Fusion of Monadic (Co)Recursive Programs*

Alberto Pardo†
Instituto de Computación
Universidad de la República
Montevideo, Uruguay
pardo@fing.edu.uy

1 Introduction

Generic recursive functionals on data types —such as fold (catamorphism), unfold (anamorphism), or primitive recursion, among others— have been typically used as a tool for structuring 'pure' functional programs. A key feature of these standard functionals is the fact that they can be uniformly derived from data type definitions by using the categorical interpretation of recursive types. Using categorical properties, it is also possible to state general algebraic laws for the recursive functionals that hold for a whole class of data types and are useful in derivations, program transformations and general reasoning (see e.g. [10, 3, 8, 11]).

Some of these general laws are the so-called *fusion laws*, which essentially help to eliminate intermediate data structures that arise in function compositions. (This is sometimes referred to as *deforestation*.) In functional programming, the interest in this kind of laws is mainly motivated by the wide use of the popular design technique by which complex functions are built up by gluing together simpler ones using function composition. In effect, gluing turns out to be a good device for program modularization, but sometimes inadequate at execution time. This is because it may lead to time and space inefficiencies caused by the generation (i.e. allocation) and immediate consumption (i.e. processing and de-allocation) of intermediate data in each function composition. Recent works [20, 16] have shown that fusion laws are specially suitable for deforestation purposes mainly when functions are represented in terms of so-called *hylomorphisms*. A hylomorphism is equivalent to the composition of an anamorphism followed by a catamorphism, but with the virtue of not generating the intermediate data structure produced in such a composition.

In the last years it has become well-established that, using *monads* [21], functional programs can also be structured by the *effects* they produce (or mimic to produce). Monads permit to capture in an unified framework a wide variety of computational effects occurring in programs, such as side-effects, exceptions, non-determinism, continuations or Input/Output. The growing use of monads in functional programming has had a considerable impact in the pragmatics of writing functional programs as well as in language design (see e.g. [19, 7]).

The introduction of monadic effects within programs adds a new dimension that needs to be considered when analyzing programs for program transformation, mainly when they involve recursion. In fact, we may find situations in which intermediate data structures are generated by monadic recursive processes and are impossible to be systematically eliminated by the existing

*Work presented at MPC'98 Workshop on Generic Programming, Marstrand, Sweden, June 1998. This work was developed while at Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany.

†Web-page: http://www.fing.edu.uy/~pardo/

deforestation techniques for 'pure' programs. In this sense, some recent works [4, 5, 13] have focused on the study of the combination between fold computations and monads, introducing a functional called *monadic catamorphism*.

The starting point of our research has been the study of the dual case, i.e. the combination of corecursion with monads. We refer to the arising notion as *monadic corecursion*, corresponding to functions that generate a data structure in a "corecursive manner" while producing some effect represented by a monad. That is, similarly to standard corecursion, the body of this class of functions is dictated by the structure of the values they produce. Function definitions of this kind are captured by a new functional called *monadic anamorphism*.

Monadic anamorphisms are functions of type $A \to MD$, where D is the type of the data structure being generated and M the monad. To see a simple example, consider a Haskell function `getLine` that reads a line from standard input:

```
getLine :: IO String
getLine  = getChar >>= \ c ->
           if c = '\n'
              then return []
              else getLine >>= \ s -> return (c : s)
```

where `getChar :: IO Char` is the function that reads a character from standard input and returns it. Recall that

```
type IO a = World -> (a,World)
return :: a -> IO a
>>=    :: IO a -> (a -> IO b) -> IO b
```

is the definition of the I/O monad in Haskell [2]. Clearly, the body of function `getLine` is guided by the list it constructs, while the side-effect is given by the actions produced on the standard input.

Going further, we have investigated the introduction of a notion of *monadic hylomorphism* as well. This new functional corresponds to the composition of a monadic anamorphism followed by (the lifting of) a 'pure' catamorphism.

$$\mathsf{mhylo} \;=\; B \xrightarrow{\mathsf{mana}} MD \xrightarrow{M\,\mathsf{cata}} MA$$

Similarly to hylomorphism, the virtue of monadic hylomorphism is the fact of expressing this composition as a single function, avoiding therefore the generation of the intermediate data structure. The relevance in this case is given by the fact that the elimination of the intermediate data structure is performed in the presence of a monad.

For example, a monadic hylomorphism is what we get by composing `getLine` with a function given by `foldr f b` (i.e. a catamorphism on lists) that performs some calculation with the the line read. That is,

```
h = getLine >>= \ xs -> return (foldr f b xs)
```

where we have used the fact that M`(f) = \m -> m >>= \x -> return (f x)`. The factorization property of monadic hylomorphism tells us that this composition corresponds to the definition:

```
h :: IO String
h  = getChar >>= \ c ->
     if c = '\n'
        then return b
        else h >>= \ y -> return (f c y)
```

In other words, although function `h` is logically organized by the signature of the intermediate list representing the line read, the list itself is actually never generated.

In the remainder of this paper we present the definitions and some fusion laws corresponding to monadic anamorphism and monadic hylomorphism. We also give a brief account of some non-trivial applications that can be represented in terms of these functionals. (A detailed treatment of the topics presented in this paper can be found in [17].)

2 Preliminaries

2.1 Data type Theory

The category-theoretic explanation of recursive types is based on the idea that types constitute objects of a category $\mathcal{C}$, and type constructors are functors on $\mathcal{C}$. Throughout we shall assume that $\mathcal{C}$ is the category **Cpo** whose objects are (pointed) *cpos* —i.e. complete partial orders possessing a least element $\perp$— and whose morphisms are continuous functions. The final object of $\mathcal{C}$ is denoted as 1 and is given by the singleton set $\{\perp\}$.

A recursive type corresponds to a solution to a equation $X \cong FX$, i.e. a fixpoint of F, where $F : \mathcal{C} \to \mathcal{C}$ is an endofunctor that captures the recursive shape (or signature) of the type. The *canonical solution* to an equation $X \cong FX$ is given by a cpo μF together with an isomorphism $in_F : F\mu F \to \mu F$ that encodes the *constructors* of the datatype, its inverse $out_F : \mu F \to F\mu F$ gives the *destructors*.

We assume that type signatures are given by *regular functors*, which are described next. We first introduce what we consider basic functors. $I : \mathcal{C} \to \mathcal{C}$ stands for the identity functor. For each object $A \in \mathcal{C}$, $\underline{A}$ denotes the constant functor $X \mapsto A$. $\Pi_1, \Pi_2 : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ are the projection bifunctors. The product bifunctor $\times$ is given by cartesian product. We write $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ to denote the (left and right) projections. The pairing of two arrows $f : C \to A$ and $g : C \to B$ is written $\langle f, g \rangle : C \to A \times B$. The sum bifunctor $+$ is given by *separated sum*. We write $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ for the sum inclusions. For $f : A \to C$ and $g : B \to C$, case analysis is defined as a (strict) function $[f, g] : A + B \to C$ satisfying $[f, g] \circ \mathsf{inl} = f$ and $[f, g] \circ \mathsf{inr} = g$.

The class of *regular functors* is inductively defined by the following grammar:

$$R ::= B \mid RR \mid R \dagger R \mid D$$

where B stands for the set of basic functors given above. Functor composition is just juxtaposition. For a bifunctor $\dagger$, $F \dagger G$ stands for the functor such that $A \mapsto FA \dagger GA$. (Typical cases are when $\dagger \in \{\times, +\}$.) D stands for a *type functor* and is introduced later on.

An arrow $h : FA \to A$ is usually called a F-*algebra*. For any strict algebra $h : FA \to A$, we can define a *catamorphism* $(\!|h|\!)_F : \mu F \to A$ as the unique (strict) function satisfying this equation:

$$(\!|h|\!)_F \circ in_F = h \circ F(\!|h|\!)_F$$

Catamorphism enjoys many laws for program transformation. A law that plays an important rôle is the so-called *cata-fusion*:

$$f \text{ strict } \wedge \ f \circ h = h' \circ Ff \ \Rightarrow \ f \circ (\!|h|\!)_F = (\!|h'|\!)_F$$

Dually, a F-*coalgebra* is an arrow $g : A \to FA$. The destructors $out_F : \mu F \to F\mu F$ turn out to form a *final coalgebra*. Finality means that, for any coalgebra g, there exists a unique function

$[\![g]\!]_F : A \to \mu F$, called *anamorphism*, that makes this diagram commute:

$$\begin{tikzcd} A \arrow[r, "{[\![g]\!]_F}"] \arrow[d, "g"'] & \mu F \arrow[d, "out_F"] \\ FA \arrow[r, "{F\,[\![g]\!]_F}"'] & F\mu F \end{tikzcd}$$

or equivalently, that makes this equation hold: $[\![g]\!]_F = in_F \circ F[\![g]\!]_F \circ g$. Anamorphism recursively builds up a data structure by decomposing its argument using coalgebra g. Along this paper we will refer to this recursion pattern as *corecursion*. Finality enables us to derive calculational laws for anamorphism which are dual to those of catamorphism. There is a corresponding *ana-fusion* law:

$$g \circ f = Ff \circ g' \;\Rightarrow\; [\![g']\!]_F \circ f = [\![g]\!]_F$$

For a bifunctor F, $F(A,-)$ —to be written F_A— denotes the functor $F_A\, B = F(A,B)$ and $F_A\, f = F(\mathsf{id}_A, f)$. F_A induces a parameterized datatype $DA = \mu F_A$ with constructors $in_{F_A} : F_A(DA) \to DA$, given by the solution to the type equation $X \cong F(A,X)$. D can be made into a functor $D : \mathcal{C} \to \mathcal{C}$ —called a *type functor*— by defining its action on arrows, $Df : DA \to DB$ for $f : A \to B$, equally by a catamorphism or an anamorphism.

$$Df = (\!| in_{F_B} \circ F(f, \mathsf{id}_{DB}) |\!)_{F_A} = [\![F(f, \mathsf{id}_{DA}) \circ out_{F_A}]\!]_{F_B}$$

The following fusion laws hold for type funtors: For $f : A \to B$,

$$(\!| h |\!)_{F_B} \circ Df = (\!| h \circ F(f, \mathsf{id}) |\!)_{F_A} \qquad\qquad Df \circ [\![g]\!]_{F_A} = [\![F(f, \mathsf{id}) \circ g]\!]_{F_B}$$

Example 2.1 The parameterized functor $L_A\, B = 1 + A \times B$ captures the signature of (finite and infinite) lists with elements over A. The list constructors are given by the algebra $[\mathsf{nil}, \mathsf{cons}] : 1 + A \times A^* \to A^*$, where $\mathsf{nil} : 1 \to A^*$ and $\mathsf{cons} : A \times A^* \to A^*$. For an algebra $h = [h_1, h_2] : 1 + A \times B \to B$, the catamorphism is the (strict) function $f = (\!| h |\!)_{L_A} : A^* \to B$ such that: $f(\mathsf{nil}) = h_1$ and $f(\mathsf{cons}(a, \ell)) = h_2(a, f(\ell))$. It corresponds to the usual foldr operator as it is known in functional programming [1]. The list type functor $\mathsf{List}(f) = (\!| [\mathsf{nil}, \mathsf{cons} \circ (f \times \mathsf{id})] |\!)_{L_A}$ corresponds to the usual map function on lists [1]: $\mathsf{List}(f)(\mathsf{nil}) = \mathsf{nil}$ and $\mathsf{List}(f)(\mathsf{cons}(a, \ell)) = \mathsf{cons}(f(a), \mathsf{List}(f)(\ell))$.

Suppose we are given an algebra $h : FA \to A$ and a coalgebra $g : B \to FB$. The *hylomorphism* [10] is the function $[\![h, g]\!]_F : B \to A$ defined as

$$[\![h, g]\!]_F = \mathsf{fix}(\lambda f.\, h \circ Ff \circ g)$$

That is, it is the least function satisfying the equation $[\![h, g]\!]_F = h \circ F\, [\![h, g]\!]_F \circ g$, which expresses the characteristic shape of recursion that comes with each functor. Several reasons make hylomorphism particularly interesting. One of them is the fact that catamorphism and anamorphism are special cases of it: $(\!| h |\!)_F = [\![h, out_F]\!]_F$ and $[\![g]\!]_F = [\![in_F, g]\!]_F$.

In addition, we have a factorization property which states that a hylomorphism corresponds to the composition of a catamorphism with an anamorphism on the same datatype,

$$[\![h, g]\!]_F \;=\; B \xrightarrow{[\![g]\!]_F} \mu F \xrightarrow{(\!| h |\!)_F} A$$

Hylomorphism turns out to be useful for the application of deforestation techniques [20, 16]. The basis of its success has been given by the introduction of two laws —usually referred to together as the *Acid Rain Theorem* [20]— which, under certain conditions, permit to fuse the composition of a hylomorphism with a catamorphism/anamorphism into another hylomorphism.

Theorem 2.2 (Acid Rain)

(i) *Hylo-Cata fusion:* For $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ polymorphic on A and h strict,

$$(\!|h|\!)_F \circ [\![\boldsymbol{T}(in_F), g]\!]_G = [\![\boldsymbol{T}(h), g]\!]_G$$

(ii) *Ana-Hylo fusion:* For $\boldsymbol{T} : (A \to FA) \to (A \to GA)$ polymorphic on A,

$$[\![h, \boldsymbol{T}(out_F)]\!]_G \circ [\!(g)\!]_F = [\![h, \boldsymbol{T}(g)]\!]_G$$

One of the aims of this paper is to achieve a monadic extension of the acid rain theorem so that we can apply deforestation to functions represented in terms of monadic hylomorphisms.

2.2 Monads

The use of monads has become very popular among the functional programming community. Monads permit to encapsulate in abstract terms several forms of computations, such as exceptions, state, I/O, continuations or nondeterminism [14, 21]. Computational effects are represented as a type constructor M together with two operations satisfying certain laws so that computations delivering values of type A are regarded as terms of type MA.

Formally, one of the definitions of monads corresponds to the so-called Kleisli triple. A *Kleisli triple* $(M, \mathsf{unit}, -^\star)$ over $\mathcal{C}$ is given by (the restriction to objects of) an endofunctor M, a natural transformation $\mathsf{unit}_A : A \to MA$, and an extension operator $-^\star$ which for each $f : A \to MB$ yields $f^\star : MA \to MB$, such that these equations hold: $\mathsf{unit}_A^\star = \mathsf{id}_{MA}$, $f^\star \circ \mathsf{unit}_A = f$, and $f^\star \circ g^\star = (f^\star \circ g)^\star$.

The extension operator gives a way to compose monadic functions, passing the effect around. For $f : A \to MB$ and $g : B \to MC$, the Kleisli composition is defined by

$$g \bullet f \stackrel{\text{def}}{=} g^\star \circ f$$

Now we can assign a meaning to the Kleisli triple laws. The first two laws amount to say that unit is a left and right identity with respect to Kleisli composition, whereas the last one expresses that composition is associative. In other words, the Kleisli triple laws just express that monadic morphisms $A \to MB$ form a category, called the *Kleisli category* $\mathcal{C}_M$, whose objects are those of $\mathcal{C}$. Identities are given by unit. We can then define for every monad a *lifting functor* $(\widehat{-}) : \mathcal{C} \to \mathcal{C}_M$ that is the identity on objects and whose action on arrows is such that $f \mapsto \mathsf{unit} \circ f$.

In the context of functional programming a monad is usually presented by a Kleisli triple $(M, \mathsf{unit}, \star)$ where: M is a type constructor; $\mathsf{unit} : A \to MA$ is a polymorphic function; and $\star : MA \times (A \to MB) \to MB$ is a polymorphic (infix) operator called *bind*. The Kleisli triple laws translate to corresponding equations in terms of unit and $\star$ (see [21]). An expression $m \star f$ corresponds to $f^\star(m)$. The infix notation turns out to be preferable for writing functional programs in monadic style, as it gives a graphical idea of the existing sequentiality in the execution of computations. The Kleisli star notation, on the other hand, is more suitable for performing formal manipulation. For this reason we will keep both notations for 'bind', using each one where it better suits.

3 Monadic Anamorphism

One way of approaching to monadic anamorphism is by dualizing the recursion scheme that characterizes *monadic catamorphism* as defined by Fokkinga and others [4, 5]. However, we opted to directly introduce this concept by means of an intuitive explanation of its behavior.

A first approximation to the notion of monadic anamorphism can be got by considering the usual corecursion scheme captured by the diagram:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & \mu G \\ GA \arrow[r, "Gf"'] & G\mu G \arrow[u, "in_G"'] \end{tikzcd}$$

but viewing it as being a diagram in the category $\mathcal{C}_M$ of monadic functions, for certain monad M. Proceeding that way we are thinking of each arrow as an effect-producing function, getting thus the somewhat 'imperative' idea of a corecursive process that produces some side-effect along its evaluation. Since $\mathcal{C}$ is our universe of discourse, we need to describe all components of this diagram (in $\mathcal{C}_M$) as elements of $\mathcal{C}$ in order to to get a real understanding of the program scheme. The first step is to make the computational effect explicit.

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & M\mu G \\ M(GA) \arrow[r, "(Gf)^\star"'] & M(G\mu G) \arrow[u, "in_G^\star"'] \end{tikzcd} \tag{1}$$

It rests to determine who play the rôle of G and in_G.

Since the objects of $\mathcal{C}$ and $\mathcal{C}_M$ coincide, the data structure generated by such a recursive definition necessarily corresponds to a datatype μF, for some functor F on $\mathcal{C}$. This implies that G corresponds to some *monadic extension* of F —to be denoted $\widehat{F}$— whose action on objects coincides with that of F, i.e. $\widehat{F}A = FA$, and when applied to a monadic function $f : A \to MB$, yields an arrow $\widehat{F}f : \widehat{F}A \to M(\widehat{F}A)$ —actually $\widehat{F}f : FA \to M(FA)$— whose action embeds that of Ff.

In the next subsection we introduce the definition of the monadic extension of a regular functor. (A discussion of its properties can be found in [17, 18].) Afterwards, in Subsection 3.2, we complete the definition of monadic anamorphism. Finally, in Subsection 3.3 we present some calculational laws.

3.1 The Monadic Extension of a Functor

Monadic extensions $\widehat{F} : \mathcal{C}_M \to \mathcal{C}_M$ are in bijection with natural transformations $\delta^F : FM \Rightarrow MF$ that perform the distribution of the monad over the functor. In fact, given a distribution law δ^F, we can define the action of $\widehat{F}$ on functions $f : A \to MB$ as follows:

$$\widehat{F}f \;=\; FA \xrightarrow{Ff} FMB \xrightarrow{\delta^F_B} MFB$$

Conversely, given a monadic extension $\widehat{F}$ a natural transformation δ^F can be defined by making use of the adjunction between $\mathcal{C}_M$ and $\mathcal{C}$ (see [15, 17]). A monadic extension $\widehat{F}$ is called a *lifting* of F in case it is a functor on $\mathcal{C}_M$.

For every regular functor F a natural transformation $\delta^F : FM \Rightarrow MF$ can be defined by induction on the structure of the functor.

$$\begin{array}{llll}
\delta^{\underline{C}}_A & = \mathsf{unit}_C & \delta^{+}_{(A,B)} & = [M\mathsf{inl}, M\mathsf{inr}] \\
\delta^{I}_A & = \mathsf{id}_{MA} & \delta^{FG}_A & = \delta^F_{GA} \circ F\delta^G_A \\
\delta^{\Pi_i}_{(A_1,A_2)} & = \mathsf{id}_{MA_i} & \delta^{F\dagger G}_A & = \delta^{\dagger} \circ (\delta^F_A \dagger \delta^G_A) \\
\delta^{\times}_{(A,B)} & = \psi_{A,B} & \delta^{D}_A & = (\!| Min_{F_A} \circ \delta^F_{(A,DA)} |\!)_{FMA}
\end{array}$$

In the last line F is the bifunctor that induces D. The definition of δ^D is a form of monadic catamorphism. Function $\psi_{A,B} : MA{\times}MB \to M(A{\times}B)$ corresponds to the left-to-right distribution of the monad over the product. It is given by

$$\psi(m, m') = m \star \lambda a.\, m' \star \lambda b.\, \mathsf{unit}(a, b)$$

A right-to-left distribution $\psi'_{A,B}$ can be defined analogously. These product distributions can be defined for any monad that happens to be *strong* [14].

Now, we can deduce the monadic extension for every regular functor:

$$\begin{array}{llll}
\widehat{\underline{C}} f & = \mathsf{unit}_C & f \mathbin{\widehat{+}} g & = [M\mathsf{inl} \circ f, M\mathsf{inr} \circ g] \\
\widehat{I} f & = f & \widehat{FG} f & = \widehat{F}\widehat{G} f \\
\widehat{\Pi_i}(f_1, f_2) & = f_i & (\widehat{F \dagger G}) f & = (\widehat{F} f) \mathbin{\widehat{\dagger}} (\widehat{G} f) \\
f \mathbin{\widehat{\times}} g & = \psi \circ (f \times g) & \widehat{D} f & = (\!| M in_{F_B} \circ \widehat{F}(f, \mathsf{id}) |\!)_{F_A}
\end{array}$$

In particular,

$$(\widehat{F \times G}) f = \psi \circ (\widehat{F} f \times \widehat{G} f) \qquad\qquad (\widehat{F + G}) f = [M\mathsf{inl} \circ \widehat{F} f, M\mathsf{inr} \circ \widehat{G} f]$$

$\widehat{D} f$ is usually called a *monadic map* [4].

Example 3.1 (i) For the functor $L_A = \underline{1} + \underline{A} \times I$, we have:

$$\widehat{L_A} f = [\widehat{\mathsf{inl}}, \widehat{\mathsf{inr}} \bullet \tau \circ (\mathsf{id} \times f)]$$

(ii) Consider the functor $R_A = \underline{A} \times (-)^*$ that captures the signature of rose trees (multiway branching trees). Then,

$$\widehat{R_A} f = \psi \circ (\mathsf{unit} \times \widehat{\mathsf{List}}(f)) \qquad \text{with} \qquad \widehat{\mathsf{List}}(f) = (\!| [\widehat{\mathsf{nil}}, \widehat{\mathsf{cons}} \bullet \psi \circ (f \times \mathsf{id})] |\!)$$

3.2 The Definition of Monadic Anamorphism

Recall the diagram 1 shown earlier in this section, but now using the fact that G is given by $\widehat{F}$ for some F. Following type considerations, it is possible to see that $\widehat{in_F}$ —the lifting of the constructors in_F— is the natural candidate to play the rôle of in_G.

Therefore, in summary, a *monadic anamorphism*, denoted by $[\!(g)\!]^M_F$, is defined as the least function f satisfying the following diagram in $\mathcal{C}$:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & M(\mu F) \\
M(FA) \arrow[r, "(\widehat{F} f)^\star"'] & M(F \mu F) \arrow[u, "\widehat{in_F}^\star"']
\end{tikzcd}$$

That is,

$$[\!(g)\!]^M_F = \mathsf{fix}(\lambda f. \widehat{in_F} \bullet \widehat{F} f \bullet g)$$

Example 3.2 Consider the functor $L_A B = 1 + A \times B$ that captures the signature of lists. For $g : B \to M(1 + A \times B)$ the monadic anamorphism $[\!(g)\!]^M_{L_A} : B \to MA^*$ is given by:

$$[\!(g)\!]^M_{L_A} = \mathsf{fix}(\lambda f.\, [\widehat{\mathsf{nil}}, \widehat{\mathsf{cons}}^\star \circ \psi \circ (\mathsf{unit} \times f)]^\star \circ g)$$

Inlining:

$$
\llbracket g \rrbracket^M(b) = g(b) \star \lambda x.\, \textbf{case } x \textbf{ of} \quad \begin{array}{lcl} \mathsf{inl}(\bot) & \to & \mathsf{unit}(\mathsf{nil}) \\ \mathsf{inr}(a, b') & \to & \llbracket g \rrbracket^M(b') \star \lambda \ell.\, \mathsf{unit}(\mathsf{cons}(a, \ell)) \end{array}
$$

Example 3.3 (Polytypic zip) Function $\mathsf{zip} : A^* \times B^* \to (A \times B)^*$ is well-known in functional programming [1], it takes a pair of lists and returns a list of pairs of elements at corresponding positions. Jeuring and Jansson [6] give a polytypic version of zip, called $\mathsf{pzip} : DA \times DB \to M(D(A \times B))$, that zips two terms of type DA and DB, respectively, where D is induced by a bifunctor F and $MA = A + 1$ is the maybe monad.

The presence of the maybe monad is due to the necessity to control that the two terms being zipped have the same shape. Function pzip can be given by a monadic anamorphism

$$
\mathsf{pzip} = \llbracket (\mathsf{pzstep}) \rrbracket^M_{F_{A \times B}}
$$

for $\mathsf{pzstep} = \mathsf{fzip} \circ (out_{F_A} \times out_{F_B})$ and where $\mathsf{fzip} : F(A, X) \times F(B, Y) \to M(F_{A \times B}(X \times Y))$ is certain natural transformation whose definition is given by induction on the structure of bifunctor F (see [6, 17]).

Example 3.4 (Graph Traversals) *Graph traversals* are functions that take a list of roots (entry points to a graph) and return a list containing the vertices met along the way. Let us assume that a graph is given by an *adjacency list function* $\mathsf{adj} : V \to V^*$, which for each vertex $v \in V$ returns its adjacency list.

In a graph traversal vertices are visited at most once. This leads to maintain a set where to keep track of vertices already visited in order to avoid repeats. Consider an abstract data type $\mathcal{P}_{\mathrm{f}}(V)$ of finite sets over V, with operations $\emptyset : \mathcal{P}_{\mathrm{f}}(V)$ (the emptyset constant), $\uplus : V \times \mathcal{P}_{\mathrm{f}}(V) \to \mathcal{P}_{\mathrm{f}}(V)$ (the insertion of an element in a set), and $\lhd : V \times \mathcal{P}_{\mathrm{f}}(V) \to \mathsf{Bool}$ (a membership predicate). These operations are axiomatized by:

$$
v \lhd \emptyset = \mathsf{false} \qquad\qquad v \lhd (v' \uplus s) = \begin{cases} \mathsf{true} & \text{if } v = v' \\ v \lhd s & \text{otherwise} \end{cases}
$$

We construct a formulation of graph traversal in terms of a monadic anamorphism such that the set of visited nodes is manipulated by a state monad. This makes it possible to consider an "imperative" representation for sets. For instance, if V is finite and its elements can be ordered according to some total order, then we can represent a set by a *characteristic vector* of boolean values, permitting $O(1)$ time insertions and lookups when implemented by a *mutable array*. In that case the set operations shown above represent primitives that operate over the array directly. A well-established technique for handling these imperative operations in a functional setting is to encapsulate them in a monadic ADT based on the state monad [21]:

$$
MA = [\mathcal{P}_{\mathrm{f}}(V) \to A \times \mathcal{P}_{\mathrm{f}}(V)]
$$

which in addition to unit and $\star$ possesses the operations $\emptyset^M : MA \to A$, $\uplus^M : V \to M1$ and $\lhd^M : V \to M\,\mathsf{Bool}$ given by

$$
\emptyset^M(m) = \pi_1\,(m(\emptyset)) \qquad \uplus^M(v) = \lambda s.\, (\bot, v \uplus s) \qquad \lhd^M(v) = \lambda s.\, (v \lhd s, s)
$$

To guarantee safe, in-place update the monadic ADT operations need to manipulate the set in a *single-threaded* manner, i.e they must not duplicate it (see [21]). For this purpose it is necessary to add a strictness requirement: both $\uplus^M$ and $\lhd^M$ need to be strict in the input vertex and the set (but not in the values stored in it).

Based on this ADT, we can define the monadic formulation of graph traversal as a function $\mathsf{graphtrav} : V^* \to V^*$ given by

$$\mathsf{graphtrav}(vs) = \emptyset^M(\mathsf{gtrav}(vs))$$

where $\mathsf{gtrav} : V^* \to MV^*$ is a monadic anamorphism:

$$\mathsf{gtrav} = [\![\mathsf{gopen}]\!]^M_{L_V} \qquad \text{with} \qquad \mathsf{gopen} : V^* \to M(1 + V \times V^*)$$

In each iteration, the action of gopen begins with an exploration of the current list of roots vs in order to find a vertex that had not been reached before. To this end, it removes from the front of vs all vertices u that qualify as visited until either an unvisited vertex is met or the end of the list is reached. This task is performed by a function $\mathsf{mdropS} : V^* \to MV^*$ (see [17, 18]). Once mdropS was applied, we proceed to visit the vertex at the head of the input list (if any) and to mark it (inserting it in the set). At the same time, a new 'state' of the list of roots is also computed. This is performed by a function, called $policy : V \times V^* \to V^*$, which encapsulates the policy used for managing the list of pending roots. In this form, we can achieve a formulation parametric in the strategy followed by the traversal. In summary,

$$\begin{array}{rcl} \mathsf{gopen}(\ell) & = & \mathsf{mdropS}(\ell) \star \lambda\ell'.\ \mathbf{case}\ \ell'\ \mathbf{of} \\ & & \qquad \begin{array}{lcl} \mathsf{nil} & \to & \mathsf{unit}(\mathsf{inl}(\bot)) \\ \mathsf{cons}(v, vs) & \to & \uplus^M(v) \star \lambda x. \\ & & \mathsf{unit}(\mathsf{inr}(v, policy(v, vs))) \end{array} \end{array}$$

Therefore,

$$\begin{array}{rcl} \mathsf{gtrav}(\ell) & = & \mathsf{mdropS}(\ell) \star \lambda\ell'.\ \mathbf{case}\ \ell'\ \mathbf{of} \\ & & \qquad \begin{array}{lcl} \mathsf{nil} & \to & \mathsf{unit}(\mathsf{nil}) \\ \mathsf{cons}(v, vs) & \to & \uplus^M(v) \star \lambda x. \\ & & \mathsf{gtrav}(policy(v, vs)) \star \lambda ys. \\ & & \mathsf{unit}(\mathsf{cons}(v, ys)) \end{array} \end{array}$$

Now, let us consider particular traversal strategies. For example, an efficient way to implement a depth-first traversal is adopting a LIFO (Last-In First-Out) policy by holding in a stack the roots to visit next. Thus,

$$policy(v, vs) = \mathsf{adj}(v) +\!\!+\ vs$$

where $+\!\!+$ denotes list concatenation.

In a breadth-first traversal one visits all roots at a current depth from left to right before moving on to the next depth. This is achieved by adopting a FIFO (First-In First-Out) policy, managing the list of pending roots as a queue.

$$policy(v, vs) = vs +\!\!+\ \mathsf{adj}(v)$$

Example 3.5 (Monadic Parsers) A more elaborated example of monadic anamorphism is the syntax analysis phase of a monadic parser. Recall that, for the parser monad [21, 12], parsers are functions of type

$$MA = String \to (A \times String)^*$$

A syntax analyzer is a function $\mathsf{analyzer} : MT$, where T is the data type corresponding to the *concrete syntax* of the language under consideration. (In other words, T corresponds to the type of parse trees.) The structure of the parsing process is guided by the construction of parse trees. Moreover, the syntax analyzer of a language is given by a set of mutually recursive functions, one for each syntax category and for each terminal symbol. It is possible to see that each of these component functions correspond to a monadic anamorphism. To achieve this, we use a trick consisting of regarding each of them as a function from the unit type, i.e. as a function of type $1 \to MT$ (see [17, 18]).

Further examples can be found in [17, 18].

3.3 Calculational Laws

The first law we present is called *pure fusion*.

$$g \circ f = M(Ff) \circ g' \quad \Rightarrow \quad [\![g]\!]_F^M \circ f = [\![g']\!]_F^M$$

In case $\widehat{F}$ is a lifting or a semi-functor (i.e. when only preserves compositions), we can state a *mana-fusion law*.

$$g \bullet f = \widehat{F} f \bullet g' \quad \Rightarrow \quad [\![g]\!]_F^M \bullet f = [\![g']\!]_F^M$$

The following law is called *mana-map-fusion*. It states that a monadic anamorphism composed with the lifting of a type functor is again a monadic anamorphism. Suppose that M is locally continuous. Then, for $f : A \to B$,

$$M(Df) \circ [\![g]\!]_{F_A}^M = [\![M(F(f, \mathsf{id})) \circ g]\!]_{F_B}^M$$

Finally, we present an acid rain theorem for monadic anamorphism.

Theorem 3.6 (Acid Rain: Ana-mana fusion) For $\boldsymbol{T} : (A \to FA) \to (A \to MGA)$ polymorphic on A,

$$[\![\boldsymbol{T}(out_F)]\!]_G^M \circ [\![g]\!]_F = [\![\boldsymbol{T}(g)]\!]_G^M$$

Applications of these laws can be found in [17].

4 Monadic Hylomorphism

Having developed a monadic extension of anamorphism, it seems natural to investigate the notion of monadic hylomorphism too. For the sake of simplicity, we will only present a restricted but common form of monadic hylomorphism. (A more complete treatment can be found in [17].)

Given $g : B \to M(FB)$ and $h : FA \to A$ we define the *monadic hylomorphism* as the function $\langle\!| h, g |\!\rangle_F : B \to MA$ given by

$$\langle\!| h, g |\!\rangle_F = \mathsf{fix}(\lambda f. \widehat{h} \bullet \widehat{F} f \bullet g)$$

Example 4.1 Consider the functor $L_A B = 1 + A \times B$. For $g : B \to M(L_A B)$ and $h = [h_N, h_C] : L_A C \to C$ the monadic hylomorphism $\langle\!| h, g |\!\rangle_{L_A} : B \to MC$ is given by:

$$\langle\!| h, g |\!\rangle_{L_A} = \mathsf{fix}(\lambda f.\, [\widehat{h_N}, \widehat{h_C}^{\star} \circ \psi \circ (\mathsf{unit} \times f)]^{\star} \circ g)$$

Inlining:

$$\begin{aligned} \langle\!| h, g |\!\rangle(b) = g(b) \star \lambda x.\, & \textbf{case } x \textbf{ of} \\ & \mathsf{inl}(\bot) \quad\ \to \mathsf{unit}(h_N) \\ & \mathsf{inr}(a, b') \to \langle\!| h, g |\!\rangle(b') \star \lambda c.\, \mathsf{unit}(h_C(a, c)) \end{aligned}$$

Let us now discuss some of the properties of monadic hylomorphism. First of all, we can observe that monadic anamorphism is a particular case of monadic hylomorphism: $[\![g]\!]_F^M = \langle\!|in_F, g|\!\rangle_F$. On the other hand, $(\!|\widehat{h} \bullet \delta_A^F|\!)_F = \langle\!|h, \widehat{out_F}|\!\rangle_F$. However, in general, $(\!|\widehat{h} \bullet \delta_A^F|\!)_F \neq \widehat{(\!|h|\!)_F}$ and thus $\langle\!|in_F, \widehat{out_F}|\!\rangle_F = [\![\widehat{out_F}]\!]_F^M \neq \mathsf{unit}$. They become equalities only when unit is strict.

Recall that a (plain) hylomorphism can be factorized into the composition of an anamorphism followed by a catamorphism. Similarly, we get that a monadic hylomorphism corresponds to the composition of a monadic anamorphism followed by (the lifting of) a catamorphism.

$$\langle\!|h, g|\!\rangle_F \;=\; B \xrightarrow{[\![g]\!]_F^M} M\mu F \xrightarrow{M(\!|h|\!)_F} MA$$

Thus, we can transform any composition of this kind into a single function that avoids the generation of the intermediate data structure. Note that here the fusion is accomplished within the monad, and as a consequence, the 'pure' actions performed by the catamorphism are pushed into the monadic world. This form of composition is typical in programming practice. It corresponds to the application of semantic actions (given by a catamorphism) to the parse trees generated by a parser (given by a monadic anamorphism).

The following fusion laws can be established:

$$\begin{aligned} f \text{ strict } \wedge\ f \circ h = h' \circ Ff \ &\Rightarrow\ \widehat{f} \bullet \langle\!|h, g|\!\rangle_F = \langle\!|h', g|\!\rangle_F \\ g \circ f = M(Ff) \circ g' \ &\Rightarrow\ \langle\!|h, g|\!\rangle_F \circ f = \langle\!|h, g'|\!\rangle_F \end{aligned}$$

where the first law holds provided that M is locally continuous.

To conclude, we present an acid rain theorem for this kind of monadic hylomorphism.

Theorem 4.2 (Acid Rain)

(i) *Mhylo-Cata fusion:* For $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ polymorphic on A, M locally continuous and h strict,

$$M(\!|h|\!)_F \circ \langle\!|\, \boldsymbol{T}(in_F), g|\!\rangle_G = \langle\!|\, \boldsymbol{T}(h), g|\!\rangle_G$$

(ii) *Ana-Mhylo fusion:* For $\boldsymbol{T} : (A \to FA) \to (A \to MGA)$ polymorphic on A,

$$\langle\!|h, \boldsymbol{T}(out_F)|\!\rangle_G \circ [\!(g)\!]_F = \langle\!|h, \boldsymbol{T}(g)|\!\rangle_G$$

Example 4.3 (Search Procedures) Recall the formulation of graph traversals in terms of monadic anamorphism shown in Example 3.4. Suppose we are interested in applying a catamorphism to the outcome of a graph traversal.

$$V^* \xrightarrow{\mathsf{graphtrav}} V^* \xrightarrow{(\!|h|\!)_{L_V}} A$$

But, since $\mathsf{graphtrav} = \emptyset^M \circ \mathsf{gtrav}$, by naturality of $\emptyset^M$ (i.e. $f \circ \emptyset^M = \emptyset^M \circ Mf$ for every f) we can push the catamorphism into the monad,

$$(\!|h|\!)_{L_V} \circ \mathsf{graphtrav} = \emptyset^M \circ M(\!|h|\!)_{L_V} \circ \mathsf{gtrav}$$

and since gtrav is given by a monadic anamorphism,

$$(\!|h|\!)_{L_V} \circ \mathsf{graphtrav} = \emptyset^M \circ \langle\!|h, \mathsf{gopen}|\!\rangle_{L_V}$$

Now, consider the special case in which the catamorphism is the function $\mathsf{filter}(p) : V^* \to V^*$ that removes the elements of a list that do not satisfy a predicate $p : V \to \mathsf{Bool}$. This function can be written as a catamorphism $(\!|\mathsf{fil}|\!)_{L_V}$, for certain algebra fil. Therefore,

$$\mathsf{filter}(p) \circ \mathsf{graphtrav} = \emptyset^M \circ \langle\!|\mathsf{fil}, \mathsf{gopen}|\!\rangle_{LV}$$

represents a search procedure that explores a graph in determinate order with the aim at finding all vertices fulfilling a given predicate. As before, by specifying a concrete traversal strategy, typical search procedures like DFS or BFS can be obtained.

Example 4.4 (Monadic Parsers) In the case of monadic parsers, we can incorporate semantic actions to a syntax analyzer. This is captured by the following composition:

$$1 \xrightarrow{\mathsf{analyzer}} M\ \mathsf{Tree} \xrightarrow{M\ \mathsf{semantics}} MA$$

Since the syntax analyzer is given by a monadic anamorphism $[\![(g)]\!]^M$ and the semantic actions by a catamorphism $(\!|h|\!)$, their composition can be merged into a monadic hylomorphism:

$$\langle\!| h, g |\!\rangle = M\ \mathsf{semantics} \circ \mathsf{analyzer}$$

This equation can be interpreted as stating that: *The recursive structure of an interpreter/compiler for a language is characterized by the shape of recursion that comes with any monadic hylomorphism on the concrete syntax datatype.*

The equation above tells us that we can construct a parser applying the traditional modularization technique. We can develop separately each phase of the parser and at the end join them together into a single function that performs both tasks, but avoids the generation of parse trees.

In addition, the representation of monadic parsers in terms of monadic hylomorphism permits to perform formal reasoning with them, e.g. they can be subject of fusion transformations. Fusion transformations are mainly applied to semantic actions, as they usually represent complex actions of an interpreter or compiler for the given language.

As shown by Meijer [9], semantic actions can be developed in a modular way by using a calculational approach like the presented in this paper. Meijer's starting-point is the *abstract syntax* definition of the language. Thus, to be able to couple Meijer's development with the result of a syntax analyzer, we need to convert parse trees in *abstract syntax trees*. Roughly speaking, if F is the signature of the abstract syntax and G is the signature of the concrete syntax, then the conversion $\mathsf{conv} : \mu G \to \mu F$ from concrete syntax to abstract syntax can be specified by a catamorphism $(\!| \boldsymbol{T}(in_F) |\!)_G$ in terms of a polymorphic function $\boldsymbol{T} : (FA \to A) \to (GA \to A)$. The semantic actions are then given on the abstract syntax, i.e. $\mathsf{sem} = (\!|h|\!)_F : \mu F \to A$. As a result we have the following composite:

$$1 \xrightarrow{\mathsf{analyzer}} M\ \mu G \xrightarrow{M\ \mathsf{conv}} M\mu F \xrightarrow{M\ \mathsf{sem}} MA$$

Because $M\mathsf{conv} \circ \mathsf{analyzer} = \langle\!| \boldsymbol{T}(in_F), g |\!\rangle_G$, we can apply mhylo-cata-fusion (see Theorem 4.2) obtaining that the complete interpreter / compiler is given by a monadic hylomorphism:

$$M\mathsf{sem} \circ \langle\!| \boldsymbol{T}(in_F), g |\!\rangle_G = \langle\!| \boldsymbol{T}(h), g |\!\rangle_G$$

which avoids not only the construction of parse trees but also that of abstract syntax trees.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, UK, 1988.

[2] J. Peterson et. al. Report on the programming language Haskell (version 1.3). Technical Report YALEU/DCS/RR-1107, Yale University, 1996.

[3] Maarten M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Universiteit Twente, The Netherlands, 1992.

[4] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[5] Z. Hu and H. Iwasaki. Promotional Transformation of Monadic Programs. In *Fuji International Workshop on Functional and Logic Programming*, pages 196–210. World Scientific, July 1995. Available from `http://www.ipl.t.u-tokyo.ac.jp/~hu/`.

[6] J. Jeuring and P. Jansson. Polytypic Programming. In *Advanced Functional Programming,* LNCS 1129. Springer-Verlag, 1996.

[7] S. Peyton Jones and J. Launchbury. Lazy functional state threads. In *SIGPLAN PLDI'94*, pages 24–35, 1994.

[8] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4:413–424, 1992.

[9] E. Meijer. More Advice on Proving a Program Correct: Improve a Correct Compiler. Available from `http://www.cs.ruu.nl/~erik/`.

[10] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the ACM FPCA'91,* LNCS 523. Springer-Verlag, August 1991.

[11] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *ACM FPCA '95, SIGPLAN-SIGARCH-WG2.8*, pages 324–333, 1995.

[12] E. Meijer and G. Hutton. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[13] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In *Advanced Functional Programming,* LNCS 925, pages 228–266. Springer-Verlag, 1995.

[14] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1991.

[15] P.S. Mulry. Lifting Theorems for Kleisli Categories. In *9th International Conference on Mathematical Foundations of Programming Semantics,* LNCS 802, pages 304–319. Springer-Verlag, 1993.

[16] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A Calculational Fusion System HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France*, pages 76–106. Chapman & Hall, February 1997.

[17] A. Pardo. A Calculational Approach to Monadic and Comonadic Programs, forthcoming PhD Thesis. Technische Universität Darmstadt, 1998.

[18] A. Pardo. Monadic Corecursion: Definition, Fusion Laws and Applications. In *ETAPS'98 Workshop on Coalgebraic Methods in Computer Science, Lisbon, March 28-29.* volume 11 of ENTCS, 1998.

[19] S. Peyton-Jones and P. Wadler. Imperative Functional Programming. In *Proceedings of 20th Annual ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, 1993.

[20] A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Proceedings of FPCA'95*, 1995.

[21] P. Wadler. Monads for functional programming. In *Advanced Functional Programming,* LNCS 925. Springer-Verlag, 1995.