A catamorphism is, put simply, a homomorphism from an initial data type. A consequence of defining a type by the least solution of a type equation is the unique existence of the catamorphism. For example, for cons lists, given e and $\oplus$, there exists the unique catamorphism, say $cata$, satisfying the following equations.

$$\begin{aligned} cata\ [\,] &= e; \\ cata\ (x : xs) &= x \oplus (cata\ xs) \end{aligned}$$

In essence, this definition is a *relabeling*: it replaces every occurrence of "[]" with e and every occurrence of ":" with $\oplus$ in the cons list. Since e and $\oplus$ uniquely determine a catamorphism, we shall use special braces to denote this catamorphism as

$$cata = (\!|\, e, \oplus \,|\!)_{F_L}.$$

If the functor F is clear from the context, we sometimes omit the subscript F in $(\!|\, \phi_1, \cdots, \phi_n \,|\!)_F$. Moreover we even abbreviate $(\!|\, \phi_1, \cdots, \phi_n \,|\!)$ to $(\!|\, \phi \,|\!)$ when we don't want to care much about the number of ϕ_i's.

In the world of catamorphisms, the *Promotion Theorem*[14] tells us that a composition of a function with a catamorphism is again a catamorphism under a certain condition.

Theorem 1 (Promotion)
Assume that $(\!|\, \phi_1, \cdots, \phi_n \,|\!)$ is a catamorphism with respect to the type functor $F = F_1 + \cdots + F_n$. For a given h, if there exist $\psi_1, \cdots, \psi_n$ satisfying

$$h.\phi_i = \psi_i.F_i\ h \qquad (i = 1, \cdots, n)$$

then

$$h\ .(\!|\, \phi_1, \cdots, \phi_n \,|\!) \ = (\!|\, \psi_1, \cdots, \psi_n \,|\!)\,.$$

□

[2] Some authors use in_{F_L} instead of μF_L.

3 Type Reformer

Before explaining how to factorize recursions into a catamorphism composed with a type reformer, we need to make it clear what a type reformer is. As argued in the introduction, a type reformer is used to reshape a data structure without changing the components of the original data structure. For example, a list $[a_1, a_2]$ can be reshaped to be $C_3\ a_1\ a_2\ (C_2\ a_2)$ where C_2 and C_3 are two type constructors in the new data structure; but not to be $C_3\ (2 \times a_1)\ a_2\ (C_2\ a_2)$ because the component $2 \times a_1$ in the new structure does not exist before. Reshaping structure allows elements to be duplicated or deleted but not to be modified or added.

One technique to formalize such reshaping is by means of the *map* function. For simplicity, assume that all elements in the original data structure are of the same type of a. An operation ϵ on this data structure is said to be a reshape, when for any function $f :: a \to b$ the result obtained by first applying f to each element (i.e., map f) and then manipulating by ϵ is the same as that obtained by first manipulating by ϵ and then applying f to each element, i.e.,

$$\epsilon.map\ f = map\ f.\epsilon.$$

Categorically speaking, the above "map" can be formalized as a map functor and the above "reshape" can be formalized as a natural transformation between map functors. In the following, we shall focus on this study.

A type with elements of type a will be denoted as $T\ a$. For instance, $L\ a$ denotes a list with elements of type a. A general data type may have no type parameter or have more than one type parameters. We shall first study the type with a single type parameter. Now we intend to define the map functor with respect to this type. Fortunately, we know that any *parametrized* data type like $T\ a$ comes equipped with a map functor we shall define the map functor in a similar way to Malcolm's[15], except for defining it over type functor rather than from type equations.

A bifunctor[18] provides means to abstract functor and will be used later, so we introduce it here.

Definition 1 (Bifunctor)
A *bifunctor* $\dagger$ is a binary operation taking types into types and functions into functions such that if $f :: A \to B$ and $g :: C \to D$ then $f \dagger g :: A \dagger C \to B \dagger D$, and which preserves identities and compositions:

$$\begin{aligned} id \dagger id &= id \\ (f \dagger g)\ .\ (h \dagger j) &= (f\ .h) \dagger (g\ .j). \end{aligned}$$

□

The following theorem shows how to construct a map functor in terms of bifunctor.

Theorem 2 (Map functor)
Let $T\ a$ be a parametrized type, F_T be the type functor for type $T\ a$. If there exists a bifunctor $\dagger$ satisfying

$$F_T = (!a \dagger),$$

that is,

$$\begin{aligned} F_T X &= a \dagger X \\ F_T f &= id \dagger f \end{aligned}$$

where $\dagger$ contains no occurrence of a, there exists a *map functor* (denoted as T)[3]

$$\begin{array}{ll} \text{for objects:} & T\ a = \text{the type defined by } F_T \\ \text{for functions:} & Tf = (\![\, \mu F_T.(f \dagger id)]\!)_{F_T}. \end{array}$$

Proof: To prove this theorem, it requires to verify that T is a functor, i.e.,

$$\begin{aligned} T\ id &= id \\ T(f.g) &= Tf\ .\ Tg \end{aligned}$$

This is established, since

$$\begin{array}{ll} & T\ id \\ = & \quad \{ \text{ definition of } T \ \} \\ & (\![\, \mu F_T.(id \dagger id)]\!)_{F_T} \\ = & \quad \{ \text{ bifunctor } \dagger \ \} \\ & (\![\, \mu F_T.id \,]\!)_{F_T} \\ = & \quad \{ \text{ identity } \} \\ & (\![\, \mu F_T \,]\!)_{F_T} \\ = & \quad \{ \text{ obvious } \} \\ & id \end{array}$$

and

$$\begin{array}{ll} & T(f.g) = Tf.Tg \\ \equiv & \quad \{ \text{ definition of } T \ \} \\ & (\![\, \mu F_T.((f.g) \dagger id)]\!)_{F_T} \\ & \quad = Tf.(\![\, \mu F_T.(g \dagger id)]\!) \\ \Leftarrow & \quad \{ \text{ promotion } \} \\ & \mu F_T.((f.g) \dagger id).F_T(Tf) \\ & \quad = Tf.\mu F_T.(g \dagger id) \\ \equiv & \quad \{ \text{ definition of } T \ \} \\ & \mu F_T.((f.g) \dagger id).F_T(Tf) \\ & \quad = \mu F_T.(f \dagger id).F_T(Tf).(g \dagger id) \\ \equiv & \quad \{ \text{ definition of } F_T \ \} \\ & \mu F_T.((f.g) \dagger id).(id \dagger (Tf)) \\ & \quad = \mu F_T.(f \dagger id).(id \dagger (Tf)).(g \dagger id) \\ \equiv & \quad \{ \text{ bifunctor } \dagger \ \} \\ & \mu F_T.((f.g) \dagger (Tf)) \\ & \quad = \mu F_T.((f.g) \dagger (Tf)) \\ \equiv & \quad \{ \text{ obvious } \} \\ & True \end{array}$$

□

Example 1 (Map functor L)

[3] This is the reason why we name T defined by for the map functor of type $T\ a$.

Consider the map functor L which maps type a to type $L\ a$, and maps function to function as

$$L\ f\ [a_1, a_2, \cdots, a_n] = [f\ a_1, f\ a_2, \cdots, f\ a_n].$$

As given in Section 2, the type functor for type $L\ a$ is F_L. Now we extract functor $!a$ out of F_L by bifunctor $\dagger$ such that

$$F_L = (!a \dagger).$$

That is,

$$\begin{aligned} a \dagger X &= \mathbf{1} + a \times X \\ f \dagger g &= id + f \times g. \end{aligned}$$

According to Theorem 2, map functor L is

$$Lf = (\!|\, \mu F_L.(f \dagger id) \,|\!)_{F_L} = (\!|\, [\,], (:).(f \times id) \,|\!)_{F_L}.$$

□

Example 2 (Map functor for rose trees)
This is a little complicated example. Rose trees are tree structures with an arbitrary branching factor: a tree is either a leaf or a node with a list of subtrees. The type is defined as

$$R\ a\ =\ Leaf\ a \mid Node\ (L\ (R\ a))$$

with a corresponding functor F_R

$$F_R = !a + L$$

where L is the map functor for cons lists. By a similar method, we know that the map functor for the rose tree type is

$$R\ f = (\!|\, Leaf.f, Node \,|\!)_{F_R}.$$

□

Having got the map functor, we turn to define type reformer that describes a reshape from one type to another. Recall that *natural transformation* is a structural map between functors. "Structural map" makes sense here, since we have already seen that a functor is, or represents, a structure that objects might have. Therefore, natural transformation seems fit for defining the type reformer which reshapes data structures. Intuitively, let F, G be functors, a transformation from structure F to structure G is a family t of functions $t_a :: F\ a \to G\ a$, mapping set $F\ a$ to set $G\ a$ for each a. A transformation t is *natural* if each t_a does not affect the constituents of the structured elements in $F\ a$ but only reshapes the structure from F-structure to G-structure. In other words, *reshaping* the structure by means of a natural transformation t and *subjecting* the constituents to an arbitrary morphism commute with each other:

$$t_{a'}.Ff = Gf.t_a \quad \text{For all } f :: a \to a'$$

The formal definition of natural transformation[9] is as follows.

Definition 2 (Natural transformation)
Let $\mathcal{A}, \mathcal{B}$ be categories, and $F, G : \mathcal{A} \to \mathcal{B}$ be functors. A transformation in $\mathcal{B}$ from F to G is a family t of morphisms t_a:

$$t_a : Fa \to_{\mathcal{B}} Ga \quad \text{for each } a \text{ in } \mathcal{A}.$$

A transformation t in $\mathcal{B}$ from F to G is natural, denoted as $t : F \dot{\to} G$, if

$$t_b.Ff = Gf.t_a \quad \text{for each } \ f :: a \to_{\mathcal{A}} b.$$

□

Example 3 (Natural transformation *rev*)
Consider the function that yields the reversal of its argument, i.e.,

$$rev_a\ [x_0, \cdots, x_{n-1}] = [x_{n-1}, \cdots, x_0]$$

where $x_i :: a$ $(i = 0, 1, \cdots, n-1)$.

The function rev_a reshapes a L structure into another L structure without affecting the constituents of its arguments, and the family $rev = \{rev_a\}$ is a natural transformation typed

$$rev : L \dot{\to} L,$$

since for all $f :: a \to b$

$$rev_b.Lf = Lf.rev_a$$

as is easily verified.

□

Now we are ready to define type reformer.

Definition 3 (Type reformer ϵ)
Given two parametrized types: $S\,a$ with its type functor F_S and its map functor S, and $T\,a$ with its type functor F_T and its map functor T. A type reformer ϵ from $S\,a$ to $T\,a$ is defined as a natural transformation $\epsilon : S \dot{\to} T$.

□

Example 4 (Type reformer *trans*)
The *trans* defined in the introduction is a type reformer from $L\ a$ to $M_L\ a$, This can be verified by showing that $trans : L \dot{\to} M_L$ holds.

□

Our discussion can be extended naturally to deal with type reformers mapping from multiple data structures to a new data structure. Without loss of generality, we only study the case of *two* instead of more than two. The idea is simple. We generalize the definition of the map functor from the type with a single type parameter to that with two type parameters. The map functor M shall take types a and b to type $M\ a\ b$ and take functions $f :: a \to a'$ and $g :: b \to b'$ to $M\ f\ g :: M\ a\ b \to M\ a'\ b'$. By means of bifunctors $\dagger$ and $\ddagger$ which include neither a nor b, we extract two constant functors $!a$ and $!b$ out of type functor F_M as

$$F_M = (!a\dagger).(!b\ddagger),$$

that is,

$$\begin{aligned} F_M X &= a \dagger (b \ddagger X) \\ F_M f &= id \dagger (id \ddagger f). \end{aligned}$$

Then, the map functor M (for functions) is defined as

$$M(f,g) = (\!|\, \mu F_M.f \dagger (g \ddagger id) \,|\!).$$

A type reformer from $(S\ a, T\ b)$ to $M\ a\ b$ is ϵ, satisfying

$$\epsilon_{a'b'}.(Sf \times Tg) = M(f,g).\epsilon_{ab}$$

for any $f :: a \to a'$, $g :: b \to b'$.

4 Extracting Medio-types from Recursions

Types and recursions are much related and functors play an important role in relating both of them. Although we are used to discussing a recursion over a type, it is not necessary to go from types to recursions. As a matter of fact, it is also possible to go from recursions to types.

Medio-type is a suitable type extracted from a recursion in order that the recursion can be factorized into a composition of a catamorphism over it and a type reformer. By the use of medio-type, a recursion has two forms: original recursive form and the new compositional form due to the introduction of medio-types. These two forms can be transformed to each other but used at different situation. The compositional form is used only when the recursion is to be manipulated. To make this clear, suppose that we are given the specification of

$$spec \ = \ g.f$$

and we are asked to promote function g into f which is defined by a recursion. To do so, we may factorize f to make it manipulable as

$$f = (\!|\, \phi_1, \cdots, \phi_n \,|\!).\epsilon.$$

Thus, we calculate *spec* in the following style.

$$\begin{array}{ll} & spec \\ = & \quad \{\ \text{definition}\ \} \\ & g.f \\ = & \quad \{\ \text{factorize } f \text{ to be a compositional form}\ \} \\ & g.(\!|\, \phi_1, \cdots, \phi_n \,|\!).\epsilon \\ = & \quad \{\ \text{Promotion Theorem}\ \} \\ & (\!|\, \psi_1, \cdots, \psi_n \,|\!).\epsilon \\ = & \quad \{\ \text{change compositional form to recurisive form}\ \} \\ & h \end{array}$$

From this calculation, we can see that the medio-type only exists during calculation, but not in the final result h. In this sense, medio-types serve as medium for manipulating recursions. This is the reason why we give the name of medio-types.

The essential requirement for medio-types is the existence of catamorphisms over them, otherwise the factorization becomes meaningless. Therefore, we have to impose restrictions on recursions.

r_f	$::=$	$e_1;\ \cdots;\ e_k$	recursion
e	$::=$	$f\ p = t$	equation
p	$::=$	$C\ p_1\ \cdots\ p_k$	constructor application
	$\mid$	v	variable
t	$::=$	v	variable
	$\mid$	c	global function name or constant except f
	$\mid$	$f\ p$	recursive application
	$\mid$	$t\ t$	other application

Fig. 1 Grammar of the language

We are restricted to describing recursions in a language with the grammar shown in Figure 1. In this grammar, we implicitly consider f as the function that is recursively defined.

A function f is defined by a recursion r_f that is made up of a sequence of equations $e_{f_1},\ \cdots,\ e_{f_k}$ while each equation gives the definition of f over one pattern p. This style ensures that the recursion is defined over the data structure of its argument. A pattern p is constructed with variables and type constructors. Obviously, a nested pattern is permitted. In a term t, we separate recursive application from other applications in order to guarantee that the recursions have neither nested function calls nor manipulation on the parameter of f. They outlaw such as $f\ (f\ x)$ and $f\ (2 + x)$. Obviously, this language cannot specify nested recursion like Ackerman function. But it can describe a wide class of recursions including primitive ones.

Our main extraction theorem is as follows.

Theorem 3 (Extraction of medio-types)
Given a function $f :: T \to R$ that is defined by a recursion r_f, there exists a medio-type M such that f can be described as a composition of a catamorphism on M and a type reformer from T to M.

□

This theorem says that any recursion specified in our language is surely factorizable. To prove Theorem 3, we give a constructive algorithm $\mathcal{R}$ that accepts a recursion as an input and output a triple of a type functor for medio-type, a catamorphism on the medio-type, and a type reformer from the initial data type to the medio-type. That is,

$$(F_M, (\!|\, \phi \,|\!)_{F_M}, r_\epsilon) = \mathcal{R}[\![r_f]\!]$$

where F_M stands for the type functor for Medio-type M and r_ϵ for the recursion that defines type reformer ϵ which maps the old type to M. The algorithm is shown in Figure 2.

In this algorithm, the function τ takes a variable and returns its type; $!\tau(x)$ denotes a definition of constant functor, e.g., in case x has type a, $!\tau(x) = !a$.

1: $\mathcal{R}[\![r_f]\!] = (F_M, (\![\,\phi\,]\!)_{F_M}, r_\epsilon)$
2: where
3: $F_M = F_1 + \cdots + F_k$
4: $(\![\,\phi\,]\!)_{F_M} = (\![\,\phi_1, \cdots, \phi_k]\!)_{F_M}$
5: $r_\epsilon = e_{\epsilon_1};\ \cdots;\ e_{\epsilon_k}$
6: where $(F_i, \phi_i, e_{\epsilon_i}) = \mathcal{E}[\![e_{f_i}]\!]\ \ (i = 1, \cdots, k)$
7: $e_{f_1}; e_{f_2}; \cdots; e_{f_k} = r_f$
8: $\mathcal{E}[\![f\ p_i = t_i]\!] = (F_i, \phi_i, \epsilon\ p_i = t'_i)$
9: where
10: $F_i = !\tau(x_1) \times \cdots \times !\tau(x_m) \times I_1 \times \cdots \times I_n$, if $m + n > 0$
11: $= !\mathbf{1}$, otherwise
12: $t'_i = \mu F_i\ x_1\ \cdots\ x_m\ (a_1[f/\epsilon])\ \cdots\ (a_n[f/\epsilon])$
13: $\phi_i\ x_1\ \cdots\ x_m\ s_1\ \cdots\ s_n = t[a_i/s_i]$
14: where $\{x_1, \cdots, x_m\} = FV(t_i)$
15: $\{a_1, \cdots, a_n\} = FC(t_i)$
16: $FV(v) = \{v\}$
17: $FV(c) = \{\,\}$
18: $FV(f\ p) = \{\,\}$
19: $FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$
20: $FC(v) = \{\,\}$
21: $FC(c) = \{\,\}$
22: $FC(f\ p) = \{f\ p\}$
23: $FC(t_1\ t_2) = FC(t_1) \cup FC(t_2)$

Fig. 2 Algorithm for constructing medio-type

The notation $t[x/y]$ represents a substitution for the term t by replacing all free occurrences of the term x with the term y. In addition, $FV(t)$ gives a set of free variables in t except those only appearing in recursive applications (see line 18). $FC(t)$ gives a set of recursive applications in the term t, e.g.,

$$FV(x_1 : x_2 : xs) = \{x_1, x_2, xs\},$$
$$FV(g\ x\ (h\ x\ y)\ (f\ z)) = \{x, y\}$$
where g and h are global function names
$$FC(h\ (f\ x)\ (f\ y)\ (f\ y)) = \{f\ x,\ f\ y\}$$
where h is a global function name.

The point of our algorithm is the construction of medio-type with respect to a recursion. Once the medio-type is obtained, the recursion can be factorized by simple program calculation. Intuitively speaking, the medio-type M extracted by our algorithm from the recursion of

$$\begin{aligned} f\ p_1 &= t_1 \\ f\ p_2 &= t_2 \\ &\vdots \\ f\ p_k &= t_k \end{aligned}$$

have the form something like

$$\begin{array}{rcl} M & = & C_1\ x_{11} \cdots x_{1m_1}\ M_1\ \cdots\ M_{n_1} \\ & | & C_2\ x_{21} \cdots x_{2m_2}\ M_1\ \cdots\ M_{n_2} \\ & \vdots & \\ & | & C_k\ x_{k1} \cdots x_{km_k}\ M_1\ \cdots\ M_{n_k} \end{array}$$

where all M_i's are the same as M. Each branch

$$C_i\ x_{i1} \cdots x_{im_i}\ M_1\ \cdots\ M_{n_i}$$

is derived from the term t_i, where

$$\begin{array}{ll} \{x_{i1}, \cdots, x_{im_i}\} & = FV(t_i) \\ n_i & = \text{Number of elements in } FC(t_i) \end{array}$$

The property of the algorithm is characterized by the following lemma.

Lemma 4
Let f be defined over the type $T\ a$ by a recursion r_f, and $(F_M, (\!|\ \phi\ |\!)_{F_M}, r_\epsilon) = \mathcal{R}[\![r_f]\!]$. Then $f = (\!|\ \phi\ |\!)_{F_M}.\epsilon$ and $\epsilon : T \dot{\rightarrow} M$ hold.

Proof Sketch: We shall prove that 1) Type functor F_M defines an initial data type; 2) The ϵ is a type reformer from T to M; 3) f is equivalent to $(\!|\ \phi\ |\!)_{F_M}.\epsilon$.

The proof of 1) is soon followed (line 3,10) by the fact that a type defined by a polynomial type functor (i.e. a functor only constructed by constant functor, identity functor, product and sum) is an initial data type [9].

To prove 2), we need to show that $\epsilon.T\ h = M\ h.\epsilon$ for any function h. The proof can be proceeded based on the definition of map functor and the definition of ϵ (line 8).

To prove 3), it is sufficient to show that

$$f\ p_i = (\!|\ \phi\ |\!)_{F_M}\ (\epsilon\ p_i)$$

holds for any p_i. We prove it by induction on p_i. It is not difficult to verify the base case where p_i contains no nested patterns. For the inductive case, we calculate as follows.

$$\begin{array}{ll} & f\ p_i = (\!|\ \phi\ |\!)_{F_M}\ (\epsilon\ p_i) \\ \equiv & \quad \{\ \text{line 8}\ \} \\ & t_i = (\!|\ \phi\ |\!)_{F_M}\ t'_i \\ \equiv & \quad \{\ \text{line 12}\ \} \\ & t_i = (\!|\ \phi\ |\!)_{F_M}\ (\mu F_i\ x_1 \cdots x_m\ (a_1[f/\epsilon]) \cdots (a_m[f/\epsilon])) \\ \equiv & \quad \{\ \text{Assume } a_i = f\ p'_i, \text{ line 15}\ \} \\ & t_i = (\!|\ \phi\ |\!)_{F_M}\ (\mu F_i\ x_1 \cdots x_m\ (\epsilon\ p'_1) \cdots (\epsilon\ p'_n)) \\ \equiv & \quad \{\ \text{catamorphism}\ \} \\ & t_i = \phi_i\ (F_i(\!|\ \phi\ |\!)_{F_M}\ x_1 \cdots x_m\ (\epsilon\ p'_1) \cdots (\epsilon\ p'_n)) \\ \equiv & \quad \{\ \text{line 10}\ \} \\ & t_i = \phi_i\ x_1 \cdots x_m\ ((\!|\ \phi\ |\!)_{F_M}(\epsilon\ p'_1)) \cdots ((\!|\ \phi\ |\!)_{F_M}(\epsilon\ p'_n)) \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ induction } \} \\
& t_i = \phi_i\ x_1 \cdots x_m\ (f\ p'_1) \cdots (f\ p'_m) \\
\equiv & \{ \text{ Our assumption, line 13 } \} \\
& true
\end{array}
$$

□

Example 5 (*sum*)
Assume that N denotes the type of natural number defined by

$$N = 0 \mid S\ N.$$

Consider the following recursion r_{sum} used to define function $sum :: N \to N$:

$$
\begin{array}{ll}
sum\ 0 & = 0; \\
sum\ (S\ n) & = S\ n\ +\ sum\ n
\end{array}
$$

The problem with *sum*, being unable to be specified by a catamorphism, is that it "takes its argument and keeps it too". Such kind of recursive patterns are called *paramorphisms*[16] which cover all the primitive recursions. We shall demonstrate how to extract a medio-type from this recursion. Although $\mathcal{R}[\![r_{sum}]\!]$ is given in top-down, it may be better to explain it in bottom-up. For the first equation, since $FV(0) = FC(0) = \{\ \}$, it follows that

$$\mathcal{E}[\![sum\ 0 = 0]\!] = (F_1 = !\mathbf{1}, 0, \epsilon\ 0 = \mu F_1))$$

For the second equation, since $FV(S\ n\ +\ sum\ n) = \{n\}$ and $FC(S\ n\ +\ sum\ n) = \{sum\ n\}$, it follows that

$$
\begin{array}{l}
\mathcal{E}[\![sum\ (S\ n) = S\ n + sum\ n]\!] = (F_2, \phi_2, \epsilon\ (S\ n) = t'_2) \\
\quad \text{where} \\
\qquad F_2 = !\tau(n) \times I = !N \times I \\
\qquad \phi_2\ n\ s_1 = S\ n + s_1 \\
\qquad t'_2 = \mu F_2\ n\ (\epsilon\ n).
\end{array}
$$

So we obtain our result:

$$
\begin{array}{l}
\mathcal{R}[\![r]\!] = (!\mathbf{1} + !N \times I, ([\![\, 0, \lambda n.\lambda s_1.S\ n + s_1]\!]), r_\epsilon) \\
\quad \text{where } r_\epsilon \text{ is the following collection of equations:} \\
\qquad \epsilon\ 0 = \mu F_1 \\
\qquad \epsilon\ (S\ n) = \mu F_2\ n\ (\epsilon\ n)
\end{array}
$$

By naming μF_i with constructor name C_i, we can rewrite our derived medio-type, say M_N, clearly as

$$M_N\ =\ C_1 \mid C_2\ N\ M_N.$$

□

Our algorithm can be easily extended to be applicable for the case where a recursion is defined over multiple data structures. The only difference lies in the definition of equations and recursive applications, where each occurrence of $f\ p$ should be replaced by $f\ (p_1, \cdots, p_l)$. So, the modification for our algorithm is simply to change all occurrences $f\ p$ to $f\ (p_1, \cdots, p_l)$ and $\epsilon\ p$ to $\epsilon\ (p_1, \cdots, p_l)$.

Example 6 (Medio-type for *zip*)
Consider the function $zip :: (L\,a, L\,b) \to L\,(a, b)$ defined by

$$\begin{array}{ll} zip\ ([\,], ys) & = [\,]; \\ zip\ ((x : xs), [\,]) & = [\,]; \\ zip\ ((x : xs), (y : ys)) & = (x, y) : zip\ (xs, ys). \end{array}$$

The medio-type derived by our algorithm is as follows:

$$\begin{array}{rcl} M\ a\ b & = & C_1 \\ & | & C_2 \\ & | & C_3\ a\ b\ (M\ a\ b). \end{array}$$

Here we name type constructors as C_1, C_2 and C_3. By this medio-type, *zip* is factorized as follows.

$$\begin{array}{l} zip = (\!|\, [\,], [\,], \lambda x.\lambda y.\lambda s_1.(x, y) : s_1 \,|\!)\ .\epsilon \\ \quad \text{where } \epsilon\ ([\,], ys) = C_1 \\ \qquad\quad \epsilon\ ((x : xs), [\,]) = C_2 \\ \qquad\quad \epsilon\ ((x : xs), (y : ys)) = C_3\ x\ y\ (\epsilon\ (xs, ys)) \end{array}$$

□

There are some points to be noticed. First, our algorithm is fully mechanical and can be implemented as an automatic transformation system.

Secondly, our algorithm is quite different from a simple reexpression of a recursion as a catamorphism composed with another function. The latter may introduce trivial and meaningless case. For example, any recursion can be expressed as an identity function composed with itself, because we know that any identity function is itself a catamorphism. The characteristic of our factorization is that the derived type reformer only concerns reshaping of the old data structure while the derived catamorphism covers as much computation as possible.

Finally, since our algorithm does not concern the order of free variables and recursive application calls during the construction of medio-types in the algorithm (line 14,10), one may reorder them to get other medio-types. However, these medio-types are the "same" in essence. In this sense, this algorithm gives only one of these medio-types.

5 Some Applications

We have given a general study on making recursions manipulable by constructing medio-types. The following two simple examples may help to see how our algorithm is applied to practical calculation to improve programs.

5.1 Manipulating *length.zip*

We will improve

$$spec = length.zip,$$

where $length = (\!|\, 0, \lambda x \lambda p.S\ p \,|\!)_{F_N}$ and *zip* is defined in Section 4. The same example has also appeared in [8], where a special promotion transformation was given for recursions over multiple inductive structures. Here, we shall show how to

manipulate on *zip*. Using the result obtained in Section 4 where a medio type has been extracted from *zip*, we know that

$$zip = (\!|\, [\,], [\,], \lambda x.\lambda y.\lambda s_1.(x,y) : s_1 \,|\!)\ .\epsilon.$$

So

$$\begin{array}{ll} & length.zip \\ = & \quad \{ \text{ above } \} \\ & length.(\!|\, [\,], [\,], \lambda x.\lambda y.\lambda s_1.(x,y) : s_1 \,|\!)\ .\epsilon \\ = & \quad \{ \text{ promotion on catamorphism } \} \\ & (\!|\, 0, 0, \lambda x.\lambda y.\lambda p.S\ p \,|\!)\ .\epsilon \end{array}$$

Finally we eliminate the medio-type by replacing the type constructors C_1, C_2, C_3 in the definition of ϵ with corresponding operators $0, 0, \lambda x.\lambda y.\lambda p.S\ p$ which appears in the derived catamorphism, and obtain the following improved program after simple calculation.

$$\begin{array}{lll} spec\ ([\,], ys) & = & 0; \\ spec\ ((x:xs), [\,]) & = & 0; \\ spec\ ((x:xs), (y:ys)) & = & S\ (spec\ (xs, ys)) \end{array}$$

Note that our result is simpler than Fegaras'[8] whose recursion consists of four equations for all combinations of two patterns. We believe that it is unnecessary to express a recursion to such a degree that Fegaras did.

5.2 Longest common subsequences problem

We will derive an efficient program for *longest common subsequences* problem – computing the length of the longest common subsequences between two given sequences. The specification is as follows[4].

$$\begin{array}{l} lcs = max/.(length.fst) * .(==) \triangleleft .gen \\ \quad \text{where} \\ \qquad gen :: (L\ a, L\ b) \to Set(L\ a, L\ b) \\ \qquad gen\ (xs, ys) = \{(x,y) | x \leftarrow subs\ xs, y \leftarrow subs\ ys\} \\ \qquad subs\ [\,] = \{\{\}\} \\ \qquad subs\ (x:xs) = subs\ xs\ \cup\ (x:) * (subs\ xs) \end{array}$$

All the possible pairs of two subsequences of two given sequences (represented by cons lists) are generated, and only those pairs that are equal remain. Finally their lengths are calculated and the maximum is left as the result.

Promotion will act as a driving force in our calculation, so we want to make *gen* manipulable. In the practical development of program, it is not the case that medio-types can always be extracted directly. Instead, some preprocessing are needed. Since *gen* is not an explicit recursion, we calculate it by inducting upon *xs* and *ys* to obtain the following recursion.

$$\begin{array}{lll} gen\ ([\,], ys) & = & \{(\{\}, \{\})\}; \\ gen\ ((x:xs), [\,]) & = & \{(\{\}, \{\})\}; \\ gen\ ((x:xs), (y:ys)) & = & ((x:) \times (y:)) * (gen\ (xs, ys)) \\ & & \cup\ gen\ (xs, (y:ys)) \\ & & \cup\ gen\ ((x:xs), ys) \end{array}$$

[4] For the notation, refer to [3].

According to our algorithm, the medio-type M extracted from the above recursion is

$$\begin{aligned} M\ a\ b = & \ C_1 \\ | & \ C_2 \\ | & \ C_3\ a\ b\ (M\ a\ b)\ (M\ a\ b)\ (M\ a\ b), \end{aligned}$$

and gen is expressed as a composition of a catamorphism and a type reformer.

$$\begin{aligned} gen = & (\!|\ \{(\{\}, \{\})\}, \{(\{\}, \{\})\}, \alpha\ |\!)_{F_M}.\epsilon \\ & \text{where } \alpha\ x\ y\ s_1\ s_2\ s_3 = ((x :) \times (y :)) * s_1 \cup s_2 \cup s_3 \\ & \qquad \epsilon\ ([\,], ys) \qquad\quad = C_1 \\ & \qquad \epsilon\ ((x : xs), [\,]) \quad = C_2 \\ & \qquad \epsilon\ ((x : xs), (y : ys)) = C_3\ x\ y\ (\epsilon(xs, ys)) \\ & \qquad\qquad\qquad\qquad\quad (\epsilon((x : xs), ys)) \\ & \qquad\qquad\qquad\qquad\quad (\epsilon(xs, (y : ys)) \end{aligned}$$

After gen has been rewritten to this form, it becomes manipulable. Thus we can promote $(==)\triangleleft$, $(length.fst)*$ and max into gen step by step according to the Promotion Theorem. We omit the details of the derivation, and only give the final result here.

$$\begin{aligned} lcs = & (\!|\ 0, 0, g\ |\!)_{F_M}.\epsilon \\ & \text{where } g\ x\ y\ l_1\ l_2\ l_3 = 1 + l_1, \qquad x = y \\ & \qquad\qquad\qquad\quad = max\ l_2\ l_3, \text{ otherwise} \end{aligned}$$

Finally, by eliminating medio-type, we obtain the following program.

$$\begin{aligned} & lcs\ ([\,], ys) = 0; \\ & lcs\ ((x : xs), [\,]) = 0; \\ & lcs\ ((x : xs), (y : ys)) \\ & \quad = 1 + lcs\ (xs, ys), \quad if\ x = y \\ & \quad = max\ (lcs\ (x : xs, ys))\ (lcs\ (xs, y : ys)), \quad \text{otherwise} \end{aligned}$$

The execution of these two programs under pfc[5][23] system has proved that the derived program is much more efficient than the original. With memoisation mechanism, the derived program is quadric regardless of expense for looking up memoising table, while the original program is exponential. For instance, in the case where two input sequences are $[1, 3, 5, 7, 9]$ and $[1, 4, 7, 10, 13]$, the derived program takes 2776 reductions while the original one takes 66450 reductions.

6 Conclusions and Future Work

In this paper, we propose a new concept called medio-types for the purpose of making recursions manipulable, and present an algorithm to factorize recursions into a composition of a catamorphism and a type reformer. In addition, we demonstrate that many recursions including paramorphisms and those over multiple data structures can be dealt by our method. Even though some of our theory in this paper is within the realm of sets and functions, it could be made much more general by category-theoretical means.

[5] The pfc is a functional language with memoisation mechanism developed by Takeichi Laboratory in University of Tokyo.

Besides the theoretical study, we also give some applications to show how to make use of our method in practical program calculation. One may find other applications. For example, our method may be useful to enhance the power of Sheard's normalization algorithm so that it can automatically calculate improvement of programs expressed in a language with more general recursive schemes rather than simple catamorphisms.

We put restrictions on our recursions. But this does not mean that we can deal with only such recursions. In fact, many transformation techniques can be used as a preprocess to transform others to the forms of ours. Some examples are shown below.

- *mutually recursive function*
  Suppose that $f_1, \cdots, f_n$ are mutually recursively defined, according to the well known tupling transformation, we may define $f\ x = (f_1\ x, \cdots, f_n\ x)$ and hence $f_i\ x = ith\ (f\ x)$ where ith stands for the function to pick out the ith element from a tuple. Having done so, we could remove this mutual recursion into a non-mutual one by replacing each f_i with $ith(f\ x)$.
- *recursive function with non-inductive parameter*
  So far as we have concerned, each recursive application must have the form of $f\ p_1\ \cdots\ p_m$, where $p_1, \cdots, p_m$ are patterns. It seems as if we ruled out those functions some of whose parameters are not patterns (i.e. containing some computations rather than just rearranging the structure). As a matter of fact, we can deal with them. Suppose in the recursive definition r_f, the recursive applications have the form of $f\ p_1\ \cdots\ p_m\ x_1\ \cdots\ x_n$ where x_1 to x_n are not simple patterns. We may define f as a higher order function over p_1 to p_m, so that our algorithm is also applicable. For example, the recursion of

$$\begin{aligned} f\ [\,]\ y &= y \\ f\ (x : xs)\ y &= f\ xs\ (x + y) \end{aligned}$$

  may be transformed to

$$\begin{aligned} f\ [\,] &= id \\ f\ (x : xs) &= \lambda y.((f\ xs)\ (x + y)) \end{aligned}$$

  which is in the scope of our recursions. It should be noted that for higher order function, our algorithm will generate a higher order catamorphism which is also manipulable[12, 17].

Careful readers, however, may have found that when we rearrange data structure to the medio-type structure by type reformer ϵ, some elements may have many occurrences. This arrangement gives much possibility for multiple computations and multiple traversals of data structures. Fortunately, implementation techniques of functional programs such as partial evaluation, lazy evaluation [21, 22], memoisation [2, 24], and tupling technique[2, 6, 5] can help us to avoid multiple computations and multiple traversals of data structures. At present, we releave us from it, but we plan to study how to remove these recomputations in medio-types.

It is of interest for us to find that our type reformer may be a kind of anamorphisms[18], generic version of our familiar unfold on cons lists. If this is true,

our algorithm would provide a mechanical way to rewrite a recursion into a hylomorphism (i.e., a composition of a catamorphism and an anamorphism) [18]. Moreover, it might support in theory that a compositional form can be transformed to a recursive form as argued in Section 4 based on the fact that a hylomorphism can be rewritten into a recursion. We would like to make it clear in our future research.

Another seemly interesting topic is to investigate the relation between the Promotion Theorem and deforestation techniques [4, 10, 20, 25]. They both provide means for fusing a composition of functions. By imposing structure on functions with the aid of type functors, the Promotion Theorem seems simpler and more concise than deforestation techniques. However, in deforestation there have been many good results that we have not got from the Promotion Theorem. For example, Chin [4] has proposed a terminative deforestation algorithm for first order and higher order functional languages. We believe that starting with the Promotion Theorem may give a more general and concise study on deforestation.

Acknowledgements

We gratefully acknowledge valuable discussions with Oege de Moor on helping us understand BMF. We also wish to express our gratitude to Akihiko Takano for introducing us his work on hylomorphisms and deforestation. Our special thanks are to Liangwei Xu, Fer-Jan de Vries and the members of Takeichi Research Group for a lot of suggestions on medio-types.

References

[1] R. Backhouse. An exploration of the Bird-Meertens formalism. In *STOP Summer School on Constructive Algorithmics, Abeland*, 9 1989.

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

[3] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

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

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

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

[7] O. de Moor and R.S. Bird. Solving optimization problems with catamorphisms. In *Mathematics of Program Construction (LNCS 669)*. Springer-Verlag, 1992.

[8] L. Fegaras, T. Sheard, and T.Zhou. Improving programs which recurse over multiple inductive structures. Technical Report OGI, Tech-report 94-005, Dept. of Computer Science and Engineering, Oregon Graduate Institution of Science and Technology, 1994.

[9] M. Fokkinga. A gentle introduction to category theory – the calculational approach –. Technical Report Lecture Notes, Dept. INF, University of Twente, Netherlands, September 1992.

[10] 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.

[11] T. Hagino. *Category Theoretic Approach to Data Types*. Ph.D thesis, University of Edinburgh, 1987.

[12] Z. Hu, H. Iwasaki, and M. Takeichi. Catamorphism-based transformation of functional programs. Technique report METR 94–06, Department of Mathematical Engineering and Information Physics, University of Tokyo, June 1994.

[13] J. Jeuring. *Theories for Algorithm Calculation*. Ph.D thesis, Faculty of Science, Utrecht University, 1993.

[14] G. Malcolm. Homomorphisms and promotability. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, pages 335–347. Springer-Verlag, 1989.

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

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

[17] E. Meijer. *Calculating Compilers*. PhD thesis, University of Nijmegen, Toernooiveld, Nijmegen, The Netherlands, 1992.

[18] 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, Massachusetts, August 1991.

[19] 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.

[20] M.H. Sorensen. A grammar-based data-flow analysis to stop deforestation. In *Colloquium on Algebra in Trees and Programming*, Edinburgh, Scotland, April 1994.

[21] A. Takano. Generalized partial computation for a lazy functional language. In *Proc. PEPM '91*, pages 1–12, New Haven, USA, June 1991. ACM Press.

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

[23] M. Takeichi. Evaluation partial order and synchronization mechanisms in parallel functional programs. In *40th IFIP WG2.1 Meeting Record 629*, pages 1–12, 1989.

[24] M. Takeichi. Partial evaluation by memoising functions. Unpublished note, 1994.

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