Catamorphism-Based Transformation of Functional Programs

Zhenjiang Hu * Hideya Iwasaki † Masato Takeichi ‡

Summary. Accumulations are operators on structured object that proceed their computation on each element of the object keeping some intermediate results. Accumulations are widely used in the design of efficient sequential and parallel programs. The purpose of this paper is to deal with the transformation on accumulations so that more efficient programs can be derived. We formulate accumulations by means of higher order catamorphisms and propose a promotion theorem for accumulations. Some examples are given to explain our method.

1 Introduction

Accumulations are operators on structured object that proceed their computation on each element of the object keeping some intermediate results.

Accumulations have gained a wide interest in the design of both sequential programs [1, 2] and parallel programs [4]. Especially in parallel programming, accumulations are considered as one of basic parallel operators [3], and a special hardware for *scan* accumulations has been installed in CM5 [5] recently.

The purpose of this paper is to deal with the transformation on accumulations so that more efficient programs can be derived. We have two problems here: one is how to formulate accumulations and the other is how to perform transformation on such accumulations.

For the first problem, it has been suggested that an accumulation can be efficiently described by the *parameter accumulation* whereby a specification is generalized by the inclusion of an extra argument [1, 9, 10]. Because such parameter accumulations can be of any style, it follows that the promotion strategy for parameter accumulations seems difficult to find. To overcome this shortcoming, we make restriction on the style of parameter accumulations, requiring that parameter accumulations be specified by function-valued *catamorphisms* (i.e. *higher order catamorphisms*). With this restriction, the promotion on accumulations can be well done while the descriptive power of higher order catamorphisms for accumulations is not decreased.

* Department of Mathematical Engineering and Information Physics, Faculty 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.

After settling the first problem, the second problem only needs to find a proper and powerful promotion strategy for the transformation of accumulations. Thanks to the uniform style of our accumulations, the promotion theorem can be naturally derived.

We adopt Bird-Meertens Formalism (BMF) as our algebraic framework. The BMF was firstly a calculus for the derivation of programs developed by Bird and Meertens[2, 12], and then extended to be a more general theory on structured data types based on category theory[11]. Besides its conciseness and higher degree of abstraction, it places a heavy emphasis on the algebraic properties of data types, resulting in a rich and powerful body of laws which show the close correspondence between data structures and control structures. In BMF, there are two important concepts: *catamorphism* and *promotion*. A catamorphism is, put simply, a unique homomorphism from the specified structured data type to another similar type, while the promotion theorem is a general transformation strategy for the manipulation of catamorphisms. If we could describe an accumulation by a catamorphism, manipulation on accumulations would be reduced to that on catamorphisms. Unfortunately, there are few study on the transformation of accumulations as well as higher order catamorphisms in BMF.

This paper is organized as follows. We introduce briefly some BMF notational conventions in Section 2. In section 3, we use some examples to show that higher order catamorphisms can describe accumulations effectively. Section 4 proposes a promotion theorem for our transformation. Two applications are given in Section 5, and finally some discussions are described in Section 6.

2 Basic Notational Conventions

The notation we use is based on that of Bird [2, 11]. We denote the application of function f to argument a with $f\ a$, and denote the functional composition with an infix dot (.) as $(f\,.\,g)\ x = f\ (g\ x)$.

We often use the symbols such as $\oplus$, $\otimes$, and etc. to denote infix binary operators. These operators can be turned into unary functions by *sectioning* or partial application:

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

A data type is constructed as the *least solution* of a recursive type equation and determined by a type functor F with some type constructors τ_i $(i = 1, \cdots, n)$. A *type functor* is a function from types to types that has a corresponding action on functions which respects identity and composition. For example, the type of the cons list with elements of type a is defined by

$$Cons\ a ::= [\,] \mid a : Cons\ a,$$

and according to this type equation, F is defined as

$$\begin{array}{ll} \text{For object:} & F\ X\ = \mathbf{1} + a \times X \\ \text{For function:} & F\ f\ \ = id + id \times f \end{array}$$

where $\mathbf{1}$ stands for some distinguished one-element set, id for the indentity function, $\times$ for product and $+$ for sum. In this case, two type constructors are $[\,]$ and $:$.

Central to this paper is the notion of catamorphisms which form important functions over a given data type. They are the functions that *promote through* the type constructors. The consequence of the definition of a type as the least solution of a type equation is the unique existence of the catamorphism. For example, for the cons list, given e and $\oplus$, there exists a unique catamorphism, say $cata$, satisfying the following equations.

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

In essence, this solution 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 are likely to use special braces to denote this catamorphism:

$$cata = (\!|\, e, \oplus \,|\!)$$

The catamorphisms are *manipulatable* in the sense that they obey a number of promotion or distributivity laws that are useful for transformation.

3 Specification with Higher Order Catamorphisms

The higher order catamorphism on a specified data type $\mathcal{T}$ is a catamorphism whose result of its application to a data of type $\mathcal{T}$ is still a function. Higher order catamorphisms are more powerful than first order ones in that many accumulations which cannot be described or cannot be efficiently described by first order catamorphisms can be efficiently described by higher order catamorphisms.

The procedure to specify accumulations by higher order catamorphisms is similar to the accumulational transformation [1]. The difference is that the final result we obtain is of the style of higher order catamorphisms. Suppose that the initial algorithm is specified naively by

$$acc :: \mathcal{T} \to C,$$

where $\mathcal{T}$ is determined by functor F with constructors $\tau_1, \cdots, \tau_n$. The accumulational specification using higher order catamorphisms is derived by the following three steps:

1. Define a function acc' by including a new parameter ac:

$$acc\ xs\ =\ acc'\ xs\ ac$$

2. Find all g_i's satisfying

$$acc'\,.\,\tau_i = g_i\,.\,(F\ acc') \quad (i = 1, \cdots, n)$$

To see clearly about the above equation, we give an example of cons lists. It turns out that

- $\mathcal{T} = Cons\ a$
- $\tau_1 = [\,]$, $\tau_2 = :$
- $F\ acc' = id\ +\ id \times acc'$

What we want to find are g $(= g_1)$ and $\oplus$ $(= g_2)$ that satisfy the following equations.

$$\begin{array}{ll} acc'\ [\,] & = g \\ acc'\ (x : xs) & = x\ \oplus\ xs \end{array}$$

3. Re-express acc as

$$acc\ xs\ =\ (\!|g_1, \cdots, g_n|\!)\ xs\ ac$$

The correctness of the above transformation can be easily proved by induction on the construction of the type $\mathcal{T}$ based on the theory of Malcolm [11] and Hagino [8].

The difficulty for such transformation is to find those g_i's that meet the requirement. In BMF, this can be done by calculation. Let us see some examples.

Example 3.1 Considering a function computing the initial sum of a list, we can define it naively as follows.

$$\begin{array}{ll} isum\ [\,] & = [\,] \\ isum\ (x : xs) & = x : ((x+) *\ (isum\ xs)) \end{array}$$

where $*$ stands for the "map" operation over list. It is not efficient because "$(x+)*$" costs much, even though it is a catamorphism. Rather than perform transformation from the initial inefficient definition, we are likely to rewrite it first into accumulational form and then perform other transformation. Let

$$isum\ xs\ =\ isum'\ xs\ 0$$

and we find $g_1 = \epsilon$ and $g_2 = \oplus$ where

$$\begin{array}{ll} \epsilon & = \lambda e.[\,] \\ x \oplus p & = \lambda e.((e + x) : p\ (e + x)) \end{array}$$

It is not difficult to check that

$$isum'\,.\,\tau_i = g_i\,.\,(F\ isum') \qquad (i = 1, 2)$$

holds, where $\tau_1 = [\,]$ and $\tau_2 = :$. To this end, the function $isum'$ can be specified by $(\!|\epsilon, \oplus|\!)$, a higher order catamorphism.

As the result, the new program becomes as follows.

$$\begin{array}{ll} isum'\ [\,] & = \lambda e.[\,] \\ isum'\ (x : xs) & = x \oplus (isum'\ xs) \\ & \quad where \\ & \quad x \oplus p = \lambda e.((e + x) : p(e + x)) \end{array}$$

□

Another example is somewhat different in that the initial specification can not be written directly without an accumulational parameter.

Example 3.2 Define a function $isub$ that computes the initial subtraction of a list, e.g.

$$\begin{array}{ll} & isub\ [5, 2, 1, 4] \\ = & [5, 5 - 2, 5 - 2 - 1, 5 - 2 - 1 - 4] \\ = & [5, 3, 2, -2] \end{array}$$

In fact, the function $isub$ can not be specified by a first order catamorphism on cons list because the subtraction is not commutative. In other words, there exists no such a binary operation $\oplus$ that makes

$$isub\ (x : xs)\ =\ x\ \oplus\ (isub\ xs)$$

hold. However, with higher order catamorphisms it can be effectively specified.

We show the result below, omitting the derivation procedure.

$$\begin{array}{ll} isub' & = (\!|\, \delta, \oplus \,|\!) \\ \text{where} & \delta = \lambda f.[\,] \\ & x \oplus p = \lambda f.((f\ x) : (p\ ((f\ x)-))) \end{array}$$

It is interesting to see that functions are used for accumulation.
□

So far, we have not talked about how to select a suitable extra parameter ac. It should be noted that a program written by a higher order catamorphism is not always efficient, because it depends much on how to accumulate. To define an efficient program by a higher order catamorphism, we need to find a suitable accumulation parameter. As seen from the above, accumulation parameters can be anything such as simple data or even functions depending on the problems.

If the initial f is confined to a certain kind, the extra accumulation parameter can be easily derived. This will be studied in Section 4.

4 Manipulating Accumulations

As we have seen that many accumulational algorithms can be specified by higher order catamorphisms, transformation on accumulations is reduced to that on catamorphisms.

In the world of first order catamorphisms, the Promotion Theorem [11] tells us that the composition of a homomorphism with a catamorphism is again a catamorphism.

Theorem 4.1 (First Order Promotion) Assume that $(\!|\, \phi_1, \cdots, \phi_n \,|\!)$ is a first order catamorphism with respect to the type functor F. For a given h, if there exist $\psi_1, \cdots, \psi_n$ satisfying

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

then

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

□

As to our higher order catamorphisms, we have the similar promotion theorem.

Theorem 4.2 (Higher Order Promotion) Let $(\!|\, \phi_1, \cdots, \phi_n \,|\!) :: L \to (A \to B)$ be a higher order catamorphism on type L with respect to the functor F. If there exists $\psi_1, \cdots, \psi_n$ satisfying

$$(h\,.)\,.\,\phi_i = \psi_i\,.\,F(h\,.) \quad (i = 1, \cdots, n)$$

then

$$h \,.\, ((\![\, \phi_1, \cdots, \phi_n]\!])\, as) = (\![\, \psi_1, \cdots, \psi_n]\!])\, as$$

□

Based on our promotion theorem together with other transformational rules, many efficient programs can be derived. To express the process of transformation, we use Feijen's proof format that provides a clear method of laying out a calculation. The calculation is displayed in the form of

$$\begin{array}{ll} & P \\ = & \{ \text{ hints as to why } P = Q \ \} \\ & Q \\ = & \{ \text{ hints as to why } Q = R \ \} \\ & R \end{array}$$

Let us see an example.

Example 4.1 Consider that we want to simplify the expression

$$ex\ xs = f * (isum'\ xs\ 0).$$

Since

$$\begin{array}{ll} & (f* \,.\, \epsilon)\ a \\ = & \{ \text{ def. of } \epsilon \ \} \\ & f * ((\lambda e.[\,])\ \ a) \\ = & \{ \text{ application } \} \\ & f * [\,] \\ = & \{ \text{ def. of map } \} \\ & [\,] \\ = & \{ \text{ abstraction } \} \\ & (\lambda e.[\,])\ a \\ = & \{ \text{ def. of } \epsilon \ \} \\ & \epsilon\ a \end{array}$$

and

$$\begin{array}{ll} & (f* \,.\, (x \oplus p))\ a \\ = & \{ \text{ def. of } \oplus \text{ in } isum' \ \} \\ & f * ((a + x) : p\ (a + x)) \\ = & \{ \text{ by the property of } * \ \} \\ & f\ (a + x) : f * (p\ (a + x)) \\ = & \{ \text{ let } (x \otimes p)\ a = f\ (a + x) : p\ (a + x) \ \} \\ & (x \otimes (f* \,.\, p))a \end{array}$$

hold, we change the above two equations to those in variable free notation:

$$\begin{array}{l} (f* \,.) \,.\, !\epsilon = \epsilon \,.\, id = \epsilon \,.\, F(f* \,.) \\ (f* \,.) \,.\, \oplus = \otimes \,.\, (id \times (f* \,.)) = \otimes \,.\, F(f* \,.) \end{array}$$

Based on the Promotion Theorem, we can simplify the original expression.

$$
\begin{array}{ll}
 & f * (isum' \ xs \ 0) \\
= & \{ \text{ composition and application } \} \\
 & (f* \, . \, (isum' \ xs)) \ 0 \\
= & \{ \text{ the derived result and Promotion Theorem } \} \\
 & ((f* \, .) \, . \, (\![\epsilon, \otimes]\!]) \ xs \ 0 \\
= & \{ \text{ promote } f* \text{ into catamorphism } \} \\
 & (\![\epsilon, \otimes]\!) \ xs \ 0
\end{array}
$$

The derived program is as follows.

$$
\begin{array}{l}
ex \ xs = ex' \ xs \ 0 \\
ex' \ [\,] = \lambda e.[\,] \\
ex' \ (x : xs) = x \otimes (ex' \ xs) \\
where \\
x \otimes p = \lambda e.(f \ (e + x) : p \ (e + x))
\end{array}
$$

□

The example shows that transformations on higher order catamorphisms are as direct as those on first order ones.

To derive an efficient program for a complicated initial specification, we proceed by the following two steps.

- Step 1: We optimize the inefficient simple functions in the specification to obtain a new specification;
- Step 2: We derive an efficient higher order catamorphism based on the promotional transformation and other strategies.

It should be noted that Step 2 of our derivation can be automatic but Step 1 must be done by ourselves. Fortunately, Step 1 is not so difficult, because the functions we deal with are simple. For some cases, Step 1 can be made automatic as shown in Theorem 4.3.

Theorem 4.3 (Generalization) Given a catamorphism $f = (\![\phi_1, \cdots, \phi_n]\!)$, if for all i, $\phi_i = \psi_i.F_i((\otimes))$ where $\otimes$ is associative and has its unit element e, then

$$(\![\phi_1, \cdots, \phi_n]\!) \ xs = (\![(\otimes).\psi_1, \cdots, (\otimes).\psi_n]\!) \ xs \ e$$

Proof Scheme:

$$
\begin{array}{ll}
 & (\![\phi_1, \cdots, \phi_n]\!) \ xs \\
= & \{ \ e \text{ is the unit element of } \otimes \ \} \\
 & (\otimes)((\![\phi_1, \cdots, \phi_n]\!) \ xs) \ e \\
= & \{ \text{ composition and application } \} \\
 & ((\otimes).((\![\phi_1, \cdots, \phi_n]\!) \ xs)) \ e \\
= & \{ \text{ assumption and by corollary 4.2 } \} \\
 & (\![(\otimes).\psi_1, \cdots, (\otimes).\psi_n]\!)
\end{array}
$$

□

By way of illustration of the Generalization Theorem, we trace the derivation of an efficient higher order catamorphism for the initial $reverse = (\![\, \phi_1, \phi_2]\!)$ which reverses a cons list.

$$\begin{array}{ll} \phi_1 & = [\,] \\ \phi_2\ a\ p & = p +\!\!+ [a] \end{array}$$

Since $\phi_1 = \lambda().[\,].F_1(+\!\!+)$, $\phi_2 = (\lambda a.\lambda p.p.(a :)).F_2(+\!\!+)$ where $+\!\!+$ is associative and has unit element $[\,]$, we have

$$reverse\ xs = (\![\, (+\!\!+).[\,], (+\!\!+).\lambda a.\lambda p.p.(a :)]\!)\ \ xs\ [\,]$$

the right hand side of which can be simplified to

$$(\![\, id, \lambda a.\lambda p.p.(a :)]\!)\ \ xs\ [\,].$$

This is the same efficient program as that in [10].

5 Some Applications

5.1 Downwards tree accumulations

The initial motivation for us to associate higher order catamorphisms with accumulations is to find a method to formulate downwards tree accumulations which are both efficient and manipulatable. Gibbons[7] proposed this problem and claimed that some restrictions on the initial downwards tree accumulations are necessary. However, his complicated discussion leaded us to seek another simpler and more concise method. Using higher order catamorphisms to specify downwards tree accumulations, we not only solve the problem, but also make Gibbons' restrictions unnecessary.

Downwards tree accumulations, depending on three operations f, $\oplus$ and $\otimes$, are defined on the tree type of

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

as:

$$\begin{array}{ll} da_{(f,\oplus,\otimes)}\ (Leaf\ a) & = f\ a \\ da_{(f,\oplus,\otimes)}\ (Node\ a\ x\ y) & = Node\ (f\ a)\ (da_{(((f\ a)\oplus),\oplus,\otimes)}\ x)\ (da_{(((f\ a)\otimes),\oplus,\otimes)}\ y). \end{array}$$

Many tree algorithms are described by them. For example, the application of $da_{(id,+,+)}$ to the tree of

$$Node\ 1\ (Leaf\ 2\ (Node\ 3\ (Leaf\ 4)\ (Leaf\ 5)))$$

will produce the tree of

$$Node\ 1\ (Leaf\ 3\ (Node\ 4\ (Leaf\ 8)\ (Leaf\ 9)))$$

as shown Figure 1.

It is clear that $da_{(f,\oplus,\otimes)}$ may be implemented in parallel by allocating one processor for each node in the tree. Unfortunately, such definition is not manipulatable, because it is not a catamorphism.

Gibbons claimed that only under some conditions could downwards tree accumulation be expressed as a catamorphism. In fact, the catamorphism he referred to

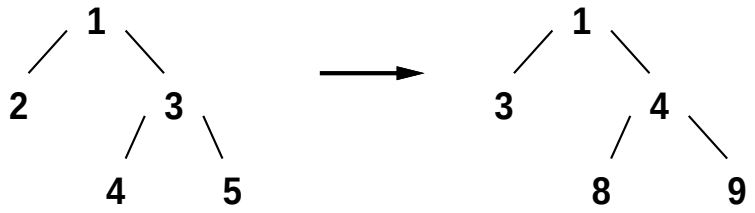

Fig. 1 Computation of $da_{(id,+,+)}$

is first order catamorphism. If we use higher order catamorphisms for description, many problems can be solved.

As usual, we rewrite downwards tree accumulations into a higher order catamorphism. Let

$$da_{(f,\oplus,\otimes)}\ tr = da'_{(f,\oplus,\otimes)}\ tr\ f$$

and we get

$$da'_{(f,\oplus,\otimes)} = (\![\ \lambda g.\, Leaf\ (g\ a)\ ,\ dnode]\!)$$

where

$$dnode\ a\ u\ v\ = \lambda g.\, Node\ (g\ a)\ (u\ ((g\ a)\oplus))\ (v\ ((g\ a)\otimes))$$

The new definition of the downwards tree accumulation may also be efficiently implemented both in sequential and parallel. Moreover, based on our promotion theorem, transformation on such accumulation can be performed, and as the result, many efficient parallel tree algorithms are derived.

5.2 Finding palindromic words

Consider that we want to derive an efficient program for the problem of finding all the words that are palindromes in a given character list.

5.2.1 The specification

The problem can be solved by three steps:

1. Define the function $fields$ to break up a line (represented as a list of character) into a list of words. It may be defined as follows.

$$\begin{array}{lll} fields\ [\,] & = [\,] & \\ fields\ (c:l) & = fields\ l, & if\ c == Space \\ & = word\ [c]\ l, & otherwise \\ word\ w\ (c:l) & = word\ (w +\!\!+ [c])\ l, & if\ c \neq Space \\ word\ w\ l & = w : (fields\ l), & otherwise \end{array}$$

2. Construct another word list in which all words obtained in step 1 are reversed.

$$reverse* \,.\, fields$$

3. Compare two word lists in step 1 and 2 and select the words that have the same spelling.

$$cs\ = \pi_1 * \,.\, eq_w \triangleleft \,.\, zip$$

where

$$\begin{array}{lll} eq_w\ [\,]\ [\,] & = True & \\ eq_w\ (x:xs)\ (y:ys) & = False, & if\ x \neq y \\ & = eq_w\ xs\ ys, & otherwise \end{array}$$

and

$$\begin{array}{lll} zip\ ([\,],[\,]) & = [\,] & \\ zip\ (x:xs\,,\ y:ys) & = (x,y) : zip\ (xs,ys) & \\ p \triangleleft\ [\,] & = [\,] & \\ p \triangleleft\ (x:xs) & = x : (p \triangleleft\ xs), & if\ p\ x \\ & = p \triangleleft\ xs, & otherwise \\ \pi_1\ (x,y) & = x & \end{array}$$

To summarize, the initial specification for the problem is

$$pw\ xs = (cs\,.\,pl)\ xs$$

where

$$pl\ xs = (fields\ xs\,,\ (reverse*\,.\,fields)\ xs).$$

It is a quadratic program, which is not so efficient.

5.2.2 The derivation

The derivation starts from the local optimization of $fields$, and then performs promotional transformation repetitly to find the optimized program for the whole specification.

For simplicity, we assume that each word in the given character list is followed by a single space.

Making $fields$ linear

As usual, by adding an extra accumulational parameter, we transform $fields$ into $fields'$ to be a higher order catamorphism. At the first glance, the accumulational parameter might be defined as a list holding parts of the scanning word and being concatenated to its end with the currently scanning character. But this is not enough because the concatenating operation $(+\!\!+)$ costs too much. Therefore, we use Hughes' idea [10], which represents the list $xs = [x_1, x_2, \cdots, x_{n-1}, x_n]$ by the following function composition.

$$f = (x_1:)\,.\,(x_2:)\,.\,\cdots\,.\,(x_{n-1}:)\,.\,(x_n:)$$

To get xs from f, we have only to apply the empty list to f. By this representation, concatenating a character to the end of a list can be performed in constant time.

The following is the result of rewriting $fields$ into an efficient higher order catamorphism. Let

$$fields\ xs\ =\ fields'\ xs\ id$$

and define $fields'$ in which w is for the use of the word accumulation.

$$\begin{array}{lll} fields' & = (\!|\,\alpha, \ominus\,|\!) & \\ \text{where} & & \\ \alpha\ w & = [\,] & \\ (x \ominus p)\ w & = p\ (w\,.\,(x:)), & if\ x \neq Space \\ & = (w[\,]) : (p\ id), & otherwise \end{array}$$

The transformed $fields$ is a linear program.

Promoting $reverse*$ into $fields$

Since we have got an efficient program for $fields$, we hope to derive an efficient program for $reverse*\,.\,fields$ by promoting $reverse*$ into $fields$. With the similar procedure as in Exercise 4.1, β and $\otimes$ can be derived satisfying

$$\begin{aligned}&(reverse*\,.)\,.\,\alpha = \beta\\&(reverse*\,.)\,.\,\ominus = \otimes\,.\,(id \times (reverse*\,.))\end{aligned}$$

where

$$\begin{aligned}\beta\ w &= [\,]\\(x \otimes p)\ w &= p\ ((x:)\,.\,w), \quad if\ x \neq Space\\&= (w[\,]) : (p\ id),\ otherwise.\end{aligned}$$

The details of this derivation are not addressed here, but it should be noted that during the derivation we have to use the following property.

$$reverse\ ((ws\,.\,w)\ [\,]) = (w\,.\,reverse\,.\,ws)\ [\,]$$

Based on the promotion theorem, we get

$$reverse*\,.\,(fields'\ xs) = (\![\,\lambda w.[\,]\,,\ \otimes]\!)\ \ xs$$

So the whole transformation becomes:

$$\begin{aligned}&(reverse*\,.\,fields)\ xs\\=\ &\{\ \text{new definition of fields}\ \}\\&reverse*\,(fields'\ xs\ id)\\=\ &\{\ \text{functional composition}\ \}\\&(reverse*\,.\,(fields'\ xs))\ id\\=\ &\{\ \text{result above}\ \}\\&(\![\,\lambda w.[\,]\,,\ \otimes]\!)\ \ xs\ id\end{aligned}$$

The derived program is an efficient linear program.

Making pl as a catamorphism

According to the above transformation, the definition of pl becomes as follows.

$$pl\ xs = ((\![\,\alpha, \ominus]\!)\ xs\ id\,,\ (\![\,\beta, \otimes]\!)\ xs\ id)$$

It will be shown that pl can be transformed into a higher order catamorphism. This transformation will make it easier for the next step of transformation.

For notational convenience, we define

$$(f\ ||\ g)\ (x, y) = (f\ \ x\,,\ g\ \ y)$$

and $unzip$ which is an inversion of zip, e.g.

$$unzip\ [(1, a), (2, b), (3, c)] = ([1, 2, 3], [a, b, c])$$

Now expressing pl by $||$, we have

$$pl\ xs = pl'\ xs\ (id, id)$$

where

$$pl'\ xs = (([\![\, \alpha, \ominus]\!])\ xs\ ||\ ([\![\, \beta, \otimes]\!])\ xs).$$

The derivation of a higher order catamorphism for pl', say $([\![\, \gamma, \odot]\!])$, is shown below.

First, we find γ.

$$\begin{array}{ll} & pl'\ [\,]\ (w_1, w_2) \\ = & \{\ \text{def. of } pl'\ \} \\ & (([\![\, \alpha, \otimes]\!])\,[\,]\ ||\ ([\![\, \beta, \otimes]\!])\,[\,])\ (w_1, w_2) \\ = & \{\ \text{def of catamorphism}\ \} \\ & (\alpha\,||\,\beta)\ (w_1, w_2) \\ = & \{\ \text{define } \gamma = (\alpha\,||\,\beta)\ \} \\ & \gamma\ (w_1, w_2) \end{array}$$

Next we find $\odot$.

$$\begin{array}{ll} & pl'\ (x : xs)\ (w_1, w_2) \\ = & \{\ \text{def. of } pl'\ \} \\ & (([\![\, \alpha, \otimes]\!])\,(x : xs)\ ||\ ([\![\, \beta, \otimes]\!])\,(x : xs))\ (w_1, w_2) \\ = & \{\ \text{def of catamorphism}\ \} \\ & ((x \ominus (([\![\, \alpha, \ominus]\!])\,xs))\ ||\ (x \otimes (([\![\, \beta, \otimes]\!])\,xs)))\ (w_1, w_2) \\ = & \{\ ||\ \text{and def. of } \ominus \text{ and } \otimes\ \} \\ & (([\![\, \alpha, \ominus]\!])\ xs\ (w_1.(x:))\,,\ ([\![\, \beta, \otimes]\!])\ xs\ ((x:).w_2)), \\ & \quad if x \neq Space \\ & (w_1[\,] : (([\![\, \alpha, \ominus]\!])\ xs\ id), (w_2[\,] : (([\![\, \beta, \otimes]\!])\ xs\ id))) \\ & \quad otherwise \\ = & \{\ ||\ \} \\ & (([\![\, \alpha, \ominus]\!])\ xs\ ||\ ([\![\, \beta, \otimes]\!])\ xs)\ (w_1.(x:), (x:).w_2), \\ & \quad if x \neq Space \\ & unzip\ ((w_1[\,], w_2[\,]) : \\ & (([\![\, \alpha, \ominus]\!])\ xs\ ||\ ([\![\, \beta, \otimes]\!])\ xs)\ (id, id))), \\ & \quad otherwise \\ = & (x \odot (pl'\ xs))\ (w_1, w_2) \\ & \text{where} \\ & (x \odot p)\ (w_1, w_2) \\ & \quad = p\ (w_1.(x:), (x:).w_2),\ if x \neq Space \\ & \quad = unzip\ ((w_1[\,], w_2[\,]) : p\,(id, id)),\ otherwise \end{array}$$

To this end, we have reached our result.

$$pl\ xs = ([\![\, \gamma, \odot]\!])\ \ xs\ \ (w_1, w_2)$$

Promoting *cs* into *pl*

The last step of our derivation is to promote *cs* into *pl* to get a tight program. We show only the final result, because the derivation procedure is similar to that

discussed already.

$$
\begin{array}{ll}
 & (cs\,.\,pl)\ xs \\
= & \{\ \text{def. of } cs \text{ and } pl\ \} \\
 & ((\pi_1\,.\,eq_w \triangleleft\,.\,zip).(pl'\ xs))\ (w_1, w_2) \\
= & \{\ \text{promote } \pi_1.eq_w \triangleleft .zip \text{ into } pl'\ \} \\
 & ([\![k, \oslash]\!])\ \ xs\ \ (w_1, w_2) \\
 & \text{where} \\
 & \quad k = \lambda w.[\,] \\
 & \quad (x \oslash p)\ (w_1, w_2) \\
 & \qquad = p\ (w_1.(x:), (x:).w_2),\ \ if x \neq Space \\
 & \qquad = w_1[\,] : p\ (id, id),\ \ if\ \ eq_w\ (w_1[\,])\ (w_2[\,]) \\
 & \qquad = p\ (id, id),\ \ otherwise
\end{array}
$$

The last derived result is linear. This ends our whole derivation.

6 Discussions

Accumulation strategies for program optimization is very powerful [1]. However, it seems suitable for small problems. To derive efficient programs for complicated problems, we formulate accumulations with higher order catamorphisms to make accumulations manipulatable. Through the derivations for some non-trivial problems, it turned out to be promising.

Our work concerning about higher order catamorphisms is related to Fokkinga and Meijer's specification of attribute grammar with catamorphisms [6]. However, their main interest is in specification, while we are interested in manipulation. In [13], Meijer proposed the following promotion theorem for higher order catamorphisms.

$$\frac{F(\circ g)a = F(f\circ)b \Rightarrow \phi a \circ g = f \circ \psi b}{f \circ ([\![\psi]\!]\ xs) = ([\![\phi]\!])\ xs \circ g}$$

Comparing with Corollary 4.2, it has too many free parameters which make derivation difficult. In fact, the above theorem can be obtained by performing promotion transformation for $(f\circ) \circ ([\![\psi]\!])$ and $(\circ g) \circ ([\![\phi]\!])$ according to Corollary 4.2.

It is also related to Sheard's work [14] where he discussed to some extent about transformation of higher order catamorphisms. However, his second-order promotion theorem is restricted in the sense that it only concerns the transformation from $(g\circ) \circ ([\![\phi]\!])$ to $(\circ g) \circ ([\![\psi]\!])$.

Hughes' representation of data structure with functions [10] suggested us to use functions as accumulation parameters in our higher order catamorphisms.

One of our future work is to apply our method to the derivation of correct parallel programs. It has been shown that accumulations are becoming more and more important in parallel programs. Blelloch [3] and many others argued that the accumulations could be regarded as a basic parallel operators and many useful parallel programs can be constructed by them. We hope that our study will be useful for the development of efficient parallel programs based on accumulations.

Another future work that seems interesting is to find how to derive efficient parallel programs by manipulating accumulations of parallel data structures (e.g.

trees, arrays and etc.). It is said that parallel data structures are of great important in parallel programming. It has been shown in the paper that the downwards tree accumulation becomes manipulatable by using higher order catamorphism without losing its efficiency in parallel implementation. Based on this result, we will undertake to derive efficient parallel programs for tree problems.

Acknowledgement

The authors would like to thank Dr. Oege de Moor for reading the draft and making a number of helpful remarks. We would also like to thank Dr. Xu for many enjoyable discussions.

References

[1] R. Bird, *The promotion and accumulation strategies in transformational programming*, ACM Transactions on Programming Languages and Systems **6** (1984), no. 4, 487–504.

[2] ———, *An introduction to the theory of list*, Logic of Programming and Calculi of Discrete Design (M. Broy, ed.), Springer-Verlag, pp. 5–42.

[3] G. Blelloch, *Scans as primitive parallel operations*, Proceedings of the International Conference on Parallel Processing, pp. 355–362.

[4] G.E. Blelloch, *Vector models for data-parallel computing*, MIT Press, 1990.

[5] C.E. Leiserson et al, *The network architecture of the connection machine cm-5*, Tech. Report Technique Report, Thinking Machine Corporation, 1992.

[6] M. Fokkinga, J. Jeuring, L. Meertens, and E. Meijer, *A translation from attribute grammers to catamorphisms*, Squiggolist (1990), 1–6.

[7] J. Gibbons, *Upwards and downwards accumulations on trees*, Mathematics of Program Construction (LNCS 669), Springer-Verlag, pp. 122–138.

[8] T. Hagino, *A typed lambda calculus with categorical type constructors*, Category Theory and Computer Science (D.H. Pitt, A. Poigné, and D.E. Rydeheard, eds.), Springer Lecture Notes in Computer Science, 283, pp. 140–157.

[9] P. Henderson, *Functional programming : Application and implementation*, Prentice Hall International, 1980.

[10] R. J. M. Hughes, *A noval representation of lists and its application to the function of reverse*, Information Processing Letters **22** (1986), no. 3, 141–144.

[11] G. Malcolm, *Homomorphism and promotability*, Mathematics of Program Construction (J.L.A. van de Snepscheut, ed.), Springer-Verlag, pp. 335–347.

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

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

[14] T. Sheard and L. Fegaras, *A fold for all seasons*, Proc. Conference on Functional Programming Languages and Computer Architecture (Copenhagen), pp. 233–242.