Calculating Accumulations

Zhenjiang HU, Hideya IWASAKI, Masato TAKEICHI

Department of Information Engineering
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656 JAPAN

{hu,iwasaki,takeichi}@ipl.t.u-tokyo.ac.jp

Received June 1995. Revised December 1997. Accepted August 1998.

Keywords Program Calculation, Bird Meertens Formalisms, Promotion Transformation, Accumulation, Functional Programming.

§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[13]. However, there remain two major difficulties in this accumulation strategy. One is to determine *where* and *when* to generalize the original function. By "where," we mean what part in the definition of the function should be generalized; we may generalize a constant to a variable or an expression to a new function[24]. By "when," we mean how many steps of *unfolding* are needed to find a suitable place for generalization. One general way, known as *forcing generalization*, is to do generalization in case *folding* cannot be done during unfold/fold transformations[2, 6, 24, 25], 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 is very important, particularly in functional programming where bigger functions are often defined as a composition of smaller and simpler functions. So, for a rather complicated function

whose accumulation is difficult to obtain, we may decompose it into several simpler ones whose accumulations can be easier to find, and then fuse them to give the result.

To overcome these two difficulties, we employ the technique of *program calculation*[3, 20, 22, 7, 19] (i.e., calculation with programs) to systematically study the accumulation strategy. Program calculation is a kind of program transformation based on the theory of *Constructive Algorithmics*, a *program calculus* giving the theory for constructing *an algebra of programs*[4] in a systematic way. With the technique of program calculation, programs are first defined in terms of a small fixed set of recursive patterns, such as *catamorphisms*, *anamorphisms* and *hylomorphisms*, which are derivable from type definitions. This imposes certain structures upon programs facilitating program calculation. Then, calculation proceeds by repeatedly applying calculation laws (i.e., rules) or a calculation theorems that tell bigger calculation steps. It has been shown that the important program transformations of *deforestation* (or called *fusion*), *tupling transformation* and *parallelization* can be effectively formalized using this calculational technique[27, 15, 16, 17].

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. Some 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 several 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 is given in Section 6, and the related work and the conclusion are described in Section 7 and 8 respectively.

§2 Preliminaries

In this section, we shall briefly review the previous work[12, 22, 8] on Constructive Algorithmics, explaining the basic concepts and the notations that will be used in the rest of this paper. The default category is assumed to be

$\mathcal{CPO}$[22]. The notations used in this paper are much similar to that in some other papers[22, 27, 15, 16].

In the following, the application of a function f to its argument a is denoted by $f\ a$, and the function composition by an infix circle ($\circ$) as $(f \circ g)\ x = f\ (g\ x)$. Beside, the symbols like $\oplus$, $\otimes$, $\cdots$ are often used to denote infix binary operators. These operators can be turned into binary or unary functions by *sectioning*, namely $(\oplus)\ a\ b = (a\oplus)\ b = a \oplus b = (\oplus b)\ a$.

2.1 Functors

In Constructive Algorithmics, *polynomial endofunctors* are used to capture the structure of data types. They are built up by the following four basic functors.

- The **identity** functor I on type X and its operation on functions are defined as follows.

$$I\ X = X, \quad I\ f = f$$

- The **constant** functor $!A$ on type X and its operation on functions are defined as follows.

$$!A\ X = A, \quad !A\ f = id$$

  where id stands for the identity function.
- The **product** $X \times Y$ of two types X and Y and its operation to functions are defined as follows.

$$\begin{array}{lcl} X \times Y & = & \{\,(x, y) \mid x \in X,\ y \in Y\,\} \\ (f \times g)\ (x, y) & = & (f\ x,\ g\ y) \\ \pi_1\ (a, b) & = & a \\ \pi_2\ (a, b) & = & b \\ (f\ \triangle\ g)\ a & = & (f\ a,\ g\ a) \end{array}$$

- The **separated sum** $X + Y$ of two types X and Y and its operation to functions are defined as follows.

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

Although the product and the separated sum are defined on two parameters, they can be naturally extended for n parameters. For example, the separated sum over n parameters can be defined by $\Sigma_{i=1}^{n} X_i = \cup_{i=1}^{n}(\{\mathsf{i}\} \times X_i)$ and $(\Sigma_{i=1}^{n} f_i)\ (\mathsf{j}, x) = (\mathsf{j},\ f_j\ x)$ for $1 \leq \mathsf{j} \leq n$.

2.2 Data Types

Rather than being involved in theoretical study[12, 22, 8], we illustrate by some examples how data types can be captured by endofunctors. In fact, from a common data type definition, an endofunctor can be automatically derived to capture its structure[26]. As a concrete example, consider the data type of *cons lists* with elements of type A, which is usually defined by[*1]

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

In our framework, we use the following endofunctor to describe its recursive structure:

$$F_{L_A} = !\mathbf{1} + !A \times I$$

where $\mathbf{1}$ denotes the final object, corresponding to ()[*2]. Besides, we use $in_{F_{L_A}}$ to denote the *data constructor* in $\mathit{List}\ A$:

$$in_{F_{L_A}} = \mathit{Nil} \triangledown \mathit{Cons}.$$

In fact, the $\mathit{List}\ A$ is the least solution of X to the equation $X = in_{F_{L_A}}\ (F_{L_A}\ X)$ as discussed by Hagino[12]. The $in_{F_{L_A}}$ has its inverse, denoted by $out_{F_{L_A}}$: $\mathit{List}\ A \to F_{L_A}\ (\mathit{List}\ A)$, which captures the *data destructor* of $\mathit{List}\ A$, i.e.,

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

Another example is the data type of *binary trees* with leaves of type A usually defined by

$$\mathit{Tree}\ A = \mathit{Leaf}\ A \mid \mathit{Node}\ (A,\ \mathit{Tree}\ A,\ \mathit{Tree}\ A).$$

The corresponding functor F_{T_A} and data constructor $in_{F_{T_A}}$ are:

$$F_{T_A} = !A + !A \times I \times I, \quad in_{F_{T_A}} = \mathit{Leaf} \triangledown \mathit{Node}.$$

[*1] Note that for notational convenience, we sometimes use [] for *Nil* and infix operator : for *Cons*. Thus, for example, $x : xs$ stands for $\mathit{Cons}(x, xs)$ and $[a]$ for $\mathit{Cons}(a, \mathit{Nil})$.

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

2.3 Catamorphisms

Catamorphisms, one of the most important concepts in Constructive Algorithmics, form a class of important recursive functions over a given data type. They are the functions that *promote through* the type constructors. For example, for the type of cons lists, given e and $\oplus$, there exists a unique catamorphism, say *cata*, satisfying the following equations.

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

In essence, this solution is a *relabeling*: it replaces every occurrence of $[\,]$ with e and every occurrence of $:$ with $\oplus$ in the cons list. Because of the uniqueness property of catamorphisms (i.e., for this example e and $\oplus$ uniquely determines a catamorphism over cons lists), we are likely to use special braces to denote this catamorphism as $cata = (\!|e \triangledown \oplus|\!)_{F_{L_A}}$. In general, a catamorphism over a data type captured by functor F is characterized by:

$$h = (\!|\phi|\!)_F \quad \equiv \quad h \circ in_F = \phi \circ Fh.$$

It follows that our familiar *foldr* over lists can be described as $foldr(\oplus), e = (\!|e \triangledown \oplus|\!)_{F_{L_A}}$. As a matter of fact, with catamorphisms many functions can be defined. For example, the function sum, computing the sum of the elements of a cons list, can be defined by $(\!|0 \triangledown plus|\!)_{F_{L_{Int}}}$. When no ambiguity happens, we usually omit the subscript of F in $(\!|\phi|\!)_F$.

Catamorphisms play an important role in program transformation (program calculation), as they satisfy a number of nice calculational properties of which the *promotion theorem* is of greatest importance:

Theorem 2.1 (Promotion)

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

We abbreviate the equation $f \circ h = g \circ h$ to $f = g \ \mathsf{mod}\ h$.

Promotion theorem gives the condition that has to be satisfied in order to promote (fuse) a function into a catamorphism to obtain a new catamorphism. It actually provides a *constructive* but powerful mechanism to derive a "bigger" catamorphism from a program in a compositional style, a typical style for functional programming.

§3 Accumulations and Catamorphisms

An accumulation[2] 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 an example, consider the following definition of $isum$ which computes the initial prefix sums of a list, i.e., $isum\ [x_1, x_2, \cdots, x_n]\ 0 = [0, x_1, x_1 + x_2, \cdots, x_1 + x_2 + \cdots + x_n]$.

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

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

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

which defines a catamorphism $(\!|e\ \triangledown\ \otimes|\!)$ over cons lists where

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

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 3.1 (Accumulation)
An accumulation, which properly[*3] inducts over T (captured by functor F_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 $\phi : F_T(A \to B) \to (A \to B)$.

Some points on this proposition are worth noting. Firstly, in spite of the syntactic restriction 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 lambda abstraction.

[*3] A recursion is said to be *properly* inducts over a structure if the parameters of the recursive calls occurred in the definition body are substructures of the original one.

Secondly, there is no restriction on the type of the accumulating parameter, which may be either a function or a basic value. This map help to describe accumulations in a more concise way by choosing a suitable accumulation parameter. 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:

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

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 accumulation algorithm would have become quite complicated because the subtraction is not associative.

Thirdly, our accumulations are defined over any freely constructed algebraic types such as lists and trees, rather than just lists. In fact, one of our early motivations for associating higher order catamorphisms with accumulations was to find a method to make downwards tree accumulations manipulable and efficient. Gibbons[10] 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, whereas certain conditions are turned to be necessary for downwards tree accumulation to be expressed as a catamorphism. As will be seen below, using higher order catamorphisms can make Gibbons' conditions unnecessary.

Example 3.1 (Downwards tree accumulation)

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.

In fact, the function $(f, \oplus, \otimes)^{\Downarrow}$ cannot be specified by an efficient first order catamorphism if these three operations do not satisfy certain conditions, while looking for such conditions may lead to a very complicated discussion[10]. If we use a higher order catamorphism, we can describe it simply as:

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

where F_{T_A} is the functor capturing the tree structure as given in Section 2.2. This formulation makes the downwards tree accumulation be manipulable for calculating efficient parallel tree algorithms[14].

§4 Derivation of Accumulations

After formulating accumulations by higher order catamorphisms, we turn to deal with the problem of *where* to generalize a function, the first problem in the accumulation strategy given in the introduction. We shall propose a Generalization Theorem for deriving accumulations (higher order catamorphisms) from programs in terms of *first order catamorphisms*.

Lemma 4.1

Let $(\!|\phi|\!)_F$ be a first order catamorphism. If there exists a binary operator $\oplus$ with a right identity e such that $(\oplus) \circ \phi = \psi \circ F(\oplus)$ 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 4.1 tells us that the transformation from a first order catamorphism to a higher order one can be considered to find a suitable binary operator $\oplus$ for relating the original catamorphism with the newly introduced accumulating part. To see more clearly how to derive $\oplus$ 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 the following Generalization Theorem.

Theorem 4.1 (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 4.1, 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,

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

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 4.1 (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 4.1 when we derive ψ_2.

Example 4.1 ($isum$)
By way of illustration, consider the function $isum$ again. Suppose we are given the following first order catamorphism:

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

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.

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

$$
\begin{array}{ll}
 & \psi_2\ (x, p')\ y \\
= & \quad \{\ \text{By Corollary 4.1, assume } p' = (p\oplus)\ \} \\
 & \psi_2\ (x, (p\oplus))\ y \\
= & \quad \{\ \text{Generalization Theorem}\ \} \\
 & ((\oplus) \circ g_2)\ (x, (p\oplus))\ y \\
= & \quad \{\ \text{Function composition}\ \} \\
 & g_2(x, (p\oplus)) \oplus y \\
= & \quad \{\ \text{Def. of } g_2 \text{ and } \oplus\ \} \\
 & (y+) * (0 : (p \oplus x)) \\
= & \quad \{\ \text{Map}\ \} \\
 & y : ((y+) * (p \oplus x)) \\
= & \quad \{\ \text{Def. of } \oplus\ \} \\
 & y : ((y+) * ((x+) * p)) \\
= & \quad \{\ \text{Map and associativity of ``+''}\ \} \\
 & y : (((y + x)+) * p) \\
= & \quad \{\ \text{Def. of } \oplus\ \} \\
 & y : (p \oplus (y + x)) \\
= & \quad \{\ \text{Sectioning}\ \} \\
 & y : (p\oplus)(y + x) \\
= & \quad \{\ \text{Assumption above}\ \} \\
 & y : p'\ (y + x)
\end{array}
$$

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

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

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.

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, which is only used in the process of calculation neither existing nor being used in the final result, as in the following example. More practical study can be found in Section 6.2.

Example 4.2 (*subs*)
Consider the function *subs* that accepts a cons list and yields the set of all its subsequences:

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

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

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

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:

$$\begin{aligned} rev\ xs &= (\!|\phi_1' \triangledown \phi_2'|\!)\ xs\ [\,] \\ &\quad \text{where} \quad \phi_1'\ () = id \\ &\qquad\quad\ \ \phi_2'\ (a,p) = p \circ (a :) \end{aligned}$$

which is the well-known efficient accumulation as that given by Hughes[18].

Sheard[26] 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 two general promotion rules for manipulating accumulations; one is to fuse the composition of a function and an accumulation into a new accumulation, and the other is to perform transformation on accumulating parameters.

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

Theorem 5.1 (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 5.1
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 4.3, we can thus use Theorem 5.1 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 5.1, 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 5.1 does not tell anything about manipulation on accumulating parameter, which is also very important. Our following theorem is for this purpose.

Theorem 5.2 (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 5.1 by specializing h to $(\circ g)$ in the Promotion Theorem.

Example 5.2
Consider the point of the calculation in Example 5.1 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 5.2 such that

$$((\!|\eta_1 \triangledown \eta_2|\!)\ xs) \circ g = (\!|\psi_1 \triangledown \psi_2|\!)\ xs.$$

Observing that

$$\begin{array}{lll}
(\circ g) \circ \eta_1 & = & \quad \{\ \text{Theorem 5.2}\ \} \\
 & & \psi_1 \circ F_1(\circ g) \\
 & = & \quad \{\ F_1 f = id, \text{ Def. of } \phi_1\ \} \\
 & & \lambda().length \\
 & = & \lambda().(id \circ length) \\
 & = & (\circ length) \circ (\lambda().id),
\end{array}$$

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

$$\begin{array}{lll}
(\circ g) \circ \eta_2 & = & \quad \{\ \text{Theorem 5.2}\ \} \\
 & & \psi_2 \circ F_2(\circ length) \\
 & = & \quad \{\ \text{Def. of } \psi_2 \text{ and } F_2\ \} \\
 & & (\lambda(a,p).p \circ (a :)) \circ (id \times (\circ length)) \\
 & = & \lambda(a,p).p \circ length \circ (a :) \\
 & = & \quad \{\ length \circ (x :) = (1+) \circ length\ \} \\
 & & \lambda(a,p).p \circ (1+) \circ length \\
 & = & (\circ length) \circ (\lambda(a,p).p \circ (1+)) \\
 & = & \quad \{\ \text{Def. of } g\ \} \\
 & & (\circ g) \circ (\lambda(a,p).p \circ (1+))
\end{array}$$

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

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

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

Fig. 1 An Example of a Graph and its Representation

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

§6 An Application

To see how our theorems for calculating accumulations works, we shall consider a rather involved example: calculating an efficient program for the *longest subsequences paths* problem. Bird[2] 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 they 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

The longest subsequences paths 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, Fig. 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 specification of the problem reads:

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

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.

[1] Deriving an accumulation for $subs$

This derivation has been already given in Example 4.2. The result is as follows.

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

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

[2] 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 5.1.

Let $k = path\triangleleft$. We calculate ξ_1, ξ_2 from ψ_1, ψ_2 and k based on the

condition in Theorem 5.1. Observing that

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

we thus get ξ_1:

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

Now calculate ξ_2

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

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.

$$\begin{array}{ll} & k((y:)*(p\ x)) \\ = & \quad \{\ \text{By Corollary 4.1 let } p = (p'\oplus),\ \text{def. of } k\ \} \\ & path \triangleleft\ ((y:)*((x:)*p')) \\ = & \quad \{\ \text{Map}\ \} \\ & path \triangleleft\ (((y:)\circ(x:))*p') \\ = & \quad \{\ \text{Def. of } path \text{ and } \triangleleft\ \} \\ & \text{if } arc\ y\ x \text{ then } (y:)*(path \triangleleft\ ((x:)*p')) \\ & \quad \text{else } path \triangleleft\ ((x:)*p') \\ = & \quad \{\ \text{Def. of } k, \text{ and the above "let" assumption}\ \} \\ & \text{if } arc\ y\ x \text{ then } (y:)*(k\ (p\ x)) \text{ else } k\ (p\ x) \\ = & \quad \{\ \epsilon : xs = xs\ \} \\ & \text{if } arc\ y\ x \text{ then} \\ & \quad (\text{if } y \neq \epsilon \text{ then } (y:)*(k\ (p\ x)) \text{ else } k\ (p\ x)) \\ & \quad \text{else } k\ (p\ x) \\ = & \quad \{\ \text{Property of } if\ \} \\ & \text{if } arc\ y\ x \wedge y \neq \epsilon \text{ then } (y:)*(k\ (p\ x)) \text{ else } k\ (p\ x)\end{array}$$

So

$$\begin{array}{ll} & (k\circ) \circ \psi_2 \\ = & \lambda(x,p).\lambda y.k\ (p\ y) \cup \\ & (\text{if } arc\ y\ x \wedge y \neq \epsilon \text{ then } (y :) * (k\ (p\ x)) \text{ else } k\ (p\ x)) \\ = & (\lambda(x,p').\lambda y.(p'\ y) \cup \\ & (\text{if } arc\ y\ x \wedge y \neq \epsilon \\ & \quad \text{then } (y :) * (p'\ x) \text{ else } (p'\ x)) \circ F_2(k\circ). \end{array}$$

Thus we get ξ_2 defined as

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

Finally, according to Theorem 5.1, we obtain

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

[3] 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.

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

[4] Our program

In-lining the last derived accumulation in Gofer gives 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, this program shows a substantial progress. However, as some careful readers may have found, 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[1] or memoisation[23]. Since these discussions are beyond the scope of this paper, we omit it here. Alternatively, a Gofer system with embedded memoisation mechanism[28] 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.

Before closing our example, we should compare our derivation with the previous calculational approach to fusion transformation with respect to $subs$. Jeuring[19] also considered a calculational corollary to deal with longest-p subsequence problems, where p is required to be both *prefix closed*, namely

$$p\,(xs +\!\!+ [x]) \;\Rightarrow\; p\,xs$$

and *extension-equivalent*, namely

$$p\,xs \wedge p\,ys \;\Rightarrow\; p\,(xs +\!\!+ [z]) \equiv p\,(ys +\!\!+ [z]).$$

For the longest subsequence path problem, although the predicate $path$ is prefix closed, it is not extension-equivalent at all, and thus the corallary can not be directly applied to calculate its efficient program. This shows the power of our method.

§7 Related Work

Much work has been devoted to the "forcing strategy" for the derivation of an accumulation. Pettorossi[25, 24] showed, through many interesting examples, how to generalize functions in case folding fails. All these studies are usually ad-hoc and are essentially based on search-based program transformation rather than our *calculation-based* transformation. The search-based approach relies on the well-known *fold-unfold* transformations[5]. It is called "search-based" because it basically has to keep track of all occurring function calls and introduce function definitions to be searched in the folding step. The process of keeping track of function calls and controlling the steps cleverly to avoid infinite unfolding introduces substantial cost and complexity, which prevents it from being

implemented. Differently, calculational transformation puts emphasis on the exploration of recursive constructs in functions so that transformation can be performed directly by applying a direct calculational law (rule). This application of a law rule can be seen as a canned application of unfold/simplify/fold whereas memoisation of function calls can be avoided.

Our work was greatly inspired by Bird's pioneer work[2] 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 the work of specifying attribute grammars by catamorphisms[9]. But the interest there was in specification while we are interested in calculation. Another interesting work is from Meijer[21] 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 5.1 and Theorem 5.2.

We are also related to Sheard's work[26] 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.

§8 Conclusion

In this paper, we report the first attempt to systematically study accumulation strategy by using the calculational approach. This is in sharp contrast to the previous work. The main contributions of our work are as follows.

- We provide a natural but precise formulation of accumulations. We naturally capture accumulations by using higher order functions, paving the

way for later formal calculation by using catamorphisms.

- We propose two kinds of general laws for calculating accumulations: deriving possible accumulations from non-accumulational programs, and manipulating accumulations by the promotion calculation.
- We study the transformation of accumulations in calculational forms. Therefore, our study preserves the advantages of transformation in calculational forms, as seen in other work[11, 27, 16, 17]. That is, our transformation is very general in that it can be applied to recursions over any data structures other than lists, and it is correct and promising to be implemented.

We are now interested in applying our approach to the calculation of memoisation functions[23], because a memo function may be considered as a sort of accumulation whose accumulating parameter remembers all computed results of the application of the specified function.

Acknowledgment The authors would like to thank Oege de Moor for reading a draft of this paper and making a number of helpful remarks. Thanks are also to the anonymous referees for their invaluable comments.

References

1) Bird, R. Tabulation techniques for recursive programs. *ACM Computing Surveys 12*, 4 (1980), 403–417.
2) Bird, R. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems 6*, 4 (1984), 487–504.
3) Bird, R. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design* (1987), M. Broy, Ed., Springer-Verlag, pp. 5–42.
4) Bird, R., and de Moor, O. *Algebras of Programming.* Prentice Hall, 1996.
5) Burstall, R., and Darlington, J. A transformation system for developing recursive programs. *Journal of the ACM 24*, 1 (Jan. 1977), 44–67.
6) Feather, M. A survey and classification of some program transformation techniques. In *TC2 IFIP Working Conference on Program Specification and Transformation* (Bad Tolz, Germany, 1987), North Holland, pp. 165–195.
7) Fokkinga, M. A gentle introduction to category theory — the calculational approach —. Tech. Rep. Lecture Notes, Dept. INF, University of Twente, The Netherlands, Sept. 1992.
8) Fokkinga, M. *Law and Order in Algorithmics.* Ph.D thesis, Dept. INF, University of Twente, The Netherlands, 1992.
9) Fokkinga, M., Jeuring, J., Meertens, L., and Meijer, E. A translation from attribute grammars to catamorphisms. *Squiggolist* (Nov. 1990), 1–6.

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

11) Gill, A., Launchbury, J., and Jones, S. P. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, June 1993), pp. 223–232.

12) Hagino, T. *Category Theoretic Approach to Data Types.* Ph.D thesis, University of Edinburgh, 1987.

13) Henderson, P. *Functional Programming : Application and Implementation.* Prentice Hall International, 1980.

14) Hu, Z., Iwasaki, H., and Takeichi, M. Promotion strategies for parallelizing tree algorithms. In *11st Conf. Proc. Jpn Soc. for Software Sci. and Technical (JSSST '94)* (Osaka, Japan, Nov. 1994), pp. 421–424.

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

16) Hu, Z., Iwasaki, H., Takeichi, M., and Takano, A. Tupling calculation eliminates multiple data traversals. In *ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997), ACM Press, pp. 164–175.

17) Hu, Z., Takeichi, M., and Chin, W. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages* (San Diego, California, USA, Jan. 1998), pp. 316–328.

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

19) Jeuring, J. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.

20) Malcolm, G. Data structures and program transformation. *Science of Computer Programming*, 14 (Aug. 1990), 255–279.

21) Meijer, E. *Calculating Compilers.* PhD thesis, University of Nijmegen, Toernooiveld, Nijmegen, The Netherlands, 1992.

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

23) Michie, D. Memo functions and machine learning. *Nature 218* (1968), 19–22.

24) Pettorossi, A., and Proietti, M. Rules and strategies for program transformation. In *IFIP TC2/WG2.1 State-of-the-Art Report* (1993), LNCS 755, pp. 263–303.

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

26) Sheard, T., and Fegaras, L. A fold for all seasons. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, June 1993), pp. 233–242.

27) Takano, A., and Meijer, E. Shortcut deforestation in calculational form. In *Proc. Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, June 1995), pp. 306–313.

28) YAMASHITA, N. *Build-in Memoisation Mechanism for Functional Programs.* Master thesis, Dept. of Information Engineering, University of Tokyo, 1995.