Calculating Accumulations

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

Summary. The *accumulation strategy* consists of generalizing a function by inclusion of an extra parameter, an *accumulating parameter*, for reusing and propagating intermediate results. It has gained a wide interest in the design of efficient programs. In this paper, we shall formulate accumulations as *higher order catamorphisms*, and propose several general transformation rules for calculating accumulations by *calculation-based* program transformation methods. Several examples are given for illustration.

1 Introduction

The *accumulation strategy* consists of generalizing a function over an algebraic data structure by inclusion of an extra parameter, an *accumulating parameter*, for reusing and propagating intermediate results. It is one of the standard optimization techniques taught to functional programmers[10].

We are faced with two difficulties in the accumulation strategy. One is to determine *where* and *when* to generalize the original function. By "where", we mean what part of the function should be generalized. We may have many alternatives such as generalizing a constant to a variable or generalizing an expression to a function[21]. By "when", we mean how many steps of *unfolding* are needed to find a suitable place for generalization, since a proper place usually comes out after several unfoldings. One general way, known as *forcing generalization*, is to do generalization in case *folding* cannot be done during unfold/fold transformations[3, 6, 21, 22], although related studies remain in an ad-hoc level.

The other difficulty, surprisingly not yet receiving its worthy consideration, is how to manipulate accumulations. We believe this more important for the following reason. As we know, one advantage of functional programming is that it allows greatly improved modularity. Functions can be defined in terms of smaller and simpler functions which are "glued" together to give their result. So, for a rather complicated function whose accumulation is difficult to obtain, we may decompose it into several simpler ones whose accumulations are easier to find, and then recompose the result.

[*] Department of Information Engineering, Graduate School of Engineering, The University of Tokyo (hu@ipl.t.u-tokyo.ac.jp) .

[†] Educational Computer Centre, The University of Tokyo (iwasaki@rds.ecc.u-tokyo.ac.jp) .

[‡] Department of Mathematical Engineering and Information Physics, Faculty of Engineering, The University of Tokyo (takeichi@u-tokyo.ac.jp) .

We are trying to overcome these difficulties and to give a systematic study on the accumulation strategy. To this end, we require that functions be defined in a structural way. This is so-called *structural programming* advocated by Meijer[18] and Sheard[23] who argued that programming with a small fixed set of recursive patterns derivable from type definitions can impose an orderly structure upon functional programs and such structures can be exploited to facilitate program calculation. One important recursive pattern in which we are interested is called *catamorphism*, generic version of our familiar *foldr* on lists. Catamorphisms are significant in program calculation [4, 11, 16, 18, 23], since there exists a general transformation rule known as *Promotion Theorem*.

In this paper, we shall formulate accumulations as *higher order catamorphisms*, and propose several general transformation rules for calculating accumulations (i.e., finding and manipulating accumulations) by *calculation-based* (rather than a search-based) program transformation methods. Several examples are given for illustration.

This paper is organized as follows. We briefly review some basic concepts in Section 2. In Section 3, we formulate accumulations as higher order catamorphisms and demonstrate through many examples that higher order catamorphisms can describe accumulations effectively. In Section 4, we propose our *Generalization Theorem* for deriving accumulations. Section 5 proposes two general *Accumulation Promotion Theorems* for manipulating accumulations. An example of the derivation of an efficient algorithm for longest path problem are given in Section 6, and some related works and conclusions are described in Section 7.

2 Preliminaries for Program Calculation

In this section, we briefly review the previous work in the program calculation[1, 4, 7, 15, 16, 18, 19] and explain some basic facts which provide theoretic basis of our method. In this paper, our default category C has as objects types, has as morphisms continuous functions, and has as composition general functional composition $\circ$.

We shall denote the application of a function f to its argument a by $f\ a$, and denote the functional composition by an infix circle ($\circ$) as $(f \circ g)\ x = f\ (g\ x)$. We abbreviate the equation

$$f \circ h = g \circ h$$

to

$$f = g \ \mathsf{mod}\ h.$$

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

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

2.1 Functors

Endofunctors on category C (functors from C to C) are used to capture the signatures of data types. In this paper, we assume that all data types are defined by endofunctors which are built up from I (identity functor), $!a$ (constant functor), $\times$ (product) and $+$ (separated sum). Such endofunctors are known as *polynomial functors*. We follow the definitions of product, separated sum functors, and related combinators as in [24].

Definition 1 (Product) The product $X \times Y$ of two types X and Y and its operation to functions are defined as:

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

□

Definition 2 (Separated Sum) The product $X + Y$ of two types X and Y and its operation to functions are defined as:

$$\begin{array}{ll} X + Y & = \{0\} \times X \cup \{1\} \times Y \\ (f + g)\ (0,x) & = (0, f\ x) \\ (f + g)\ (1,y) & = (1, g\ y) \end{array}$$

One combinator that is related to the separated sum functor is $\triangledown$ defined by

$$\begin{array}{ll} (f \triangledown g)\ (0,x) & = f\ x \\ (f \triangledown g)\ (1,y) & = g\ y. \end{array}$$

□

2.2 Categories of Functor Algebras

Let C be a category and F be an endofunctor on C.

Definition 3 (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 $\phi :: F\,X \to X$, called the operation of the algebra. □

Definition 4 (F-homomorphism) Given are 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 satisfying $h \circ \phi = \psi \circ F\,h$. □

Definition 5 (Category of F-algebras) The *category of F-algebras* has as objects the F-algebras and has as 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 provided F is a polynomial functors[16]. The representative we fix for the initial algebra is denoted by μF. Let $(T, in_F) = \mu F$, we call $in_F :: F\,T \to T$ the constructor of the initial algebra. Since the algebra (T, in_F) is initial in the category of F-algebras,

we have for every F-algebra (X, ϕ) there exists precisely a single $f :: T \to X$ such that

$$f \circ in_F = \phi \circ F\, f.$$

We denote the unique solution of f in the above equation by $(\![\phi]\!)_F$. The F-homomorphism $(\![\phi]\!)_F$ is called *F-catamorphism*. Initiality of (T, in_F) is fully captured by the law:

$$f = (\![\phi]\!)_F \quad \equiv \quad f \circ in_F = \phi \circ F\, f.$$

If the functor F is clear from the context, we omit the subscript F in $(\![\phi]\!)_F$ and in_F. Catamorphisms play an important role in program transformation (program calculation) in that they satisfy a number of nice calculational properties of which *promotion* is of greatest important:

Theorem 1 (Promotion)

$$f \circ \phi = \psi \circ F\ f \ \ \mathsf{mod}\ \ F(\![\phi]\!) \quad \Rightarrow \quad f \circ (\![\phi]\!) = (\![\psi]\!)$$

□

Promotion theorem gives the condition that has to be satisfied in order to "promote" a function into a catamorphism to obtain a new catamorphism.

2.3 Data Type Theory

Data type can be defined as an initial algebra. For example, the data type of cons lists with elements of type a, usually given by the equation

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

is defined as the initial algebra $(L\ a, [\,] \triangledown\ :)$ in the category of F_L-algebras, where F_L is the endofunctor defined by

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

in which $\mathbf{1}$ stands for some distinguished one-element set. As another example, the data type of binary trees, usually declared by

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

is the initial algebra $(Tree\ a,\ Leaf \triangledown Node)$ in the category of F_T-algebras, where F_T is the endofunctor defined by

$$F_T\ = !a + !a \times I \times I.$$

Central to this paper is the concept of catamorphisms, a homomorphism from an initial algebra to another algebra as defined in Section 2.2. Because a data type can be defined as an initial algebra, catamorphisms form an important class of functions over the given data type. For example, a catamorphism over type $L\ a$ can be generally represented as a function $(\![e \triangledown \otimes]\!) :: L\ a \to Y$, where Y is another type, e is a function with type $\mathbf{1} \to Y$, and $\otimes$ is a function with type $(a, Y) \to Y$. According to the definition of catamorphisms, we know that

$$(\![e \triangledown \otimes]\!) \circ ([\,] \triangledown\ :) = (e \triangledown \otimes) \circ F_L(\![e \triangledown \otimes]\!),$$

which can be in-lined to our familiar recursive equation:

$$\begin{array}{ll} ([e \triangledown \otimes])\ [\,] & = e \\ ([e \triangledown \otimes])\ (x : xs) & = x \ \otimes \ ([e \triangledown \otimes])\ xs \end{array}$$

In essence, $([e \triangledown \oplus])$ is *relabeling*: it replaces every occurrence of "[]" with e and every occurrence of ":" with $\otimes$ in the cons list. For example, $([0, +])$ is a function computing the sum of a cons list.

Note that we may have function $f :: \mathbf{1} \to X$ which is defined over $\mathbf{1}$. For example, $[\,]$ and e are such functions. In such case, we often regard f as a value with type X. In other words, we may use f to represent $\lambda().f$ when it is clear from the context.

3 Accumulations and Catamorphisms

An accumulation[3] is a kind of computation which proceeds over an algebraic data structure while keeping some information in an accumulating parameter to be used as an intermediate result. We shall borrow the word "accumulation" to refer to the function which performs accumulating computation.

As a simple example, consider the following definition of $isum$ which computes the initial prefix sums of a list, i.e., $isum\ [x_1, x_2, \cdots, x_n]\ 0 = [0, x_1, x_1{+}x_2, \cdots, x_1{+}x_2 + \cdots + x_n]$.

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

In this definition, the second parameter of $isum$ is the accumulating one that keeps partial sums for the later reuse, leading to an efficient linear algorithm.

Now by abstracting d in both equations, we obtain

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

which defines a catamorphism $([e \triangledown \otimes])$ over cons lists where

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

This is a *higher order catamorphism* in the sense that it takes a list to yield a function. Generally, we can formulate accumulations with the use of higher order catamorphisms as in the following proposition.

Proposition 2 (Accumulation) Let the data type T be the initial F-algebra, i.e., $(T, in) = \mu F$. An accumulation, which inducts over T using an accumulating parameter of type A and yields a value of type B, can be represented by a higher order catamorphism $([\phi]) :: T \to A \to B$ where $\phi :: F(A \to B) \to (A \to B)$. □

Some points on this proposition are worth noting. Firstly, in spite of the restrictions of higher order catamorphisms, such accumulations have no loss in descriptive power. Recalling our example of $isum$, we can naturally rewrite it to be a higher order catamorphism by means of abstraction.

Secondly, there is no restrictions on the type of the accumulating parameter, which may be either a function or a basic value. This helps to describe an accumulation concisely if a proper accumulation parameter is chosen. Consider the function $idif$ that computes the initial differences of a list, e.g. $idif\ [5,2,1,4] = [5, 5-2, 5-2-1, 5-2-1-4] = [5,3,2,-2]$. It can be defined efficiently as:

$$\begin{array}{lll} idif\ xs & = & idif'\ xs\ id \\ idif' & = & (\!|e \triangledown \otimes|\!) \\ \text{where} & & e = \lambda f.[\,] \\ & & x \otimes p = \lambda f.(f\ x : p\ ((f\ x)-)), \end{array}$$

Here, id denotes the identity function. In this definition, a function is used as the accumulating parameter. If we insisted on using non-function values, the algorithm would have become quite complicated because the subtraction is not associative.

Thirdly, our accumulations is valid for any freely constructed types such as lists, trees and so on. In fact, one of our motivations for associating higher order catamorphisms with accumulations was to find a method to make downwards tree accumulations manipulable and efficient. Gibbons[9] proposed the problem and tried to solve it with the aid of catamorphisms as we do here. Catamorphisms that he referred to are first order ones, and after a complicated discussion certain conditions are turned to be necessary for downwards tree accumulation to be expressed as a catamorphism. As will be seen in Example 1, we can give a concise definition using higher order catamorphism and make Gibbons' conditions unnecessary (see [11] for detail).

Example 1 (Downwards tree accumulation) We shall demonstrate how to describe an accumulation over trees. We have shown that the type of binary trees with elements of type a can be defined as the initial F_T-algebra in Section 2.3.

Downwards tree accumulation passes information downwards, from the root towards the leaves; each element is replaced by some functions on its ancestors. We denote a downwards accumulation by $(f, \oplus, \otimes)^{\Downarrow} :: Tree\ a \to Tree\ b$, which depends on three operations $f :: a \to b$, $(\oplus) :: b \to a \to b$ and $(\otimes) :: b \to a \to b$. This function is defined by

$$\begin{array}{l} (f, \oplus, \otimes)^{\Downarrow}\ (Leaf\ a) = Leaf\ (f\ a) \\ (f, \oplus, \otimes)^{\Downarrow}\ (Node\ (a, x, y)) \\ \quad = Node\ (f\ a, (((f\ a)\oplus), \oplus, \otimes)^{\Downarrow}\ x, (((f\ a)\otimes), \oplus, \otimes)^{\Downarrow}\ y). \end{array}$$

For instance, $(id, +, +)^{\Downarrow}$ is a function which replaces each node with the sum of all its ancestors.

Obviously, the function $(f, \oplus, \otimes)^{\Downarrow}$ cannot be specified by a first order catamorphism if these three operations do not satisfy certain conditions. And looking for such conditions may lead to a very complicated discussion as Gibbons did[9]. If we use a higher order catamorphism, we can describe it as follows.

$$\begin{array}{l} (f, \oplus, \otimes)^{\Downarrow}\ t = (\!|l \triangledown n|\!)_{F_T}\ t\ f \\ \quad \text{where} \\ \quad\quad l\ a\ \rho = Leaf(\rho\ a) \\ \quad\quad n\ (b, u, v)\ \rho = Node\ (\rho\ b, u((\rho\ b)\oplus), v((\rho\ b)\otimes)) \end{array}$$

We have demonstrated its use in calculating efficient parallel tree algorithms in [13]. □

4 Derivation of Accumulations

In this section, we shall propose our Generalization Theorem for deriving an accumulation from a structured specification in catamorphisms.

Recall that our accumulations are sort of higher order catamorphisms. If the specification is an *first order catamorphism*, namely a catamorphism whose result is a value instead of a function, the derivation of an accumulation can be considered as a transformation from a first order catamorphism to a higher order one. The following lemma is useful in such transformation.

Lemma 3 Let $([\phi])_F$ be a first order catamorphism. If there exists a binary operator $\oplus$ with a right identity e such that

$$(\oplus) \circ \phi = \psi \circ F(\oplus) \ \mathsf{mod}\ F\,([\phi])_F,$$

then

$$([\phi])_F\ xs = ([\psi])_F\ xs\ e.$$

Proof:

$$\begin{array}{ll} & ([\phi])_F\ xs \\ = & \quad \{\ e \text{ is a right identity of } \oplus\ \} \\ & (\oplus)\ (([\phi])_F\ xs)\ e \\ = & \quad \{\ \text{Function application and composition}\ \} \\ & ((\oplus) \circ ([\phi])_F)\ xs\ e \\ = & \quad \{\ \text{Promotion Theorem}\ \} \\ & ([\psi])_F\ xs\ e \end{array}$$

□

Lemma 3 tells us that the transformation from a first order catamorphism to a higher order one can be considered to find a suitable binary operator for relating the original catamorphism with the newly introduced accumulating part. However, from Lemma 3, we know only what property it should obey, not how it should be derived. To see how to derive such a binary operator $\oplus$ and thus obtain ψ, let us compare the both sides of the equation of

$$(\oplus) \circ \phi = \psi \circ F(\oplus) \ \mathsf{mod}\ F\,([\phi])_F,$$

and we can see that ϕ is expected to have the form of $g \circ F(\oplus)$ so that both sides can end with $F(\oplus)$. This suggests us to derive $\oplus$ from ϕ, and hence calculate ψ, which leads to our Generalization Theorem.

Theorem 4 (Generalization) Let $([\phi])_F$ be the given first order catamorphism. If $\phi = g \circ F(\oplus)$ where $\oplus$ is a binary operator with right identity e, then

$$([\phi])_F\ xs = ([\psi])_F\ xs\ e,$$

where $\psi = (\oplus) \circ g\ \ \mathsf{mod}\ \ F((\oplus) \circ ([\phi])_F)$.

Proof: According to Lemma 3, it is enough to prove that

$$(\oplus) \circ \phi = \psi \circ F\,(\oplus) \ \ \mathsf{mod}\ \ F([\phi])_F.$$

This can be easily verified with the composite property of functor F, i.e., $F\,f \circ F\,g = F\,(f \circ g)$. □

In fact, the Generalization Theorem provides us a *generalization procedure* for deriving accumulations. In practical program calculation the given catamorphism is likely to have the form of $([\phi_1 \triangledown \cdots \triangledown \phi_n])_{F_1+\cdots+F_n}$.

1. Rewrite each ϕ_i in a given catamorphism $([\phi_1 \triangledown \cdots \triangledown \phi_n])_{F_1+\cdots+F_n}$ to a form of $g_i \circ F_i(\oplus)$ such that $\oplus$ is a binary operator with a right identity e.
2. Calculate ψ_i according to the equation

$$\psi_i = (\oplus) \circ g_i \ \ \mathsf{mod}\ \ F_i((\oplus) \circ ([\phi_1 \triangledown \cdots \triangledown \phi_n])_{F_1+\cdots+F_n}).$$

3. Group ψ_i's to be $([\psi_1 \triangledown \cdots \triangledown \psi_n])_{F_1+\cdots+F_n}$. Obviously,

$$([\phi_1 \triangledown \cdots \triangledown \phi_n])_{F_1+\cdots+F_n}\ xs = ([\psi_1 \triangledown \cdots \triangledown \psi_n])_{F_1+\cdots+F_n}\ xs\ e.$$

For convenience, we define that a parameter x_j in $\phi_i(x_1, ..., x_m) :: T$ is said to be a *recursive parameter* if x_j has the type of T. In what follows, we would like to use p_j's instead of x_j's to explicitly denote recursive parameters. One property with recursive parameter is as follows.

Corollary 5 (Recursive parameter) Any recursive parameter p of function ψ_i obtained by the generalization procedure can be represented by $(p' \oplus)$ using another function p'.

Proof: A direct result from the equation:

$$\psi_i = (\oplus) \circ g_i \ \ \mathsf{mod}\ \ F_i((\oplus) \circ ([\phi_1 \triangledown \cdots \triangledown \phi_n])_{F_1+\cdots+F_n}).$$

□

One use of this corollary can be seen in Example 2 when we derive ψ_2.

Example 2 ($isum$) By way of illustration, consider the function $isum$ again. Suppose we are given the following first order catamorphism:

$$\begin{array}{l} isum = ([\phi_1 \triangledown \phi_2])_{F_1+F_2} \\ \quad \text{where} \quad \phi_1() = [0] \\ \qquad\quad\ \ \phi_2(x, p) = 0 : (x+) * p \end{array}$$

where $F_1 = !\mathbf{1}$ and $F_2 = !a \times I$. It is an inefficient quadratic algorithm, so we are trying to derive an efficient accumulation for it.

We begin by rewriting ϕ_1 and ϕ_2 to find g_1, g_2 and $\oplus$. It is trivial to see that

$$g_1 = \phi_1$$

from $\phi_1 = g_1 \circ F_1(\oplus)$ since $F_1\ f = id$. Now we hope to find g_2 and $\oplus$ satisfying the equation $\phi_2 = g_2 \circ F_2(\oplus)$ where $F_2\ f = id \times f$. This is the same to find g_2 and

$\oplus$ satisfying $\phi_2(x,p) = g_2(x,(p\ \oplus))$. Since in the definition of ϕ_2 the operation on p is $(x+)*$, we may define $\oplus$ as

$$p \oplus y = (y+) * p,$$

and we have

$$g_2(x,p') = 0 : p'\ x.$$

Note that $\oplus$ has the right identity 0 since $p \oplus 0 = (0+) * p = p$.

Next, we turn to calculate ψ_1 and ψ_2.

$$\begin{array}{ll} & \psi_1\ ()\ y \\ = & \quad \{\ \text{Generalization Theorem}\ \} \\ & ((\oplus) \circ g_1)\ ()\ y \\ = & \quad \{\ \text{Function composition}\ \} \\ & g_1() \oplus y \\ = & \quad \{\ \text{Def. of } g_1 \text{ and } \oplus\ \} \\ & (y+) * [0] \\ = & \quad \{\ \text{Map}\ \} \\ & [y] \end{array}$$

$$\begin{array}{ll} & \psi_2\ (x,(p\oplus))\ y \\ = & \quad \{\ \text{Generalization Theorem}\ \} \\ & ((\oplus) \circ g_2)\ (x,(p\oplus))\ y \\ = & \quad \{\ \text{Function composition}\ \} \\ & g_2(x,(p\oplus)) \oplus y \\ = & \quad \{\ \text{Def. of } g_2 \text{ and } \oplus\ \} \\ & (y+) * (0 : (p \oplus x)) \\ = & \quad \{\ \text{Map}\ \} \\ & y : ((y+) * (p \oplus x)) \\ = & \quad \{\ \text{Def. of } \oplus\ \} \\ & y : ((y+) * ((x+) * p)) \\ = & \quad \{\ \text{Map and associativity of ``+''}\ \} \\ & y : (((y+x)+) * p) \\ = & \quad \{\ \text{Def. of } \oplus\ \} \\ & y : (p \oplus (y+x)) \\ = & \quad \{\ \text{Sectioning}\ \} \\ & y : (p\oplus)(y+x) \end{array}$$

According to the second step of our procedure, it follows that

$$\begin{array}{ll} \psi_1\ () & = \lambda y.[y] \\ \psi_2\ (x,p') & = \lambda y.(y : p'\ (y+x)). \end{array}$$

Finally, according to the Generalization Theorem, we get

$$isum\ xs = ([\psi_1 \triangledown \psi_2])\ xs\ 0$$

which is the same as we introduced at the beginning of Section 3. $\square$

The generalization procedure requires us to derive a suitable binary operator $\oplus$. But sometimes, we cannot find a right identity for such $\oplus$. Techniquely, in this case, we may create a virtual one as in the following example.

Example 3 ($subs$) Consider the function $subs$ which accepts a cons list and yields the set of all its subsequences:

$$\begin{array}{lll} subs\ [\,] & = & \{[\,]\} \\ subs\ (x : xs) & = & subs\ xs \cup (x :) * subs\ xs, \end{array}$$

i.e.,

$$\begin{array}{l} subs = ([\phi_1 \triangledown \phi_2]) \\ \quad \text{where} \quad \phi_1() = \{[\,]\} \\ \qquad\quad\ \ \phi_2(x, p) = p \cup (x :) * p. \end{array}$$

With a similar derivation as for $isum$, we can define the binary operator for $subs$ as

$$p \oplus y = (y :) * p.$$

The problem with such $\oplus$ is that it has no right identity. Nevertheless, we could assume a virtual right identity ϵ and continue the calculation. By the generalization procedure we can have

$$\begin{array}{l} subs\ xs = ([\psi_1, \psi_2])\ xs\ \epsilon \\ \quad \text{where} \quad \psi_1\ ()\ y = \{[y]\} \\ \qquad\quad\ \ \psi_2\ (x, p)\ y = p\ y \cup (y :) * (p\ x). \end{array}$$

Section 6 will show how this definition is used in practical program derivation. □

The following is an often-quoted example illustrating the role of accumulation. We shall give the derivation of such accumulation using our approach.

Example 4 (rev) Consider the rev function which reverses a list. The initial quadratic specification is

$$\begin{array}{l} rev = ([\phi_1 \triangledown \phi_2]) \\ \quad \text{where} \quad \phi_1() = [\,] \\ \qquad\quad\ \ \phi_2\ (a, p) = p +\!\!+ [a] \end{array}$$

According to the generalization procedure, we see that the binary operator $\oplus$ can be instantiated to $+\!\!+$ whose right identity is $[\,]$. We omit other calculation but give the final result:

$$\begin{array}{l} rev\ xs = ([\phi'_1 \triangledown \phi'_2])\ xs\ [\,] \\ \quad \text{where} \quad \phi'_1\ ()\ = id \\ \qquad\quad\ \ \phi'_2\ (a, p) = p \circ (a :) \end{array}$$

which is the well-known efficient accumulation as that in [14]. □

Sheard[23] also studied the generalization of structure programs, but he requires the associativity of $\oplus$ and puts many restrictions on the structure of $\phi'_i s$. On the

contrary, we remove as many restrictions as possible in our theorem and give a much more general but practical generalization procedure. Our method covers Sheard's but not vice versa. For instance, our examples of the *isum* and the *subs* can not be dealt with by Sheard's.

5 Manipulating Accumulations

In this section, we propose several general rules for manipulating accumulations. By "manipulate accumulations" we mean the derivation of new accumulations from the old ones.

In compositional style of programming, it is usually the case where a function is composed with an accumulation. To find an accumulation for this composition, we propose the following theorem.

Theorem 6 (Accumulation Promotion 1) Let $([\phi])_F$ and $([\psi])_F$ be two accumulations. If

$$(h\circ) \circ \phi = \psi \circ F\ (h\circ)\ \ \mathsf{mod}\ \ F\ ([\phi])_F$$

then

$$h \circ (([\phi])_F\ xs) = ([\psi])_F\ xs.$$

Proof: This can be easily proved by specializing the Promotion Theorem in which h is replaced by $(h\circ)$. □

Example 5 Suppose that we want to derive an accumulation algorithm for the composition of $length \circ rev$, where $length$ is a function computing the length of a list. Recall that we have got the accumulation algorithm $rev\ xs = ([\phi_1' \triangledown \phi_2'])\ xs\ [\,]$ in Example 4, we can thus use Theorem 6 and calculate as follows.

$$\begin{array}{ll} & (length\circ) \circ \phi_1' \\ = & \quad \{\ \text{Def. of } \phi_1'\ \} \\ & (length\circ) \circ \lambda().id \\ = & \quad \{\ \text{Calculation}\ \} \\ & \lambda().length \\ = & \quad \{\ \ F_1\ f = id, \text{ define } \psi_1 = \lambda().length\ \} \\ & \psi_1 \circ F_1(length\circ) \end{array}$$

$$\begin{array}{ll} & (length\circ) \circ \phi_2' \\ = & \quad \{\ \text{Def. of } \phi_2'\ \} \\ & (length\circ) \circ (\lambda(a,p).p \circ (a:)) \\ = & \quad \{\ \text{Calculation}\ \} \\ & \lambda(a,p).length \circ p \circ (a:) \\ = & \quad \{\ F_2 f = id \times f, \text{ define } \psi_2 = \lambda(a,p).p \circ (a:)\ \} \\ & \psi_2 \circ F_2(length\circ) \end{array}$$

Therefore, by Theorem 6, we have

$$(length \circ rev)\ xs = ([\psi_1 \triangledown \psi_2])\ xs\ [\,].$$

□

Although we have successfully promoted $length$ into rev, our derived accumulation is quite unsatisfactory. The reason is that Theorem 6 does not tell anything about manipulation on accumulating parameter, which is also very important. Our following theorem is for this purpose.

Theorem 7 (Accumulation Promotion 2) Let $([\phi])_F$ and $([\psi])_F$ be two accumulations. If

$$(\circ g) \circ \phi = \psi \circ F\,(\circ g) \;\; \mathsf{mod} \;\; F\,([\phi])_F$$

then

$$(([\phi])_F \; xs) \circ g = ([\psi])_F \; xs$$

Proof: The proof is similar to that for Theorem 6 by specializing h to $(\circ g)$ in the Promotion Theorem. □

Example 6 Consider the point of the calculation in Example 5 where we have reached

$$(length \circ rev) \; xs = ([\psi_1 \triangledown \psi_2]) \; xs \; [\,].$$

We are going to derive η_1, η_2 and g based on Theorem 7 such that

$$(([\eta_1 \triangledown \eta_2]) \; xs) \circ g = ([\psi_1 \triangledown \psi_2]) \; xs.$$

Observing that

$$\begin{array}{rl} (\circ g) \circ \eta_1 = & \{ \text{ Theorem 7 } \} \\ & \psi_1 \circ F_1(\circ g) \\ = & \{ \; F_1 f = id, \text{ Def. of } \phi_1 \; \} \\ & \lambda().length \\ = & \lambda().(id \circ length) \\ = & (\circ length) \circ (\lambda().id), \end{array}$$

we may let $g = length$ and $\eta_1 = \lambda().id$. To derive η_2, we calculate as follows.

$$\begin{array}{rl} (\circ g) \circ \eta_2 = & \{ \text{ Theorem 7 } \} \\ & \psi_2 \circ F_2(\circ length) \\ = & \{ \text{ Def. of } \psi_2 \text{ and } F_2 \; \} \\ & (\lambda(a,p).p \circ (a :)) \circ (id \times (\circ length)) \\ = & \lambda(a,p).p \circ length \circ (a :) \\ = & \{ \; length \circ (x :) = (1+) \circ length \; \} \\ & \lambda(a,p).p \circ (1+) \circ length \\ = & (\circ length) \circ (\lambda(a,p).p \circ (1+)) \\ = & \{ \text{ Def. of } g \; \} \\ & (\circ g) \circ (\lambda(a,p).p \circ (1+)) \end{array}$$

It is immediate that $\eta_2 = \lambda(a,p).p \circ (1+)$. Therefore,

$$\begin{array}{rl} (length \circ rev) \; xs = & (((([\eta_1 \triangledown \eta_2]) \; xs) \circ length) \; [\,] \\ = & ([\eta_1 \triangledown \eta_2]) \; xs \; (length \; [\,]) \\ = & ([\eta_1 \triangledown \eta_2]) \; xs \; 0 \end{array}$$

By in-lining the definition of catamorphisms, we get our familiar recursive definition:

$$
\begin{array}{l}
(length \circ rev)\ xs = h\ xs\ 0 \\
\quad \text{where}\ \ h\ [\,]\ y\ =\ y \\
\qquad\qquad h\ (x : xs)\ y\ =\ h\ xs\ (1 + y).
\end{array}
$$

□

6 An Application

In this section, we explain how to apply our rules to a rather complicated example: calculating an efficient program for the *longest subsequences paths* problem. Bird[3] proposed an impressive study on this example. We review it in order to show that some of the Bird's explanation of where to generalize and how to proceed program transformation can be made more systematic by program calculation in a theorem-driven manner. In other words, we proceed the derivation by repeatedly trying to calculate program to a form that meets the conditions of our theorems so that our theorems become applicable.

Beginning with a simple and straightforward specification of the problem in hand, our calculational method coerces the specification into an executable and acceptably efficient program in some given functional language such as Gofer.

6.1 Specification

Our problem is to determine the length of the longest subsequence of a given sequence of vertices that forms a connected path in a given directed graph G. For simplicity we suppose that G is presented through a predicate arc so that $arc\ a\ b$ is true just in the case that (a, b) is an arc of G from vertex a to vertex b. For example, Figure 1 gives a graph and its representation. If the input sequence is $[C, A, B, D, A, C, D, E, B, E]$. The length of the longest path sequence is 5, particular length for solutions of $[C, D, A, B, E]$ and $[A, B, C, B, E]$. Our definition of the problem reads:

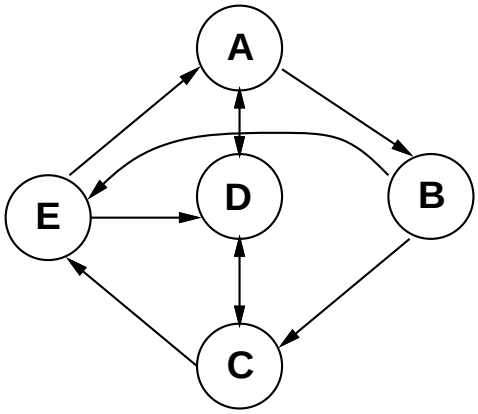
$$
\begin{array}{l}
psp\ = max \circ (length*) \circ (path\triangleleft) \circ subs \\
\quad \text{where}\ \ path\ [\,] = True \\
\qquad\qquad path\ [x] = True \\
\qquad\qquad path\ (x_1 : x_2 : xs) = arc\ x_1\ x_2\ \wedge\ path\ (x_2 : xs)
\end{array}
$$

Here $p\triangleleft$ is a function that takes a set and removes those elements not satisfying predicate p.

It is important to observe that the above does describe an algorithm to solve the problem, but it is not an efficient one. Clearly, the algorithm is exponential in the length of the given sequence.

6.2 Program Derivation

Our derivation of an accumulation for psp begins with finding an accumulation for $subs$, and then manipulates accumulations according to the Accumulation Promotion Theorems.



arc A B
arc A D
arc B C
arc B E
arc C D
arc C E
arc D A
arc D C
arc E A
arc E D

Fig. 1 An Example of a Graph and its Representation

Deriving an accumulation for *subs*

This derivation has been already given in Example 3. The result is as follows.

$$\begin{array}{rl} subs\ xs = & ([\psi_1 \triangledown \psi_2])_{F_1+F_2}\ xs\ \epsilon \\ \text{where} & \psi_1\ ()\ y = \{[y]\} \\ & \psi_2\ (x,p)\ y = p\ y \cup (y :) * (p\ x) \end{array}$$

where $F_1 = !\mathbf{1}$ and $F_2 = !a \times I$.

Manipulating $(path\triangleleft) \circ subs$

After obtaining the accumulation for *subs*, we step to derive an accumulation for the composition of the function $path\triangleleft$ with the *subs* based on Theorem 6.

Let $k = path\triangleleft$. We calculate ξ_1, ξ_2 from ψ_1, ψ_2 and k based on the condition in Theorem 6. Observing that

$$\begin{array}{rl} (k\circ) \circ \psi_1 = & (k\circ) \circ (\lambda().\lambda y.\{[y]\}) \\ = & \lambda().\lambda y.(k\ \{[y]\}) \\ = & \lambda().\lambda y.\{[y]\} \\ = & (\lambda().\lambda y.\{[y]\}) \circ F_1(k\circ) \end{array}$$

we thus get ξ_1:

$$\xi_1\ ()\ y\ =\ \{[y]\}.$$

Now calculate ξ_2

$$\begin{array}{rl} (k\circ) \circ \psi_2 = & (k\circ) \circ (\lambda(x,p).\lambda y.(p\ y \cup (y :) * (p\ x))) \\ = & \lambda(x,p).\lambda y.k\ (p\ y) \cup \underline{k((y :) * (p\ x))}). \end{array}$$

Before continuing the calculation, we break to look into ϵ (see Section 4), the right identity of $\oplus$. It should represent a vertex in the graph and satisfy $\epsilon : xs = xs$. Unfortunately, such vertex does not exist. To fix this problem, we consider ϵ as a virtual vertex with edges going to every vertex, i.e., $arc\ \epsilon\ v = True$ for each

vertex v in graph G. To keep the graph's path soundness, we do not allow any edge going from any vertex of G to ϵ, i.e., $arc\ v\ \epsilon = False$. Now return to our calculation for the underlined part. In the following, "$p \rightarrow q; r$" is used to denote "if p then q else r".

$$
\begin{array}{ll}
 & k((y:) * (p\ x)) \\
= & \quad \{ \text{ By Corollary 5 let } p = (p' \oplus), \text{ def. of } k \ \} \\
 & path \triangleleft\ ((y:) * ((x:) * p')) \\
= & \quad \{ \text{ Map } \} \\
 & path \triangleleft\ (((y:) \circ (x:)) * p') \\
= & \quad \{ \text{ Def. of } path \text{ and } \triangleleft \ \} \\
 & arc\ y\ x \rightarrow (y:) * (path \triangleleft\ ((x:) * p')); path \triangleleft\ ((x:) * p') \\
= & \quad \{ \text{ Def. of } k, \text{ and the above "let" nassumption } \} \\
 & arc\ y\ x \rightarrow (y:) * (k\ (p\ x)); k\ (p\ x) \\
= & \quad \{ \ \epsilon : xs = xs \ \} \\
 & arc\ y\ x \rightarrow (y \neq \epsilon \rightarrow (y:) * (k\ (p\ x)); k\ (p\ x)); k\ (p\ x) \\
= & \quad \{ \text{ Property of } \rightarrow \ \} \\
 & arc\ y\ x \wedge y \neq \epsilon \rightarrow (y:) * (k\ (p\ x)); k\ (p\ x)
\end{array}
$$

So

$$
\begin{array}{rl}
(k\circ) \circ \psi_2 = & \lambda(x,p).\lambda y.k\ (p\ y) \cup \\
 & (arc\ y\ x \wedge y \neq \epsilon \rightarrow (y:) * (k\ (p\ x)); k\ (p\ x)) \\
= & (\lambda(x,p').\lambda y.(p'\ y) \cup \\
 & (arc\ y\ x \wedge y \neq \epsilon \rightarrow (y:) * (p'\ x); (p'\ x)) \circ\ F_2(k\circ).
\end{array}
$$

Thus we get ξ_2 defined as

$$
\begin{array}{l}
\xi_2\ (x,p)\ y = (p\ y) \cup h \\
\quad \text{where}\ \ h = arc\ y\ x \wedge y \neq \epsilon \rightarrow (y:) * (p\ x); (p\ x).
\end{array}
$$

Finally, according to Theorem 6, we obtain

$$((path\triangleleft) \circ subs)\ xs = ([\xi_1 \triangledown \xi_2])\ xs\ \epsilon.$$

Promoting $max \circ (length*)$ into the obtained accumulation

We continue promoting $max \circ length*$ into the derived accumulation according to the Accumulation Promotion Theorems, as we did above. We omit the detail calculation but give the last result.

$$
\begin{array}{l}
psp\ xs = ([\eta_1 \triangledown \eta_2])\ xs\ \epsilon \\
\quad \text{where}\ \ \eta_1\ ()\ y = y = \epsilon \rightarrow 0; 1 \\
\qquad\qquad \eta_2\ (x,p)\ y = max\ (p\ y)\ h \\
\qquad\qquad\quad \text{where}\ \ \ h = arc\ y\ x \wedge y \neq \epsilon \rightarrow 1 + p\ x; p\ x
\end{array}
$$

Our program

By in-lining the last derived accumulation in Gofer, we get the following program. Note that we assign a positive number as the identifier of each vertex, and we assume that ϵ has the identifier of 0.

```
type Vertex = Int
psp :: [Vertex] -> Int
psp xs = acc xs 0
acc [] y = if y==0 then 0 else 1
acc (x:xs) y = max (acc xs y) h
    where h = if ((arc y x)&&(y/=0))
              then (1+acc xs x)
              else (acc xs x)
arc 0 v = True
arc v 0 = False
```

Comparing with the initial specification, we can see that substantial progress has been made. Careful readers may have found that if the above program is implemented naively, it still requires exponential time to give its answer. But this is not a problem. Different from the initial program, our derived program is suitable to be made optimized by some standard techniques such as tabulation [2] or memoisation[20]. Since these discussions are beyond the scope of this paper, we omit it here. Alternatively, a Gofer system with embedded memoisation mechanism [25] can give a direct efficient implementation.

As maybe easily verified, there are only $O(n^2)$ distinct values of acc requiring in the computation of $psp\ xs$, where $n = length(xs)$. So our final program brings the running time down to $O(n^2)$ if we ignore the time for manipulating memo-table.

7 Related Work and Conclusions

Much work has been devoted to the "forcing strategy" for the derivation of an accumulation. Pettorossi[22, 21] showed, through many examples, how to generalize functions in case folding fails. All these studies are search-based program transformation rather than our calculational based transformation.

The main contributions of our work are as follows.

- We formulate accumulations as higher order catamorphisms facilitating program calculation.
- We have provided several general rules for calculating accumulations, i.e., finding and manipulating accumulations. Each application of the rule can be seen as a canned application of unfold/simplify/fold in the traditional transformational programming[5]. Our calculational approach can avoid the process of keeping track of function calls and the clever control to avoid infinite unfolding which always introduces substantial cost and complexity in search-based transformation.
- Instead of ad-hoc study on accumulation strategy, our study is systematic and some of them can be embedded in an automatic transformation system. On the other hand, our theory is built upon category theory which is more general. For example, our definition of accumulations is applicable to any type besides lists.

- Our study of generic theorems for higher order catamorphisms which return functions (rather than values), we believe, is also a particular valuable avenue to pursue since programs that deal with *state* are quite common and can be handled algebraically.

Our work was greatly inspired by Bird's[3] pioneer work where he treated promotion as the guide strategy for achieving efficiency, and the parameter accumulation is the method by which promotion is effected. We improve Bird's work in three respects. First, we have extended his strategy from lists to any data type based on the categorical theory of data type. Secondly, we have shown that some of the Bird's explanation of where to generalize and how to proceed program transformation can be made more systematic by program calculation in a theorem-driven manner. Thirdly, we provides general rules for manipulating accumulations to obtain new ones.

Our work concerning higher order catamorphisms was much influenced by Fokkinga and Meijer's specification of attribute grammar with catamorphisms [8]. But their interest is in specification while we are interested in calculation. Another interesting work is from Meijer[17] who proposed the following promotion theorem for higher order catamorphisms for calculating compiler.

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

Comparing with our Accumulation Promotion Theorems, it has too many free parameters which make derivation difficult. In fact, the above theorem can be obtained from Theorem 6 and Theorem 7.

We are also related to Sheard's work [23] where he discussed to some extent about transformation of higher order catamorphisms. But his interest is in how to remove computations that are not amenable to his normalization algorithm. In addition, his promotion theorem for higher order catamorphisms are not so general as ours.

So far, we only studied the algorithms programmed in catamorphisms. How to handle the algorithms unable to be programmed in catamorphisms? In [12], we have shown that by using medio-type, more algorithms can be specified as a catamorphism with a type reformer. Thus our theorems in this paper are also applicable. We shall report some results about it in the future.

We are now interested in applying our approach to the calculation of memoisation functions[20], because a memo function is no more than a special accumulation whose accumulating parameter remembers all computed results of the application of the specified function.

Acknowledgment

The authors would like to thank Dr. Oege de Moor for reading an earlier draft and making a number of helpful remarks during his visit at Takeichi Laboratory of Tokyo University. We would also like to thank Mr. Xu for many enjoyable discussions.

References

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

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

[3] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.

[4] 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.

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

[6] M.S. Feather. A survey and classification of some program transformation techniques. In *TC2 IFIP Working Conference on Program Specification and Transformation*, pages 165–195, Bad Tolz (Germany), 1987. North Holland.

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

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

[9] J. Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction (LNCS 669)*, pages 122–138. Springer-Verlag, 1992.

[10] P. Henderson. *Functional Programming : Application and Implementation.* Prentice Hall International, 1980.

[11] Z. Hu, H. Iwasaki, and M. Takeichi. Catamorphism-based transformation of functional programs. Technical Report METR 94–06, Faculty of Engineering, University of Tokyo, June 1994.

[12] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving efficient functional programs by constructing medio-types. In *Proc. JSSST-FP '94*, pages 17–32, Kyoto, Japan, November 1994. Modern Science Publisher.

[13] Z. Hu, H. Iwasaki, and M. Takeichi. Promotion strategies for parallelizing tree algorithms. In *Proc. JSSST '94*, Osaka, Japan, November 1994.

[14] R. J. M. Hughes. A novel representation of lists and its application to the function reverse. *Information Processing Letters*, 22(3):141–144, March 1986.

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

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

[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, Massachuetts, August 1991.

[19] E. Meijer and G. Hutton. Bananas in space: Exteding fold and unfold to exponential types. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, California, June 1995.

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

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

[22] A. Pettorossi and A. Skowron. Higher-order generalization in program derivation. In *Conf. on Theory and Practice of Software Development*, pages 182–196, Pisa, Italy, 1987. Springer Verlag (LNCS 250).

[23] 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.

[24] 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.

[25] N. Yamashita. *Build-in Memoisation Mechanism for Functional Programs.* Master thesis, Dept. of Information Engineering, University of Tokyo, 1995.