Promotional Transformation of Monadic Programs

Zhenjiang Hu
Department of Information Engineering, Graduate School of Engineering
The University of Tokyo, Tokyo 113, Japan
E-mail: hu@ipl.t.u-tokyo.ac.jp

and

Hideya Iwasaki
Educational Computer Centre, The University of Tokyo
E-mail: iwasaki@rds.ecc.u-tokyo.ac.jp

ABSTRACT

Monads are becoming an increasingly important tool for structural functional programming, because they provide a uniform framework for describing a wide range of programming language features. To facilitate program transformation, Fokkinga derived a sufficient assumption under which there is a kind of so-called monadic catamorphisms which satisfy several general laws useful for the transformation of monadic programs. However, his theory is too restrictive to be applied. In this paper, we propose a new theory on monadic catamorphism by moving Fokkinga's assumption on the monad to the condition on a map between monadic algebras so that our theory is valid for arbitrary monads including, for example, the state monad that is not allowed in Fokkinga's theory. Our theory covers Fokkinga's as a special case but can be applied to a wider class of monadic programs. Many examples will be used to illustrate our idea.

1. Introduction

Monads[15,19] are becoming an increasingly important tool for structural functional programming[7,9,10], because they provide a uniform framework for describing a wide range of programming features including, for example, state, I/O, continuations, exceptions, parsing and non-determinism, without leaving the framework of a purely functional language. By using monads we can separate the concern of exceptions or states or other features from program construction, leading to a program with clear structure (see Section 3.2 for an example). However, resulting programs may be so inefficient that further program transformation is necessary.

Many studies have been devoted to the transformation of programs without monads. One promising technique is to code the control structure of programs based on the input data type structure. It has been shown that each data type comes equipped with a catamorphism[12,13] (i.e., a generalized fold[16]) which satisfies several laws being very useful for program transformation. Can this approach be applied to the transformation of programs using monads (which will be called *monadic programs* hereafter)?

It requires paying much attention to do so because the control structure of a

monadic program depends not only on its input data type structure but also on its output data type structure (i.e., monadic structure). Theoretically, Fokkinga[4] derived a sufficient assumption on monad under which there is a kind of so-called monadic catamorphisms which satisfy similar laws and can thus be used for transformation of monadic programs. Recently, Meijer and Jeuring[14] discussed the use of monadic catamorphisms in practical functional programming. They convincingly demonstrated through many examples that by programming in this style, resulting monadic programs are of an astonishing clarity, conciseness and manipulability.

Categorically speaking, monadic catamorphisms are the lifting of normal catamorphisms to the Kleisli Category[4]. This is also the approach taken in the Algebraic Design Language (ADL)[8]. Despite its mathematical elegance, Fokkinga's theory contains an assumption on monad that is not valid for several known monads, in particular it is not valid for state monad, I/O monad etc. For these monads, therefore the whole reasoning of Fokkinga's doesn't make sense (see the conclusion in his paper[4]). Meijer and Jeuring[14] solved this problem informally in their case example of deriving an efficient abstract G-machine from an initial naive monadic program related to the state monad; they defined their specific monadic catamorphism in terms of a normal catamorphism and derived the specific transformation rules based on the *fusion laws* (i.e., *promotion theorem*) equipped for the normal catamorphism. However, a general study on direct transformation of monadic catamorphisms was not given.

In this paper, we propose a new theory on monadic catamorphism by moving Fokkinga's assumption on monad to the condition of a map between monadic algebras so that our theory is valid for arbitrary monads including, for example, the state monad that is not allowed in Fokkinga's theory. Our theory covers Fokkinga's as a special case but can be applied to a wider class of monadic programs. Moreover, we shall show that Meijer and Jeuring's case study is actually an instance of our theory.

This paper is organized as follows. We briefly review some basic concepts of program calculation in Section 2, and show how to write programs using monads in Section 3. In Section 4, we propose our construction of the category of monadic F-algebras, give the definition of monadic catamorphisms, and present the promotion theorem for the transformation of monadic programs. An example of an instance of our theory for calculating G-machine is given in Section 5. Some discussions of future work are in Section 6.

2. Preliminaries for Program Calculation

We briefly review the previous work in the program calculation[2,3,6,11,12,13] and explain some basic facts which provide theoretic basis of our method. Throughout this paper, our default category C is $Sets$ with sets being objects, continuous functions being morphisms, and the composition being the general functional composition

(denoted by $\circ$).

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*.

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

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

□

Definition 2 (Separated Sum) The separated sum $X + Y$ of two types X and Y and its operation to functions are defined as follows.

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

□

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 in category C satisfying $h \circ \phi = \psi \circ F\, h$. □

Definition 5 (Category of F-algebras) The *category of F-algebras* has as its objects the F-algebras and has as its morphisms all F-homomorphisms between F-algebras. Composition in the category of F-algebra is taken from C, and so are the identities. □

It is known that an initial object in the category of F-algebras exists provided F is a polynomial functors[11]. 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, for every F-algebra (X, ϕ) there exits precisely a unique $f :: T \to X$ such that

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

We denote this unique solution of f by $(\![\phi]\!)_F$, and call it *F-catamorphism*. If the functor F is clear from the context, we omit the subscript F in $(\![\phi]\!)_F$ and in_F, and call $(\![\phi]\!)_F$ catamorphism. Catamorphisms play an important role in program transformation (program calculation) in that they satisfy a number of nice calculational properties among which *promotion* is of greatest importance. Promotion theorem gives the condition that has to be satisfied in order to "promote" a function into a catamorphism to obtain a new catamorphism.

Theorem 1 (Promotion)

$$f \circ \phi = \psi \circ F\ f \quad \Rightarrow \quad f \circ (\![\phi]\!) = (\![\psi]\!) \qquad \square$$

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

$$L\ a = Nil \mid Cons\ (a, L\ a)$$

is defined as the initial object $(L\ a,\ Nil \triangledown Cons)$ 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

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

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

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

3. Monadic Programming

In this section, we briefly review the basic concept of monads and show a simple example of how to write programs using monads.

3.1. Monads

Wadler[19] defines a monad as a unary type constructor M together with two functions *result* and *bind* whose types are given by:

$$\begin{array}{lcl} result & :: & a \to M\ a \\ bind & :: & M\ a \to (a \to M\ b) \to M\ b. \end{array}$$

In addition, these two functions are required to satisfy a number of laws. The left-unit law and right-unit law remove occurrences of *result* from an expression.

$$\begin{array}{lcl} result\ a\ `bind`\ k & = & k\ a \\ m\ `bind`\ result & = & m \end{array}$$

Furthermore, *bind* has to be associative, i.e.,

$$(m\ `bind`\ \lambda x \to f\ x)\ `bind`\ g = m\ `bind`(\lambda x \to f\ x\ `bind`\ g).$$

For instance, the exception monad used to model programs with exception is defined by the type constructor *Maybe*

$$Maybe\ a = Just\ a \mid Nothing$$

with the two functions *result* and *bind* defined by

$$\begin{array}{lcl} result\ a & = & Just\ a \\ Nothing\ `bind`\ f & = & Nothing \\ Just\ a\ `bind`\ f & = & f\ a. \end{array}$$

An intersting use of monads is to model programs that make use of an internal state. Computation of this kind can be represented by function of type $s \to (s, a)$ (often referred to as *state transformer*) mapping an initial state to a pair containing the next state and the result. It can be defined as the monad *State s* as follows.

$$\begin{array}{lcl} State\ s\ a & = & s \to (s, a) \\ result\ a & = & \lambda s \to (s, a) \\ m\ `bind`\ f & = & \lambda s \to \mathbf{let}\ (s', a) = m\ s\ \mathbf{in}\ f\ a\ s'. \end{array}$$

Definition 6 (M-monadic Function) Let M be a monad. A function is said to be *M-monadic* if its type is $a \to M\ b$ for some types a and b. □

If M is clear from the context, we may simply say monadic function instead of M-monadic function. As will be seen later, the well known *Kleisli Category* concerning monadic functions is very important in our theory.

3.2. An Example of Programming with Monads

To help understanding the use of monads in programming and the motivation of promotional transformation on monadic programs, consider the following simple problem. Given a binary tree in which each leaf has a name and given a list of pairs associating names with values, we are asked to calculate the sum of values associated with tree leaves. For example, if the tree is $Node\ (Leaf\ \text{“good”}, Leaf\ \text{“luck”})$ and the associated list is $[(\text{“good”}, 2), (\text{“luck”}, 4)]$, we hope to give the result of 6. If there

is a leaf whose associated value does not exist in the associated list, an error should be returned. We may solve the problem by defining a function st:

$$st :: Tree\ String \to [(String, Value)] \to Maybe\ Value.$$

Although the problem is simple, programming is not so easy because it usually has to carry the association list explicitly all the way and to pay much attention to any possibility of error happening. To avoid this, we may define a suitable monad so that programs can be constructed in a quite simple way over this monadic computation. So we define the monad $RM\ r$ as follows.

$$\begin{array}{lcl} RM\ r\ x & = & r \to Maybe\ x \\ result\ x & = & \lambda_ \to Just\ x \\ m\ `bind`\ f & = & \lambda r \to m\ r\ `bind`\ \lambda a \to f\ a\ r \\ zero & = & \lambda_ \to Nothing \end{array}$$

The monad $RM\ r$ is in fact a composition of two standard monads the *Reader Monad* and the *Exception Monad*[7]. Now we can rewrite the type of st to

$$st :: Tree\ String \to RM\ [(String, Value)]\ Value.$$

The st takes a tree of type $Tree\ String$ and yields a computation of the monad $RM\ [(String, Value)]$. The result of this computation is a value.

According to the specification of the problem, we define st concisely as follows.

$$\begin{array}{lcl} st\ t & = & subst\ t\ `bind`\ \lambda t' \to result\ (sumtr\ t') \\ \\ subst & :: & Tree\ String \to RM\ [(String, Value)]\ (Tree\ Value) \\ subst\ (Leaf\ a) & = & lookup\ a\ `bind`\ \lambda b \to result\ (Leaf\ b) \\ subst\ (Node\ (l, r)) & = & subst\ l\ `bind`\ \lambda l' \to subst\ r\ `bind`\ \lambda r' \to \\ & & result\ (Node\ (l', r')) \\ \\ sumtr & = & Tree\ Value \to Value \\ sumtr & = & (\!|id, (+)|\!) \end{array}$$

It uses two functions: $subst$ for replacing each leaf in the tree with its associated value and $sumtr$ for summing up all leaves. Besides, it uses a monadic function

$$lookup :: String \to RM\ [(String, Value)]\ Value$$

which returns $result \circ v$ if the name has associated value v and returns $zero$ otherwise. It is worth noting here that we do not need to care about error happening and do not need to carry explicitly the association list all the way in the definition of st.

One big problem with functional programs is that some intermediate results might be produced and then be consumed, resulting in inefficient programs[17,18]. Monadic

programs are not an exception. For example, the definition of st has two parts composed with $bind$; $subst\ t$ produces a whole tree which is then consumed by $result \circ sumtr$. Can we fuse them together to make it efficient?

Fokkinga[4] solved this problem by structuring monadic programs with monadic catamorphisms for which there is a general promotion rule for fusing monadic programs. Although we can make st efficient by his theory, it contains an assumption on monads that is not valid for manipulating many monadic programs (see Section 5). In this paper, we aim to give a more general theory for fusing monadic programs.

4. Theory for Manipulating Monadic Programs

We aim at a general theory on promotional transformation of *monadic* programs. Before doing so, let's recall how we have done for *normal* programs as in Section 2. First of all, we have the base category C. We then build the category of F-algebras upon C. Finally, we define catamorphisms as homomorphisms from the initial F-algebra to another F-algebra and derive its Promotion Theorem. In this section, we shall follow this train of thought, defining monadic catamorphisms and constructing a general theory for manipulating monadic programs. In what follows, we assume that F is an endofunctor from C to C, and that (T, in) is the initial F-algebra.

4.1. The Base Category for Monadic Catamorphisms

From the viewpoint of typing, a so-called monadic catamorphism[4,14], denoted by $(\!|\ _\ |\!)$, should act on *monadic functions*:

$$\frac{\phi :: F\ X \to M\ X}{(\!|\phi|\!) :: T \to M\ X}$$

Thus the base category for monadic catamorphisms would be naturally *Kleisli Category* whose morphisms are monadic functions.

Definition 7 (Base Category C^M) Let M be a monad with $bind$ and $result$ operations. The base category for monadic catamorphisms, denoted as C^M, is defined as a Kleisli category[1]:

- Whose objects are sets;
- Whose morphisms are monadic functions, i.e., given two objects X and Y, the morphism from X to Y is the monadic function with type $X \to M\ Y$;
- Whose associative composition operator is @, defined by

$$f\ @\ g = \lambda x \to g\ x\ `bind`\ f;$$

- Whose identity is the monadic function $result$. □

Notice the difference between morphisms and monadic functions in C^M. The monadic function of type $X \to M\,Y$ denotes a morphism from object X to object Y instead of a morphism from X to $M\,Y$. To make this difference clear, we may use :: for typing of functions and and : for mapping of morphisms.

Recall that our default category C has as objects sets and has as morphisms functions from one type to another. Now, we know that C^M has as objects sets but has as morphisms monadic functions. In fact, there is an adjunction[1] between C and C^M, which is quite important since it provides means to discuss properties of C^M in terms of C.

4.2. Category of Monadic F-Algebras

To define monadic catamorphisms and the Promotion Theorem, we need to construct a category with monadic algebras as objects and monadic catamorphisms as some of its morphisms. To this end, first we define monadic algebras.

Definition 8 (Monadic F-algebra) A monadic F-algebra is a pair (X, ϕ), where X is an object in C^M and ϕ is a monadic function denoting a morphism in C^M from object $F\,X$ to X. □

For example, $(T, result \circ in)$ is a monadic F-algebra. Note that in the above definition we can apply F on an object of category C^M because we know C and C^M have the same objects.

Second, we set about to define the structure-preserving map between monadic F-algebras. Since we hope such a map could be a morphism in our future category of monadic algebras ($Alg(F, *)$), we have to define it carefully while imposing as less restriction as possible. To capture the structure of monadic F-algebras, we define a derivation of F, denoted by F^*, inducting over the construction of functor F. It maps a monadic function to another one, i.e., $F^* :: (X \to M\,Y) \to (F\,X \to M\,(F\,Y))$.

Definition 9 (F^*)

$$
\begin{array}{lcll}
F = F_1 + F_2 & \Rightarrow & F^*\,f & = F_1^*\,f + F_2^*\,f \\
F = F_1 \times F_2 & \Rightarrow & F^*\,f\,(x_1, x_2) & = F_1^*\,f\,x_1\ \text{`}bind\text{`}\ \lambda y_1 \to \\
 & & & \quad F_2^*\,f\,x_2\ \text{`}bind\text{`}\ \lambda y_2 \to result\,(y_1, y_2) \\
F = I & \Rightarrow & F^*\,f & = f \\
F = !a & \Rightarrow & F^*\,f & = result
\end{array}
$$

□

Note the definition of F^* for the case of $F = F_1 \times F_2$. It determines a computing order from $F^*\,f\,x_1$ to $F^*\,f\,x_2$. As a matter of fact, we may change this order to give other proper definitions of F^*. The two properties that F^* has to possess are:

- **Identity property**: $F^*\,result = result$;
- **Separable property**: there exists a unique d_{F^*} such that $F^*\,f = d_{F^*} \circ F\,f$.

Obviously, the above definition of F^* satisfies the identity property. For the separable property, $d_{F^*} = F^*\, id$, which can be in-lined as follows.

$$\begin{array}{lcll} F = F_1 + F_2 & \Rightarrow & d_{F^*}\,(x_1 + x_2) & = & d_{F^*}\,x_1 + d_{F^*}\,x_2 \\ F = F_1 \times F_2 & \Rightarrow & d_{F^*}\,(x_1, x_2) & = & d_{F^*}\,x_1\ `bind`\ \lambda y_1 \rightarrow \\ & & & & \quad d_{F^*}\,x_2\ `bind`\ \lambda y_2 \rightarrow result\ (y_1, y_2) \\ F = I & \Rightarrow & d_{F^*}\,x & = & x \\ F = !a & \Rightarrow & d_{F^*}\,x & = & result\ x \end{array}$$

It should be noted that F^* is not necessary to be a functor satisfying $F^*(f \mathbin{@} g) = F^* f \mathbin{@} F^* g$ for any f and g, the assumption in Fokkinga's theory[4]. It is this assumption that makes his theory invalid for several known monads. Instead, we shift his assumption to the condition of a structure-preserving map (see Definition 10 below) and make our theory valid for arbitrary monads. We will return to this point later.

Example 1 (F_T^*) Consider the functor F_T defined in Section 2.3, the F_T^* is as follows.

$$\begin{array}{lcl} F_T^*\ f & = & result + \lambda(x_1, x_2) \rightarrow f\ x_1\ `bind`\ \lambda y_1 \rightarrow \\ & & \quad f\ x_2\ `bind`\ \lambda y_2 \rightarrow result\ (y_1, y_2) \end{array}$$

□

After obtaining F^*, we can define the structure-preserving map between two monadic F-algebras.

Definition 10 (Structure-preserving Map) Let (X, ϕ) and (Y, ψ) be two monadic F-algebras. A structure-preserving map from (X, ϕ) to (Y, ψ) is a morphism h from object X to object Y in the category C^M satisfying

$$\begin{array}{lllcl} (1) & h \mathbin{@} \phi & = & \psi \mathbin{@} F^* h & \\ (2) & F^* h \mathbin{@} F^* g & = & F^*(h \mathbin{@} g) & \text{for any } g \end{array}$$

□

Our definition of structure-preserving map has an additional condition (2), compared with Fokkinga's. At the first glance, it seems that we impose more restriction. On the contrary, we have less restriction, because Fokkinga actually required that $F^* h \mathbin{@} F^* g = F^*(f \mathbin{@} g)$ for any monadic functions h and g, while we only require that this equation for a *specific* monadic function h holds for any monadic function g.

Finally, we construct the category of monadic F-algebras with respect to a monad M.

Theorem 2 (Category of monadic F-algebras) The following is the construction of monadic F-algebras, denoted by $Alg(F, *)$.

- It has as objects monadic F-algebras.
- It has as morphisms structure-preserving maps.
- The composition of morphisms is @.
- The identity morphism is $result$.

Proof: To see that the above construction establishes a category, we have to prove that (a) The $result$ is a morphism, i.e., a structure-preserving map; (b) Given two morphisms $h_1 : (X, \phi) \to (Y, \psi)$ and $h_2 : (Y, \psi) \to (Z, \eta)$, then $h_2 \mathbin{@} h_1$ is a morphism from (X, ϕ) to (Z, η); (c) The composition @ is associative; (d) $result \mathbin{@} h = h \mathbin{@} result = h$.

Since our category is built upon C^M, (c) and (d) obviously hold. To prove (a), we have to show that $result$ satisfies two conditions for being a morphism, as is easily verified. To prove (b), we proceed the following two calculations to show that $h_2 \mathbin{@} h_1$ satisfies two conditions for being a morphism.

$$
\begin{array}{ll}
 & h_2 \mathbin{@} h_1 \mathbin{@} \phi \\
= & \quad \{ \text{ condition (1) for } h_1 \text{ being a morphism } \} \\
 & h_2 \mathbin{@} \psi \mathbin{@} F^* h_1 \\
= & \quad \{ \text{ condition (1) for } h_2 \text{ being a morphism } \} \\
 & \eta \mathbin{@} F^* h_2 \mathbin{@} F^* h_1 \\
= & \quad \{ \text{ condition (2) for } h_2 \text{ being a morphism } \} \\
 & \eta \mathbin{@} F^* (h_2 \mathbin{@} h_1)
\end{array}
$$

$$
\begin{array}{ll}
 & F^* (h_2 \mathbin{@} h_1) \mathbin{@} F^* g \\
= & \quad \{ \text{ condition (2) for } h_2 \text{ being a morphism } \} \\
 & F^* h_2 \mathbin{@} F^* h_1 \mathbin{@} F^* g \\
= & \quad \{ \text{ condition (2) for } h_1 \text{ being a morphism } \} \\
 & F^* h_2 \mathbin{@} F^* (h_1 \mathbin{@} g) \\
= & \quad \{ \text{ condition (2) for } h_2 \text{ being a morphism } \} \\
 & F^* (h_2 \mathbin{@} h_1 \mathbin{@} g)
\end{array}
$$

□

4.3. Monadic Catamorphisms

In this section, we shall define monadic catamorphisms and give the Promotion Theorem for them.

Proposition 3 (Monadic Catamorphism) Let (T, in) be the initial F-algebra, F^* be a derivation of F with respect to a monad M. For any monadic F-algebra (X, ϕ), there exists a unique monadic function $h :: T \to M\, X$ satisfying

$$h \circ in = \phi \mathbin{@} F^* h$$

Here h will be called *monadic catamorphism* and will be denoted by $\langle\!|\phi|\!\rangle_{F^*}$. If F^* is clear from the context, we may omit the subscript F^* in $\langle\!|_|\!\rangle_{F^*}$.

Proof: To prove it, we calculate the equation and find the explicit definition of h as follows.

$$
\begin{array}{ll}
 & h \circ in = \phi \mathbin{@} F^* h \\
\equiv & \quad \{ \text{ Separable property of } F^* \ \} \\
 & h \circ in = \phi \mathbin{@} (d_{F^*} \circ F\, h) \\
\equiv & \quad \{ \text{ by the fact } f \mathbin{@} (g \circ h) = (f \mathbin{@} g) \circ h \ \}
\end{array}
$$

$$
\begin{array}{ll}
& h \circ in = (\phi \mathbin{@} d_{F^*}) \circ F\, h \\
\equiv & \quad \{ \text{ uniqueness of the normal catamorphism } \} \\
& h = (\!|\phi \mathbin{@} d_{F^*}|\!)_F
\end{array}
$$
□

Example 2 (Monadic catamorphism over tree) The *subst* given in Section 3.2 is a monadic catamorphism over tree data structure.

$$
\begin{array}{lll}
subst & = & (\!|\phi_1, \phi_2|\!)_{F_T^*} \\
& & \phi_1\ a\ = lookup\ a\ \text{`}bind\text{`}\ \lambda b \to result(Leaf\ b) \\
& & \phi_2\ (l,r)\ =\ result\ (Node\ (l,r))
\end{array}
$$
□

The promotion theorem for manipulating monadic catamorphisms is as follows.

Theorem 4 (Monadic Promotion) If h is a morphism from (X, ϕ) to (Y, ψ) in the category $Alg(F, *)$, i.e.,

$$
\begin{array}{lllll}
(1) & h \mathbin{@} \phi & = & \psi \mathbin{@} F^* h & \\
(2) & F^* h\ @\ F^* g & = & F^*(h \mathbin{@} g) & \text{for any morphism } g
\end{array}
$$

then

$$h \mathbin{@} (\!|\phi|\!) = (\!|\psi|\!).$$

Proof: We prove the theorem by the following calculation.

$$
\begin{array}{ll}
& h \mathbin{@} (\!|\phi|\!) = (\!|\psi|\!) \\
\equiv & \quad \{ \text{ Proposition 3 } \} \\
& h \mathbin{@} (\!|\phi|\!) \circ in = \psi \mathbin{@} F^*(h \mathbin{@} (\!|\phi|\!)) \\
\equiv & \quad \{ \text{ Condition (2) } \} \\
& h \mathbin{@} (\!|\phi|\!) \circ in = \psi \mathbin{@} F^* h \mathbin{@} F^* (\!|\phi|\!) \\
\equiv & \quad \{ \text{ Proposition 3 } \} \\
& h \mathbin{@} \phi \mathbin{@} F^* (\!|\phi|\!) = \psi \mathbin{@} F^* h \mathbin{@} F^* (\!|\phi|\!) \\
\Leftarrow & \quad \{ \text{ trivial } \} \\
& h \mathbin{@} \phi = \psi \mathbin{@} F^* h
\end{array}
$$
□

Why is our promotion theorem able to be applied to a wider class of monadic functions than Fokkinga's? Recall that his assumption, $F^* h \mathbin{@} F^* g = F^*(h \mathbin{@} g)$ for any monadic functions h and g, is an implicit condition in his promotion theorem. We believe that h is not necessary to range over all functions. In practical program calculation, h is usually given and asked to be promoted into a monadic catamorphism. Therefore, as in our promotion theorem, we only requires that for a *specific* monadic function h and any monadic function g, $F^* h \mathbin{@} F^* g = F^*(h \mathbin{@} g)$.

Example 3 (Promotional Transformation on *st*) Recall the definition of *st* in Section 3.2:

$$st\ t = subst\ t\ \text{`}bind\text{`}\ \lambda t' \to result\ (sumtr\ t').$$

We know, as in Example 2, that *subst* is a monadic catamorphism. So we can rewrite *st* to the following.

$$st = (result \circ sumtr)\ @(\!|\phi_1 \triangledown \phi_2|\!)_{F_T^*}$$

Based on Theorem 4, we can derive the following efficient monadic program[5]:

$$\begin{array}{lcl} st & = & \langle\!|\psi_1, \psi_2|\!\rangle \\ & & \psi_1 = result\,@\,lookup \\ & & \psi_2 = result \circ + . \end{array}$$

□

5. An Example

In this section, we adopt the impressive example provided by Meijer and Jeuring[14], which calculates efficient *G*-machine. We intend to show their four promotable conditions, informally provided and acted as the kernel strategy in their calculation, are just an instance of our promotion theorem. We will not address the calculation already done by Meijer and Jeuring[14] and only explain our main idea.

5.1. G-Machine

In the definition of *G*-machine, a state monad is used. Sharing and graph manipulation that goes on in a real graph reduction implementation can be modeled using the state monad whose state is a graph and whose computation result is a pointer (represented by Int) to a node of the graph. An naive G-machine evaluates an expression e by first building a graph for e and then evaluating the resulting graph:

$$\begin{array}{lcl} machine & :: & [Int] \to Expr \to State\ Graph\ Int \\ machine\ ps\ e & = & (eval\,@(build\ ps))\ e \end{array}$$

Evaluating an expression is rather inefficient: *build ps* builds a complete graph, which is subsequently consumed by function *eval*.

Since the monadic program *machine* is defined over the state monad, Fokkinga's theory cannot be used to calculate it to an efficient one. In the following we shall demonstrate an instance of our theory for this specific problem, giving the definition of monadic catamorphisms over $Expr$ and proposing the corresponding promotion theorem.

5.2. Expressing build ps by a Monadic Catamorphism over Expr

Consider a tiny first order language[14] in which all values are integers, and the only operator is addition. Formal parameters are encoded by de Bruijn indices. The

expressions are elements of the data type $Expr$:

$$\begin{array}{lcll} Expr & = & Var\ Int & \text{Variable} \\ & | & Con\ Int & \text{Constant} \\ & | & Add\ (Expr, Expr) & \text{Addition} \\ & | & App\ (Name, Expr) & \text{Function application} \end{array}$$

This data type can be defined as the initial F-algebra $(Expr, in_F)$, where $in_F = Var \triangledown Con \triangledown Add \triangledown App$ and F is an endofunctor defined by

$$F\ = !Int + !Int\ +\ I \times I + !Name \times I.$$

Now we can define a proper derivation of F as follows.

$$\begin{array}{lcl} F^*\ f & = & result\ \triangledown \\ & & result\ \triangledown \\ & & \lambda(e_1, e_2) \rightarrow f\ e_1\ `bind`\ \lambda e_1' \rightarrow \\ & & \qquad f\ e_2\ `bind`\ \lambda e_2' \rightarrow result\ (e_1', e_2')\quad \triangledown \\ & & \lambda(n, es) \rightarrow f\ es\ `bind`\ \lambda es' \rightarrow result\ (n, es') \end{array}$$

According to Proposition 3, a monadic catamorphism over $Expr$ is defined by

$$\begin{array}{lcl} (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!) & :: & Expr \rightarrow M\ Y \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (Var\ v) & = & \phi_1\ v \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (Con\ c) & = & \phi_2\ c \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (Add\ (e_1, e_2)) & = & (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ e_1\ `bind`\ \lambda e_1' \rightarrow \\ & & (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ e_2\ `bind`\ \lambda e_2' \rightarrow \\ & & \phi_3\ (e_1', e_2') \\ (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ (App\ (n, e)) & = & (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)\ e\ `bind`\ \lambda e' \rightarrow \\ & & \phi_4\ (n, e') \end{array}$$

The above monadic catamorphism over $Expr$ is exactly the same as what Meijer and Jeuring gave except that it is defined directly rather than in terms of a normal catamorphism. In fact, *build ps* can be defined as such a monadic catamorphism[14]. We omit the detail but assume that

$$build\ ps = (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!).$$

5.3. Promoting eval into build ps

Now we are faced with a monadic program $eval\ @(build\ ps)$, and we hope to promote *eval* into *build ps* to obtain an efficient one. As we know $build\ ps = (\!|\phi_1 \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4|\!)$, we can derive the general transformation rule for this calculation by instantiating Theorem 4.

Our rule says that $eval \mathbin{@} (build\ ps) = (\!|\psi_1 \triangledown \psi_2 \triangledown \psi_3 \triangledown \psi_4|\!)$, provided that

$$
\begin{array}{lllll}
(a) & eval \mathbin{@} \phi_1 & = & \psi_1 & \\
(b) & eval \mathbin{@} \phi_2 & = & \psi_2 & \\
(c_1) & eval \mathbin{@} \phi_3 & = & \lambda(e_1, e_2) \to eval\ e_1\ \text{`}bind\text{`}\ \lambda e_1' \to & \\
 & & & \qquad eval\ e_2\ \text{`}bind\text{`}\ \lambda e_2' \to & \\
 & & & \qquad \psi_3\ (e_1', e_2') & \\
(c_2) & F^* eval \mathbin{@} F^*\ g & = & F^*\ (eval \mathbin{@} g) \quad \text{for any } g & \\
(d) & eval \mathbin{@} \phi_4 & = & \lambda(n, e) \to eval\ e\ \text{`}bind\text{`}\ \lambda e' \to \psi_4\ (n, e'), &
\end{array}
$$

which is essentially the same as the four conditions informally provided by Meijer and Jeuring[14]. Note that we have (c_1) and (c_2) which implies Meijer and Jeuring's corresponding one condition.

Using the above rule, we can go on deriving ψ_i's and thus obtain an efficient monadic program for G-machine[14].

6. Future Work

So far, we have only studied the transformation of monadic programs with respect to one monad. We are trying to fuse monadic programs with respect to different monads. Besides, we believe that our work on monadic catamorphisms can be extended to its dual, *monadic anamorphism*.

Acknowledgements

We are grateful to Prof. Masato Takeichi for his helpful discussions and numerous suggestions on this work. Many thanks are also to Akihiko Takano and Peter Orbaek for their comments on an earlier draft of this paper.

References

1. M. Barr and C. Wells, editors. *Category Theory for Computing Science*. Prentice Hall, 1990.
2. 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.
3. M. Fokkinga. A gentle introduction to category theory — the calculational approach —. Technical Report Lecture Notes, Dept. INF, University of Twente, Netherlands, September 1992.
4. M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical report, Dept. INF, University of Twente, Netherlands, June 1994.

5. Z. Hu, H. Iwasaki, and M. Takeichi. Promotional transformation of monadic programs. Technique report METR 95–05, Department of Mathematical Engineering and Information Physics, University of Tokyo, June 1995.
6. J. Jeuring. *Theories for Algorithm Calculation.* Ph.D thesis, Faculty of Science, Utrecht University, 1993.
7. M.P. Jones and L. Duponcheel. Composing monads. Report yaleu/dcs/rr-1004, Department of Computer Science, Yale University, December 1993.
8. R.B. Kieburtz and J. Lewis. Algebraic design language. Technical Report OGI, Tech-report 94-002, Dept. of Computer Sceience and Engineering, Oregon Graduate Institution of Science and Technology, 1994.
9. D.J. King and P. Wadler. Combining monad. In *Proc. Glasgow Workshop on Functional Programming.* Springer-Verlag, 1993.
10. S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Proc. ACM symposium on principles of programming languages*, pages 333–343, San Francisco, 1995.
11. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, (14):255–279, August 1990.
12. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. Conference on Functional Programming Languages and Computer Architecture (LNCS 523)*, pages 124–144, Cambridge, Massachusetts, August 1991.
13. 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.
14. E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *Proc. 1st International Springschool on Advanced Functional Progamming Techniques (LNCS 925)*, May 1995.
15. E. Moggi. An abstract view of programming languages. Technical Report ECS–LSCS–90–113, LFCS, University of Edinburgh, Edingurgh, Scotland, 1990.
16. 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.
17. 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.
18. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. of ESOP (LNCS 300)*, pages 344–358, 1988.
19. P. Wadler. The essence of functional programming. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.