Promotional Transformation on Monadic Programs

Zhenjiang Hu * Hideya Iwasaki † Masato Takeichi ‡

1 Introduction

Monads, proposed by Moggi [16] of their use in structuring denotational descriptions and then popularized by Wadler[21], are becoming an increasingly important tool for structural functional programming[8, 10, 11]. The reason is that monads provide a uniform framework for describing a wide range of programming language features including, for example, state, I/O, continuations, exceptions, parsing and non-determinism, without leaving the framework of a purely functional language.

Programs using monads, which will be called *monadic programs* hereafter, can be structured once again over data type for facilitating program transformation [5, 9, 15]. It is known that each data type comes equipped with a catamorphism[13, 14] (i.e., a generalized fold[18]) which satisfies several laws that are very useful for program transformation. Fokkinga[5] derived a sufficient condition under which there is also a kind of so-called monadic catamorphism which satisfy similar laws and can thus be used for transformation of monadic programs.

Apart from the theoretical study, very recently, Meijer and Jeuring[15] discussed the simultaneous use of catamorphisms and monads in practical functional programming. They convincingly demonstrated through many examples that by programming in this style resulting programs are of an astonishing clarity and conciseness.

Categorically speaking, monadic catamorphisms are the lifting of normal catamorphisms to the Kleisli category[5]. This is also the approach taken in the Algebraic Design Language (ADL) [9]. 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 the state monad. Therefore, for these monads, the whole reasoning of Fokkinga's doesn't make sense (see the conclusion in [5]).

Meijer and Jeuring[15] solved this problem informally in their case study 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 found some transformation rules for the calculation of monadic programs based on the *fusion laws* (i.e., *promotion theorem*) for the normal catamorphism. However, a general study on direct transformation of monadic catamorphisms was not given.

* 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) .

In this paper, we propose a new theory on monadic catamorphism by moving Fokkinga's assumption on the 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. Moreover, Meijer and Jeuring's informal transformation rules of monadic programs in their case study is actually an instance of our general promotion theorem.

The rest of this paper is organized as follows. We review briefly some basic concepts on program calculation in Section 2 and the definition of monads in Section 3. To help readers be familiar with monadic programming and understand the motivation of the promotional transformation on monadic programs, we give a simple example in Section 4. In Section 5, 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 catamorphisms. An example of an instance of our theory for calculating G-machine is given in Section 6. Some discussions are given in Section 7.

2 Preliminaries for Program Calculation

In this section, we briefly review the previous work in the program calculation[1, 3, 4, 6, 7, 12, 13, 14] and explain some basic facts which provide theoretic basis of our method. In this paper, our default category C has as objects sets, has as morphisms continuous functions, and has as composition of general functional composition $\circ$.

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*. We follow the definitions of product, separated sum functors, and related combinators as in [19].

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

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

Following combinators(left/right projections and split $\triangle$) are related to the product functor:

$$\begin{aligned} exl\ (x, y) &= x \\ exr\ (x, y) &= y \\ (f \triangle g)\ x &= (f\ x, g\ x) \end{aligned}$$

□

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

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

Following combinators(left/right injections and junc $\triangledown$) are related to the separated sum functor:

$$\begin{array}{lll} inl\ x & = (0, x) \\ inr\ y & = (1, y) \\ (f \triangledown g)\ (0, x) & = f\ x \\ (f \triangledown g)\ (1, y) & = g\ y \end{array}$$

□

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 in category C satisfying $h \circ \phi = \psi \circ Fh$.

□

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[12]. 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 single $f :: T \to X$ such that

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

We denote the unique solution of f of 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)

$$f \circ \phi = \psi \circ F\ f \quad \Rightarrow \quad f \circ (\!|\ \phi\ |\!) = (\!|\ \psi\ |\!)$$

□

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

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

is defined as the initial object $(L\ a, Nil \triangledown Cons)$ of the category of F_L-algebras, where F_L is the endofunctor defined by

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

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

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

3 Monad

Wadler[20] defines a monad as a unary type constructor M together with two functions *result* and *bind* whose types are given by:

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

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

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

Furthermore, *bind* has to be associative.

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

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

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 final state and the result. This state transformer can be defined as the monad $State\ s$:

$$State\ s\ a = s \to (s, a)$$

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

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

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

□

If M is clear from the context, we may say monadic function instead of M-monadic function. The well known *Kleisli category* has as objects types and has as morphisms M-monadic functions, which will be very important for our later discussion.

4 Monadic Programming

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 tree whose leaves have the type of string indicating their names and a list associating names with values, we are asked to sum up all leaves after replacing every leaf with its associated value. If there is a leaf whose associated value does not exist in the associated list, an error should be returned.

We can solve the problem by defining a function st having the type of

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

Programming st usually requires us to carry the association list all the way and pay much attention on an error happening. If we use monads in programming, we could embed this requirement in the definition of the two functions related to a monad, namely *result* and *bind*. So we define a monad $RM\ r$ as follows.

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

The monad $RM\ r$ is in fact a composition of two standard monads the *reader monad* and the *exception monad*, and so the definition of *result* and *bind* can be derived from those of the two standard monads by means of composing monads[8].

Using the monad $RM\ r$, we can rewrite the type of st to

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

which reads that st takes a data of type $Tree\ String$ and yields a result of type $Tree\ Value$ through the computation of monad $RM\ [(String, Value)]$.

In order to define st, we need an auxiliary function

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

which returns $zero$ if the given name has no associated value and return $result \circ v$ if the name has associated value v.

Now we can define st with two functions: $subst$ for replacing each leaf with its associated value and $sumtr$ for summing up the replaced tree. Unlike programming without monads, we do not need to pay much attention to the error happening and do not need to carry explicitly the association list all the way.

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

One problem with monadic programs is that some intermediate result might be produced and then be consumed, resulting in an inefficient program. Consider the definition of st. It has two parts composed together by $bind$ operator, with $subst\ t$ produced a whole intermediate tree which then consumed by $result \circ sumtr$. Can we fuse them together to make it efficient?

Fokkinga[5] tried solving this problem by structuring monadic programs as monadic catamorphisms for which there is a general promotion rule for fusing monadic programs. Though being applicable to the example in this Section, his theory contains an assumption on monads that is not valid for manipulating many monadic programs one of which can be seen in Section 6. In this paper, we aims to give a more general theory for fusing monadic programs.

5 Monadic Catamorphisms

We aim to define monadic catamorphisms and propose general transformation rules. Before doing so, let's recall how the normal catamorphism is defined. First of all, we have the base category C. We then build the category of F-algebras upon C. After that, we define catamorphisms as homomorphisms from the initial F-algebra to another F-algebra. In this section, we shall follow this train of thought to define monadic catamorphisms.

5.1 The Base Category for Monadic Catamorphisms

Let F be an endofunctor on C, and let (T, in) be the initial F-algebra. Informally, a so-called monadic catamorphism[5, 15], denoted by $\langle\!|\ -\ |\!\rangle$, should act on *monadic functions*:

$$\frac{\phi :: F\ X \to M\ X}{\langle\!|\ \phi\ |\!\rangle :: T \to M\ X}$$

Now care should be taken of its typing. Comparing typing with the normal catamorphism:

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

One may notice that the type of ϕ is different.

Now we turn to give the definition of our base 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[2]:

- 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 in what follows.

5.2 Adjunction between C and C^M

Recall that our default category C has as objects sets and has as morphisms functions from one type to another. Now, we know that that C^M has as objects sets but has as morphisms monadic functions. What is the relationship between them?

Barr and Wells[2] present an adjunction between C and C^M. The two functors are

$$\begin{array}{lcl} -^M & : & C \to C^M \\ X^M & = & X \\ f^M & = & result \circ f \end{array}$$

$$\begin{array}{lcll} U & : & C^M \to C & \\ U\ X & = & M\ X & \\ U\ f & = & \lambda m \to m\ `bind`\ f & \\ & :: & M\ X \to M\ Y & \text{whenever } f :: X \to M\ Y \end{array}$$

Thus $-^M$ raise the target of a function from Y to $M\ Y$, and U "rebalanced" this by further raising the source of a monadic function from X to $M\ X$. It is trivial to verify the *adjunction property*:

$$f = Ug \circ result \quad \equiv \quad id\ @\ f^M = g.$$

The adjunction between C and C^M provides us means to discuss properties of C^M in terms of C.

5.3 Category of Monadic F-Algebras

In this section, we shall give the definition of the monadic algebra and show how to construct a category with monadic algebras as objects. In what follows, we assume that F is an endofunctor from C to C, and that (T, in) is the initial F-algebra.

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.

We are going to construct a category whose objects are monadic F-algebras and whose morphisms are structure-preserving maps between monadic F-algebras. To this end, we first define a derivation of F, denoted by F^*, inducting over the construction of functor F. F^* maps a monadic function to another monadic function, i.e., $F^* :: (X \to M\ Y) \to (F\ X \to M\ (F\ Y))$.

Definition 9 (F^*)

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

□

Note the definition of F^* in case $F = F_1 \times F_2$. It determines a computing order from $F^*\ f\ x_1$ to $F^*\ f\ x_2$, which is not a necessary requirement. We may change this order to give another proper definition of F^*. The F^* has two important properties:

- **Identity property**: $F^*\ result = result$
- **Separable property**: $F^*\ f = d_F \circ F\ f$, where d_F is defined as follows.

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

which are easy to be verified.

It should be noted that we do not require F^* be a functor satisfying $F^*(f@g) = F^*f@F^*g$, which is much different from Fokkinga's approach[5]. Fokkinga's assumption of F^* being a functor makes his theory not valid for several known monads such as state monads, I/O monads etc. Instead, we shift his assumption on the monad to the condition of a structure-preserving map (see Definition 10 below) so that our theory makes sense for arbitrary monad.

Example 1 (F_T^*) Consider the functor $F_T = !a + I \times I$ defined in Section 2.3, and we can calculate F_T^* as follows.

$$\begin{aligned} F_T^*\ f &= (!a)^*\ f + (I \times I)^*\ f \\ &= result + \lambda(x_1, x_2) \rightarrow \\ &\qquad I^*\ f\ x_1\ `bind`\ \lambda y_1 \rightarrow \\ &\qquad I^*\ f\ x_2\ `bind`\ \lambda y_2 \rightarrow \\ &\qquad result\ (y_1, y_2) \\ &= result + \lambda(x_1, x_2) \rightarrow \\ &\qquad f\ x_1\ `bind`\ \lambda y_1 \rightarrow \\ &\qquad f\ x_2\ `bind`\ \lambda y_2 \rightarrow \\ &\qquad result\ (y_1, y_2) \end{aligned}$$

□

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

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

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

□

Our definition of structure-preserving map has an additional condition (2), comparing with Fokkinga's. At the first glance, it seems that we impose more restrictions on such map than Fokkinga. On the contrary, we have few restriction, because Fokkinga's assumption actually requires that for any monadic functions f and g, $F^*f@F^*g = F^*(f@g)$ while our additional condition only requires that for a *specific* monadic function h and any monadic function g, $F^*h@F^*g = F^*(h@g)$ holds.

We remind readers again of that a structure-preserving map h from (X, ϕ) to (Y, ψ) is a monadic function with type $X \rightarrow M\ Y$ although it is a morphism from X to Y in category C^M. Now we are ready to construct the category of monadic F-algebras, denoted by $Alg(F, *)$, with respect to a monad M.

Theorem 2 (Category of Monadic F-algebras: $Alg(F, *)$) The category of monadic F-algebras, denoted by $Alg(F, *)$, is constructed as follows.

- 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 prove that the above construction establishes a category, we have to show that

(a) The *result* is a morphism, i.e., a structure-preserving map;
(b) Given two morphisms $h_1 : (X, \phi) \rightarrow (Y, \psi)$ and $h_2 : (Y, \psi) \rightarrow (Z, \eta)$, then $h_2@h_1 : (X, \phi) \rightarrow (Z, \eta)$;

(c) The composition @ is associative;

(d) $result@h = h@result = h$.

Since our category is built upon C^M, (c) and (d) are obviously right.

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 $h_2@h_1$ satisfies two conditions for being a morphism.

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

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

□

5.4 Monadic Catamorphisms

Motivated by many studies on direct construction of monadic catamorphisms [15, 17], we shall give the definition of monadic catamorphisms and corresponding promotion theorem. Moreover, we show that the category $Alg(F, *)$ can be extended to $Alg^+(F, *)$ including monadic catamorphisms as morphisms.

Our definition of monadic catamorphisms is based on the following fact.

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

$$h@(result \circ in) = \phi@F^*h$$

Here h will be called *monadic catamorphism* and will be denoted by $\langle\!|\, \phi \,|\!\rangle_{F^*}$.

Proof: If we can define h explicitly in terms of F, *in* and/or ϕ, we can say that h exists and is uniquely determined by F, *in* and ϕ, and hence the proposition is proved. To this

end, we calculate the equation and find the explicit definition of h as follows.

$$
\begin{array}{ll}
& h@(result \circ in) = \phi@F^*h \\
\equiv & \quad \{\ result \text{ property } \} \\
& h \circ in = \phi@F^*h \\
\equiv & \quad \{\ \text{Separable property of } F^*\ \} \\
& h \circ in = \phi@(d_F \circ F\ h) \\
\equiv & \quad \{\ \text{by the fact } f@(g \circ h) = (f@g) \circ h\ \} \\
& h \circ in = (\phi@d_F) \circ F\ h \\
\equiv & \quad \{\ \text{uniqueness of the normal catamorphism}\ \} \\
& h = (\![\ \phi@d_F]\!)
\end{array}
$$

□

Example 2 (Monadic catamorphism over tree) The *subst* given in Section 4 is a monadic catamorphism over tree data structure.

$$
\begin{array}{l}
subst = \langle\!|\ \phi_1, \phi_2 |\!\rangle_{F^*_T} \\
\qquad \phi_1\ a\ = lookup\ a\ `bind`\ \lambda b \to result(Leaf\ b) \\
\qquad \phi_2\ (l, r)\ =\ result\ (Node\ (l, r))
\end{array}
$$

If F^* is clear from the context, we may omit the subscript F^* in $\langle\!|\ - |\!\rangle_{F^*}$.

□

Our promotion theorem for manipulating monadic catamorphisms is as follows.

Theorem 4 (Monadic Promotion) If

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

Then

$$h@\langle\!|\ \phi |\!\rangle\ = \langle\!|\ \psi |\!\rangle$$

Proof:

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

□

Let's compare our promotion theorem with Fokkinga's [5] again. If the monad we deal with satisfies Fokkinga's assumption, our definition of F^* becomes a functor from C^M to C^M and so the second promotable condition in Theorem 4 can be removed since it is always true. Thus, Fokkinga's promotion theorem is derived. In one word, our theorem covers Fokkinga's. However, our theorem has no assumption on monads and is applicable to all monads including, for example, the state monad that is ruled out by Fokkinga's theory. The idea behind our method is to shift

Fokkinga's assumption to a condition of promoting a monadic function h to a monadic catamorphism.

Why is our promotion theorem able to be applied to a wider class of monadic functions than Fokkinga's? Recall that Fokkinga's assumption which for any monadic functions h and g, $F^*h@F^*g = F^*(h@g)$ must hold is one implicit necessary condition in his promotion theorem. We believe that the 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, we, as in our promotion theorem, only requires that for a *specific* monadic function h and any monadic function g, $F^*h@F^*g = F^*(h@g)$.

Example 3 (Promotional Transformation on st) Recall the definition of st in Section 4:

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

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

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

It is easy to verify that $F_T^*(f@g) = F_T^*f@F_T^*g$ for any monadic functions f and g. So the condition (2) in the Theorem 4 is always true. Therefore, we only need to find $\psi_1 \triangledown \psi_2$ satisfying the condition (1) in order to promote $result \circ sumtr$ into the monadic catamorphism $(\!|\ \phi_1 \triangledown \phi_2\ |\!)$. To this end, we do calculation as follows.

$$\begin{array}{ll} & (result \circ sumtr)@\phi_1 \\ = & \quad \{\ \text{def. of } \phi_1\ \} \\ & (result \circ sumtr)@(\lambda a \to lookup\ a\ `bind`\ \lambda b \to result\ (Leaf\ b)) \\ = & \quad \{\ \text{def. of } @\ \} \\ & (result \circ sumtr)@(result \circ Leaf)@lookup \\ = & \quad \{\ \text{associativity of } @, \text{ unity of } result\ \} \\ & (result \circ sumtr \circ Leaf)@lookup \\ = & \quad \{\ \text{def. of } sumtr\ \} \\ & result@lookup \\ = & \quad \{\ F_1 =!a, F_1^*f = result\ \} \\ & result@lookup@F_1^*(result \circ sumtr) \end{array}$$

And

$$\begin{array}{ll} & (result \circ sumtr)@\phi_2 \\ = & \quad \{\ \text{def. of } \phi_2\ \} \\ & (result \circ sumtr)@\lambda(l,r) \to result\ (Node\ (l,r)) \\ = & \quad \{\ \text{calculation}\ \} \\ & (result \circ sumtr)@(result \circ Node) \\ = & \quad \{\ \text{unity of } result\ \} \\ & result \circ sumtr \circ Node \\ = & \quad \{\ \text{def. of } sumtr, F_2 = I \times I\ \} \\ & result \circ + \circ F_2(sumtr) \\ = & \quad \{\ \text{def. of } F_2^*\ \} \\ & (result \circ +)@F_2^*(result \circ sumtr) \end{array}$$

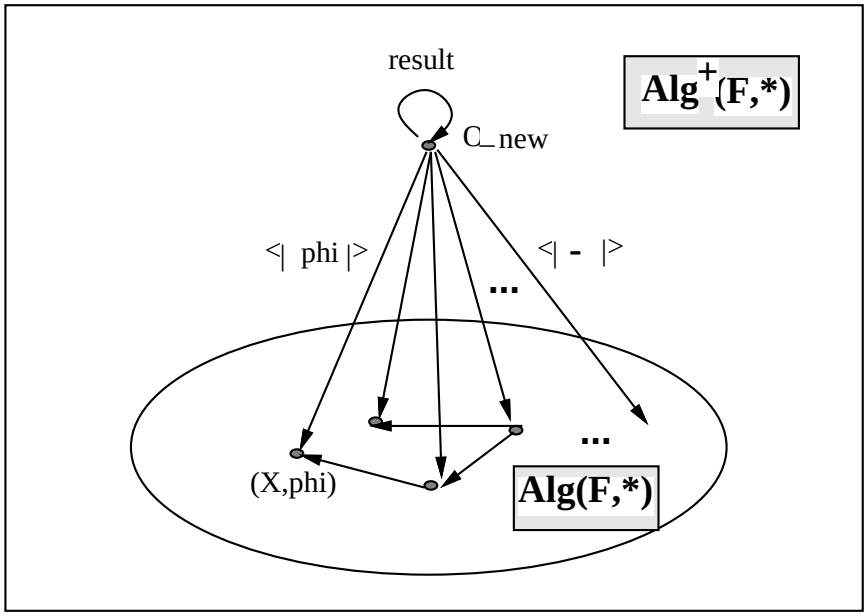

Fig. 1 The extension of $Alg(F,*)$ to $Alg^{+}(F,*)$

According to Theorem 4, we get an efficient program for st as follows.

$$\begin{aligned} st &= \langle\!\![\, \psi_1, \psi_2]\!\!\rangle \\ \psi_1 &= result@lookup \\ \psi_2 &= result \circ +. \end{aligned}$$

This is a simple example intended to show how our promotion theorem may be used rather than arguing the benefit of our theorem. Fokkinga's theorem is also applicable to this example. The example that cannot handled by Fokkinga's can be found in Section 6.

□

Now we turn to see what is the monadic catamorphism in the framework of our general theory $Alg(F,*)$ which has as morphisms structural monadic functions (i.e. structure-preserving map). We would like to show that monadic catamorphisms can be added to $Alg(F,*)$ to form a new category $Alg^{+}(F,*)$.

Theorem 5 ($Alg^{+}(F,*)$) The category of $Alg^{+}(F,*)$ is an extension of $Alg(F,*)$ in the following way, as in Fig 1.

- It has as objects the objects in $Alg(F,*)$, together with a newly added object $O_{new} = (T, result \circ in)$.
- It has as morphisms the morphisms in $Alg(F,*)$, together with the morphism $\langle\!\![\, \phi]\!\!\rangle : O_{new} \to (X, \phi)$ for any object (X, ϕ) in $Alg(F,*)$ and the identity morphism $result : O_{new} \to O_{new}$.
- It has the same composition operator as $Alg(F,*)$.

- It has the same identity morphism as $Alg(F, *)$.

Proof: To show that such extension really establishes a category, we have to prove two facts.

(1) Since $Alg^+(F, *)$ is required to have the same composition operator and the same identity morphism as $Alg(F, *)$, we have to show that the newly introduced object and morphisms are compatible with those in $Alg(F, *)$, i.e., the new object should be a monadic F-algebra and the new morphisms should be structure-preserving maps between monadic F algebras.

(2) Composition is closed in $Alg^+(F, *)$, i.e., given any two morphisms $h_1 : (X, \phi) \to (Y, \psi)$ and $h_2 : (Y, \psi) \to (Z, \eta)$, then there must exist a morphism $h : (X, \phi) \to (Z, \eta)$.

If we can prove (1) and (2), we can say that $Alg^+(F, *)$ is an extended category of $Alg(F, *)$.

We prove (1) first. It is easy to verify that O_{new} is a monadic F-algebra and that $result$ is a morphism from O_{new} to O_{new}. Now we prove that for any object (X, ϕ) in $Alg(F, *)$, $(\!|\phi|\!)$ is a morphism from O_{new} to (X, ϕ). Equivalently, we prove that $(\!|\phi|\!)$ is a structure-preserving map satisfying two conditions in Definition 10. The first condition is obviously satisfied from Proposition 3. For the second condition:

$$F^*(\!|\phi|\!) \mathbin{@} F^* g = F^*((\!|\phi|\!) \mathbin{@} g) \quad \text{for any morphism } g,$$

since the only possibility of morphism g is the identity morphism (i.e., $result$) in $Alg^+(F, *)$, we can replace g with $result$ in the equation and verify that this condition is satisfied.

For (2), since $Alg(F, *)$ is a category, it is enough to prove that the composition always exists of two morphisms one of which is a newly introduced one. This is true because there exists a morphism O_{new} to every object of $Alg^+(F, *)$.

□

Obviously, O_{new} is an initial object in $Alg^+(F, *)$. We thus obtain, in another way, our general promotion rule (Theorem 4) for monadic catamorphisms. It is worth nothing that in $Alg(F, *)$ we also have an object $(T, result \circ in)$. However whether this object is an initial object of $Alg(F, *)$ or not depends on the monad over which $Alg(F, *)$ is defined. If $Alg(F, *)$ is defined over the state monad, this object is not an initial object since there exist some objects to which no morphism is from $(T, result \circ in)$. To make our theory valid for arbitrary monads, we extend $Alg(F, *)$ introducing the object O_{new}. Although the two objects are the same, the morphisms starting from this object and the morphisms starting from O_{new} are different. It is the latter morphisms (i.e., monadic catamorphisms) that we have more interest in during program transformation.

6 An Example

In this section, we adopt the impressive example, provided by Meijer and Jeuring[15], that calculates efficient G-machine. We intend to show that the four promotable conditions, informally provided in [15] and acted as the kernel strategy in their calculation, are just one instance of our promotion theorem. We will not address the detail calculation already done in [15] and only explain our main idea.

6.1 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 pointing to a node of the graph. An naive G-machine evaluates an expression e by first building a graph for expression e and then evaluating resulting graph:

$$\begin{aligned} &machine &&:: [Pointer] \to Expr \to State\ Graph\ Pointer \\ &machine\ ps\ e &&= (eval@(build\ ps))\ e \end{aligned}$$

Evaluating an expression thus is rather inefficient: function *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 while our theory can.

In fact, Meijer and Jeuring [15] has shown that *build ps* can be described as a monadic catamorphism over $Expr$, but they defined it undirectly in terms of a normal catamorphism in order that they can find transformation rules to promote *eval* into *build ps* to obtain efficient monadic program.

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.

6.2 Expressing *build ps* by a Monadic Catamorphism over $Expr$

Consider a tiny first order language[15] 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 $Expr$:

$$\begin{aligned} Expr = {} & Var\ Int && \text{variable} \\ \mid {} & Con\ Int && \text{constant} \\ \mid {} & Add\ (Expr, Expr) && \text{Addition} \\ \mid {} & App\ (Name, Expr) && \text{Function application} \end{aligned}$$

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

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

Now we can have a derivation of F:

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

According to Proposition 3, a monadic catamorphism over $Expr$ is $\langle\!|\ \phi\ |\!\rangle$, defined

by

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

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 monadic catamorphism[15]. We omit the detail but assume that

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

6.3 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* tp 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 $eval@(build\ ps) = (\!|\, \psi_1 \triangledown \psi_2 \triangledown \psi_3 \triangledown \psi_4 \,|\!)$, provided that

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

which is essentially the same as the four conditions informally provided in [15]. Note that we have (c_1) and (c_2) while there is only one condition in [15]. It is easy to see that (c_1) and (c_2) imply Meijer and Jeuring's one condition.

Using the above rule, we can go on deriving ψ_i's and thus obtain an efficient monadic program for G-machine as done in [15].

7 Discussions

As noted in Section 5.3, our given F^* is just one possible definition. Therefore, for a given type, we have only one definition of endofunctor F but we may have a class of F^* and hence have a class of monadic catamorphisms. This is quite different from normal catamorphisms. But once F^* is determined, the corresponding monadic catamorphism and corresponding promotion rules are determined.

So far, we have only studied the transformation of monadic programs with respect to one monad. How to fuse monadic programs with respect to different monads?

Can we extend our work on monadic catamorphisms to its dual, say monadic anamorphism?

Acknowledgement

We wish to thank A. Takano for his helpful discussions and numerous suggestions on this paper.

References

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

[2] M. Barr and C. Wells, editors. *Category Theory for Computing Science.* Prentice Hall, 1990.

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

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

[5] M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical report, Dept. INF, University of Twente, Netherlands, June 1994.

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

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

[8] M.P. Jones and L. Duponcheel. Composing monads. Report yaleu/dcs/rr-1004, Department of Computer Science, Yale University, December 1993.

[9] 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.

[10] D.J. King and P. Wadler. Combining monad. In *Proc. Glasgow Workshop on Functional Programming.* Springer-Verlag, 1993.

[11] S. Liang, P. Hudak, and M.P. Jones. Monad transformers and modular interpreters. In *Proc. 22th ACM symposium on principles of programming languages*, San Francisco, 1995.

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

[13] 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.

[14] E. Meijer and G. Hutton. Bananas in space: Exteding fold and unfold to exponential types. In *FPCA '95*, June 1995.

[15] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *Proc. of 1st International Springschool on Advanced Functional Progamming Techniques*, May 1995.

[16] E. Moggi. An abstract view of programming languages. Technical Report Technique Report ECS-LSCS-90-113, LFCS, University of Edinburgh, Edingurgh, Scotland, 1990.

[17] T. Sheard. Type parametric programming with compile-time reflection. Technical Report OGI, Tech-report, Dept. of Computer Science and Engineering, Oregon Graduate Institution of Science and Technology, June 1993.

[18] 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.

[19] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *FPCA '95*, June 1995.

[20] P. Wadler. Comprehending monads. In *Conference on Lisp and Fuctional Programming*, 1990.

[21] P. Wadler. The essence of functional programming. In *19'th Annual Symposium on Principles of Programming Languages*, 1992.