Monadic Corecursion —Definition, Fusion Laws, and Applications— [1]

Alberto Pardo [2]

Department of Computer Science, Darmstadt University of Technology
64283 Darmstadt, Germany
pardo@informatik.th-darmstadt.de

Abstract

This paper investigates corecursive definitions which are at the same time monadic. This corresponds to functions that generate a data structure following a corecursive process, while producing a computational effect modeled by a monad. We introduce a functional, called *monadic anamorphism*, that captures definitions of this kind. We also explore another class of monadic recursive functions, corresponding to the composition of a monadic anamorphism followed by (the lifting of) a function defined by structural recursion on the data structure that the monadic anamorphism generates. Such kind of functions are captured by so-called *monadic hylomorphism*. We present transformation laws for these monadic functionals. Two non-trivial applications are also described.

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 that they can be uniformly derived from data type definitions by using the categorical interpretation of recursive types. By categorical properties it is also possible to state general algebraic laws for recursive functionals to be used in the derivation, transformation and general reasoning (see e.g. [12,16,4,14,8,17]). Some of the general transformation laws essentially help eliminate intermediate data structures that arise in function

[1] Work presented at the ETAPS'98 Workshop on Coalgebraic Methods in Computer Science, Lisbon, March 1988, volume 11 of Electronic Notes in Theoretical Computer Science.
[2] Current address: Instituto de Computación, Facultad de Ingeniería, Julio Herrera y Reissig 565 - Piso 5, 11300 Montevideo, Uruguay. E-mail: `pardo@fing.edu.uy`.

compositions. These are the so-called *fusion* laws. In functional programming, the interest in fusion laws is mainly due to the wide utilization 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, since 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 [26,22] 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 that in such a composition arises.

In the last years it has become well-established that functional programs can also be structured by the *effects* they produce (or mimic to produce) using *monads* [29]. 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. [24,10]). But the occurrence of monadic effects within programs introduces a new dimension that needs to be considered when analyzing programs for program transformation, mainly when they involve recursion. In fact, there may be intermediate data structures generated by monadic recursive processes which are impossible to be systematically eliminated by existing deforestation techniques for 'pure' programs. In this sense, recent works [5,7,19] have focused on the study of fold computations combined with monads, introducing a functional called *monadic catamorphism*.

The purpose of this paper is to investigate the dual case, i.e. the combination of corecursion with monads. We will refer to the arising notion as *monadic corecursion*. It captures the behaviour of functions that generate a data structure in a "corecursive manner" while producing some effect represented by a monad. That is, like normal corecursion, the structure of these 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*. Fusion laws for monadic anamorphism are also studied.

Going further, we investigate the introduction of a notion of *monadic hylomorphism* as well. This corresponds to the composition of a monadic anamorphism followed by (the lifting of) a 'pure' catamorphism which consumes the intermediate data structure just generated by the monadic anamorphism. Similarly to hylomorphism, the virtue of monadic hylomorphism is the fact that it expresses this composition as a single function, avoiding therefore the generation of the intermediate data structure that is passed in the composition. The relevance of monadic hylomorphism is given by its fusion laws, as they

deal with new cases of deforestation in which intermediate data structures are eliminated in the presence of a monad.

The paper is organized as follows. Section 2 reviews the categorical approach to recursive datatypes and program transformation. Section 3 briefly introduces monads. In Section 4 we address the definition of monadic anamorphism, while Section 5 focus on the notion of monadic hylomorphism. In Section 6 we present two non-trivial applications that can be expressed by the monadic functionals we introduce. The first example deals with traversals and search procedures on graphs, whilst the second focuses on a popular technique in functional programming like is monadic parsing. Finally, Section 7 gives some concluding remarks.

2 Recursive Types and Program Transformation

This section briefly reviews the relevant concepts concerning the categorical approach to recursive datatypes [13,11], emphasising its application to the definition of standard recursive functionals and the derivation of calculational laws to formally deal with them [12,16,4,8].

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 $\bot$— and whose morphisms are continuous functions. A function $f : A \to B$ is said to be *strict* if it preserves the least element, i.e. $f(\bot_A) = \bot_B$. $\mathbf{Cpo}_\bot$ denotes the subcategory of **Cpo** obtained by considering only strict continuous functions as morphisms. The final object of $\mathcal{C}$ is denoted 1 and is given by the singleton set $\{\bot\}$.

Recall that from a recursive type definition we can derive an endofunctor $F : \mathcal{C} \to \mathcal{C}$ that captures the recursive shape (or signature) of the type. The recursive type is then interpreted as a solution to the equation $X \cong FX$, i.e. as a fixpoint of F. We assume that type signatures are given by so-called *regular functors*, which are described next. To fix notation we first introduce what we consider basic functors. $I : \mathcal{C} \to \mathcal{C}$ stands for the identity functor. For each object $A \in \mathcal{C}$, $\underline{A} : \mathcal{C}^n \to \mathcal{C}$ 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 $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ 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 $\langle f, g \rangle : C \to A \times B$. We consider that the sum bifunctor $+ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is given by *separated sum*:

$$A + B = (\{0\} \times A \cup \{1\} \times B)_\bot$$

We write $\mathsf{inl} : A \to A+B$ and $\mathsf{inr} : B \to A+B$ for the sum inclusions, such that $\mathsf{inl}(a) = (0, a)$ and $\mathsf{inr}(b) = (1, b)$. 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 product and sum functor can be generalized to n components in an obvious way. 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 $\dagger \in \{\times, +\}$.) This can be generalized to $F\langle G_1, \ldots, G_n\rangle$ for $F : \mathcal{C}^n \to \mathcal{C}$. D stands for a *type functor*; its definition is given later in this section. A functor F on **Cpo** is said to be *locally continuous* when its operation on functions is continuous. This condition is, in particular, satisfied by all regular functors.

For an endofunctor F on an arbitrary category $\mathcal{C}$, an *F-algebra* in $\mathcal{C}$ is a pair (A, h) where A is an object of $\mathcal{C}$ (called the carrier) and $h : FA \to A$ a morphism (called the operation). A morphism of algebras, or *F-homomorphism*, between $h : FA \to A$ and $k : FB \to B$ is an arrow $f : A \to B$ between the carriers that commutes with the operations, $f \circ h = k \circ Ff$. The category of F-algebras in $\mathcal{C}$ is formed by considering F-algebras as objects and F-homomorphisms as morphisms. Composition and identities in this category are inherited from $\mathcal{C}$.

The *canonical solution* to an equation $X \cong FX$ (for F locally continuous) is specified by a cpo μF together with an isomorphism $in_F : F\mu F \to \mu F$. The arrow in_F encodes the *constructors* of the datatype, while its inverse — to be called $out_F : \mu F \to F\mu F$— gives the *destructors*. When only strict functions are considered, i.e. in $\mathbf{Cpo}_\perp$, the pair $(\mu F, in_F)$ turns out to be an initial F-algebra. Initiality means the existence of a unique homomorphism from it to any other F-algebra. This permits to associate a functional — called *catamorphism* [16] [3] — that captures function definitions by structural recursion. For any strict algebra $h : FA \to A$, catamorphism is thus the unique strict function, denoted $(\!|h|\!)_F : \mu F \to A$, such that:

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

Note how it recursively replaces the constructors of the datatype by the target algebra h. In **Cpo**, however, the pair $(\mu F, in_F)$ does not form an initial algebra. This leads to introduce catamorphism by a fixed point definition, which satisfies also the equation above:

$$(\!|h|\!)_F = \mathsf{fix}(\lambda f.\, h \circ Ff \circ out_F)$$

Catamorphism enjoys many laws for program transformation. A law that plays an important rôle is the so-called *cata-fusion*, which states that the composition of a catamorphism with a homomorphism is again a catamorphism.

$$(1) \qquad f \text{ strict } \wedge\ f \circ h = h' \circ Ff \ \Rightarrow\ f \circ (\!|h|\!)_F = (\!|h'|\!)_F$$

[3] It can be found under other names in the literature, such as *fold operator* or *iterator*.

It can be proved by a simple fixed point induction.

For a functor F, a *F-coalgebra* in $\mathcal{C}$ is a pair (A, g) such that $g : A \to FA$ (also called the operation). The functor F plays again the rôle of signature of the structure. A mapping of coalgebras, or *F-cohomomorphism*, from $g : A \to FA$ to $g' : B \to FB$ is a morphism $f : A \to B$ such that, $g' \circ f = Ff \circ g$. Like for algebras, we can form a category of F-coalgebras with F-cohomomorphisms as morphisms. In **Cpo**, the pair $(\mu F, out_F)$ turns out to be a *final coalgebra*. Finality means the existence of a unique cohomomorphism from any coalgebra g to out_F, called *anamorphism*[4] and denoted by $[\!(g)\!]_F$. That is, it is the unique function that makes this diagram commute:

$$\begin{array}{ccc} A & \xrightarrow{[\!(g)\!]_F} & \mu F \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle out_F} \\ FA & \xrightarrow[F\,[\!(g)\!]_F]{} & F\mu F \end{array}$$

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 *fusion* law, which states that the composition of a cohomomorphism with an anamorphism yields an anamorphism.

$$g \circ f = Ff \circ g' \;\Rightarrow\; [\!(g')\!]_F \circ f = [\!(g)\!]_F \tag{2}$$

By fixing the first argument of a bifunctor $F : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ one can get a (parameterized) functor $F(A, -)$, to be written F_A, such that $F_A\, B = F(A, B)$ and $F_A\, f = F(\mathsf{id}_A, f)$. Functor 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 is a type constructor that 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 $f : A \to B$, which can be equally given by a catamorphism or an anamorphism.

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

An interesting property of type functors is the so-called *map-cata-fusion* law, which states that for $f : A \to B$ and $h : F_B\, C \to C$,

$$(\!|h|\!)_{F_B} \circ Df = (\!|h \circ F(f, \mathsf{id}_C)|\!)_{F_A} \tag{3}$$

There is a corresponding *ana-map-fusion* law, which states that for $f : A \to B$ and $g : C \to F_B C$,

$$Df \circ [\!(g)\!]_{F_A} = [\!(F(f, \mathsf{id}_C) \circ g)\!]_{F_B} \tag{4}$$

[4] Also called *unfold* or *coiterator*.

Example 2.1 (i) 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 $[\mathsf{nil}, \mathsf{cons}] : 1 + A \times \mathsf{List}(A) \to \mathsf{List}(A)$, where $\mathsf{nil} : 1 \to \mathsf{List}(A)$ and $\mathsf{cons} : A \times \mathsf{List}(A) \to \mathsf{List}(A)$. For an algebra $h = [h_N, h_C] : 1 + A \times B \to B$, the catamorphism is the (strict) function $f = (\!|h|\!)_{L_A} : \mathsf{List}(A) \to B$ such that,

$$f(\mathsf{nil}) = h_N \qquad f(\mathsf{cons}(a, \ell)) = h_C(a, f(\ell))$$

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]:

$$\mathsf{List}(f)(\mathsf{nil}) = \mathsf{nil} \qquad \mathsf{List}(f)(\mathsf{cons}(a, \ell)) = \mathsf{cons}(f(a), \mathsf{List}(f)(\ell))$$

In the sequel, mainly when applied on types, we will often write A^* for $\mathsf{List}(A)$.

(ii) The functor $S_A B = A \times B$ captures the signature of streams (i.e. infinite sequences) with elements over A. A stream coalgebra is a function $g = \langle h, t\rangle : B \to A \times B$ formed by the pairing of two functions $h : B \to A$ and $t : B \to B$. In particular, the final coalgebra $out_{S_A} = \langle \mathsf{head}, \mathsf{tail}\rangle : A^\infty \to A \times A^\infty$ where $A^\infty = \mu S_A$. The inverse of the final coalgebra gives the constructor of streams, to be called $\mathsf{scons} : A \times A^\infty \to A^\infty$. For a coalgebra g, the anamorphism $[\!(g)\!]_{S_A} : B \to A^\infty$ is the unique function such that $\mathsf{head} \circ [\!(g)\!]_{S_A} = h$ and $\mathsf{tail} \circ [\!(g)\!]_{S_A} = [\!(g)\!]_{S_A} \circ t$.

Suppose we are given an algebra $h : FA \to A$ and a coalgebra $g : B \to FB$. We define the *F-hylomorphism* [16] to be the function, denoted by $[\![h, g]\!]_F : B \to A$, given by

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

Inlining the fixpoint operator we obtain 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. First of all, because of the fact that catamorphism and anamorphism are special cases of it: $(\!|h|\!)_F = [\![h, out_F]\!]_F$ and $[\!(g)\!]_F = [\![in_F, g]\!]_F$. These equation follow immediately from definitions. In addition, one can observe that a hylomorphism corresponds to the composition of a catamorphism with an anamorphism on the same datatype,

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

as can be verified by a simple fixed point induction. The relevance of this equation is that it shows we can always transform the composition of a (standard) producing function followed by a (standard) consuming one into a monolithic function that avoids the unnecessary construction of the intermediate data structure. The following fusion laws are a direct consequence of this fact:

(5) $\quad f$ strict $\;\wedge\; f \circ h = h' \circ F f \Rightarrow f \circ [\![h, g]\!]_F = [\![h', g]\!]_F$

$$g \circ f = Ff \circ g' \Rightarrow [\![h, g]\!]_F \circ f = [\![h, g']\!]_F \tag{6}$$

They correspond to the fusion laws (1) and (2) for catamorphism and anamorphism, respectively, now applied to a hylomorphism.

The expressive power of hylomorphism is very rich. In fact, most practical recursive functions of interest can be directly represented as an hylomorphism [2]. Recent work [26,22] has shown the usefulness of hylomorphism in the application of deforestation techniques. The basis of its success has been given by the introduction of two calculational laws —usually referred to together as the *Acid Rain Theorem* [26]— which, under certain conditions, permit to fuse the composition of a hylomorphism with a catamorphism/anamorphism into a hylomorphism that avoids building the intermediate data structure that is passed in the composition. To be able to present the Acid Rain Theorem, we first need to introduce the concept of a transformer [4]. A functional $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ (parametric on A) that converts each F-algebra into a G-algebra on the same carrier is said to be an *algebra transformer* whenever, for every $f : A \to B$, $h : FA \to A$ and $h' : FB \to B$, if $f \circ h = h' \circ Ff$ then $f \circ \boldsymbol{T}(h) = \boldsymbol{T}(h') \circ Gf$. That is, if every homomorphism between two F-algebras happens to be also a homomorphism between the transformed G-algebras. Of course, we can also define a similar notion for coalgebras. A functional $\boldsymbol{T} : (A \to FA) \to (A \to GA)$ is a *coalgebra transformer* whenever for every $f : A \to B$, and coalgebras $g : A \to FA$ and $g' : B \to FB$, it holds that if $g' \circ f = Ff \circ g$ then $\boldsymbol{T}(g') \circ f = Gf \circ \boldsymbol{T}(g)$.

Theorem 2.2 (Acid Rain)

Cata-hylo fusion: *Let $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ be an algebra transformer. For $h : FA \to A$ strict and $g : B \to GB$,*

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

Hylo-ana fusion: *Let $\boldsymbol{T} : (A \to FA) \to (A \to GA)$ be a coalgebra transformer. For $h : GA \to A$ and $g : B \to FB$,*

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

Proof. We only present the proof for cata-hylo fusion; the other case is analogous. Recall that every catamorphism is an algebra homomorphism. Thus, by definition of algebra transformer we obtain that $(\!|h|\!)_F : \mu F \to A$ is also a homomorphism between the G-algebras $\boldsymbol{T}(in_F) : G\mu F \to \mu F$ and $\boldsymbol{T}(h) : GA \to A$. In addition, $(\!|h|\!)_F$ is strict, since by hypothesis we assumed that h is strict. Therefore by applying hylo-fusion (5) we arrive at the desired equation. □

3 Monads

Monads have been proposed by Moggi [20] as a device for structuring denotational semantics descriptions of programming languages. Monads permit to encapsulate in abstract terms several forms of computations, such as exceptions, state, I/O, continuations or nondeterminism. In this approach compu-

tational 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. This produces an explicit distinction between *values* and *computations*. In the last years, the use of monads has become very popular among the functional programming community. The migration of Moggi's ideas from denotational semantics to functional programming was due to Wadler [28,29], who established a style of programming suitable for structuring *purely* functional programs that mimic *impure* features. A typical program in monadic style is a function $A \to MB$ that computes values of type B from values of type A, while producing some effect.

Formally, monads possess alternative definitions. The following definition presents monads as so-called Kleisli triples.

Definition 3.1 A **Kleisli triple** $(M, \mathsf{unit}, -^\star)$ over $\mathcal{C}$ is given by the restriction $M : Obj(\mathcal{C}) \to Obj(\mathcal{C})$ of a functor M to objects, a natural transformation $\mathsf{unit} : I \Rightarrow M$, and an extension operator $-^\star$ which for each $f : A \to MB$ yields $f^\star : MA \to MB$, such that these equations hold: $\mathsf{unit}^\star_A = \mathsf{id}_{MA}$, $f^\star \circ \mathsf{unit}_A = f$, 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 (or monadic) composition is defined by $f \bullet g \stackrel{\text{def}}{=} f^\star \circ g$. 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 form a category.

Definition 3.2 For each Kleisli triple $(M, \mathsf{unit}, -^\star)$ over $\mathcal{C}$, the **Kleisli category** $\mathcal{C}_M$ is defined as follows: the objects of $\mathcal{C}_M$ are those of $\mathcal{C}$; morphisms between objects A and B in $\mathcal{C}_M$ correspond to arrows $A \to MB$ in $\mathcal{C}$, i.e. $\mathcal{C}_M(A, B) \equiv \mathcal{C}(A, MB)$; identities are given by $\mathsf{unit}_A : A \to MA$; and composition is given by Kleisli composition.

Given a monad over $\mathcal{C}$, we can define a *lifting functor* $\widehat{(-)} : \mathcal{C} \to \mathcal{C}_M$ in the obvious way. On objects, $\widehat{A} = A$, while on arrows, $\widehat{f} = \mathsf{unit}_B \circ f : A \to MB$, for $f : A \to B$. We can also define a functor $U : \mathcal{C}_M \to \mathcal{C}$ such that, on objects, $UA = MA$, and on arrows, $Uf = f^\star : MA \to MB$ for $f : A \to MB$. It is simple to verify that $U\widehat{(-)} = M$. These two functors permit to establish the canonical adjunction $\widehat{(-)} \dashv U : \mathcal{C}_M \to \mathcal{C}$, with $\eta = \mathsf{unit}$ and $\epsilon = \mathsf{id}$.

A monad can be alternatively defined as a triple (M, unit, μ) given by an endofunctor $M : \mathcal{C} \to \mathcal{C}$ and two natural transformations $\mathsf{unit}_A : I \Rightarrow M$ and $\mu : MM \Rightarrow M$ (called the *multiplication*) satisfying the laws: $\mu_A \circ \mathsf{unit}_{MA} = \mathsf{id}_{MA} = \mu_A \circ M(\mathsf{unit}_A)$ and $\mu_A \circ \mu_{MA} = \mu_A \circ M(\mu_A)$. Both formulations of a monad are equivalent. In fact, from the Kleisli triple components we can define the action of M on an arrow $f : A \to B$ by $Mf = (\mathsf{unit}_B \circ f)^\star$ and $\mu_A = id^\star_{MA}$. Conversely, every Kleisli triple can be constructed from a monad

(M, unit, μ) by considering the restriction of the functor M to objects, and defining the extension of each $f : A \to MB$ to be $f^\star = \mu_B \circ Mf$.

The following fact should belong to the folklore.

Lemma 3.3 *In* **Cpo** *the multiplication μ of every monad is a strict operation.*

Proof. By definition of monad $\mu_A \circ \mathsf{unit}_{MA} = \mathsf{id}_{MA}$ for each object A. There is a fact in **Cpo** that states that if $A \to B \to C$ is strict then so is $B \to C$ (see [6]). Therefore μ_A is strict as so is the identity. □

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 [29]). 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. In fact, within functional programs it is often to find expressions of the form $m \star \lambda v.\, m'$, which are read as follows: evaluate computation m, bind the variable v to the resulting value, and then continue with the evaluation of computation m'. 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.

Example 3.4 (i) Of all monads the simplest one is the *identity monad*: $M = I$; $\mathsf{unit}_A = \mathsf{id}_A$; and $f^\star = f$. It simply captures function application, since $f \bullet g = f \circ g$.

(ii) The *exception monad* models the occurrence of exceptions in a program. If E stands for a type of exception values, then $MA = A + E$ captures computations that either succeed returning a value of type A, or fail raising an exception signaled by a value of type E. The unit and extension operator are given by:

$$\mathsf{unit}_A = \mathsf{inl} \qquad\qquad f^\star = [f, \mathsf{inr}]$$

for $f : A \to B + E$. That is, unit takes a value and returns a computation that always succeeds. The extension may be thought of as a form of strict function application which propagates the exception if it is the case. In the special case we have only one exception value ($E = 1$), the exception monad is also referred to as the *maybe monad* [19].

(iii) The *state-transformer monad* (or *state monad* for short) represents computations that take an initial state and return a value and a new state. If S stands for the state space, then $MA = [S \to A \times S]$. In functional terms, the unit and bind operator are given by

$$\mathsf{unit}(a) = \lambda s.(a, s) \qquad m \star f = \lambda s.\, \mathbf{let}\ (a, s') = m(s)\ \mathbf{in}\ f(a)(s')$$

for $f : A \to [S \to B \times S]$. That is, unit takes a value and returns a computation

that yields this value without modifying the state; whereas $\star$ sequences two computations so that the state and value resulting from the first are supplied to the second. In recent years there have been various proposals that show how the state monad can be used as a mechanism to encapsulate actual *imperative* features —such as mutable variables, in-place updatable data structures, and I/O— in a functional setting while retaining fundamental properties like referential transparency (see e.g. [29,10]). This is achieved by hiding the *real* state in an abstract data type based on the monad and equipped with operations that internally access to the real state. The technique can be used either when the state is internal or external to the program. This approach has been adopted by the Haskell [3] community.

A monad on a cartesian category $\mathcal{C}$ is said to be *strong* if it comes equipped with a natural transformation $\tau_{A,B} : A \times MB \to M(A \times B)$ —called a *strength*— satisfying certain equational axioms (see [20]). Intuitively, since a function f (thought of as a lambda term) being extended in $f^\star$ need not be closed, we need a way of distributing the free variable values in the context along the monad. In a strong monad this is possible thanks to the strength. The strength can be interpreted as the following function:

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

It is possible to define also a dualization of the strength $\tau'_{A,B} : MA \times B \to M(A \times B)$ satisfying similar axioms. The strengths induce a natural transfomation $\psi_{A,B} : MA \times MB \to M(A \times B)$ given by $\psi_{A,B} = \tau_{A,B} \bullet \tau'_{A,MB}$, which describes how the monad distributes over the product. It says that a pair of computations may be joined as a new computation by first evaluating the first argument and then the second. That is,

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

The following equation holds: $\psi_{A,B} \circ (\mathsf{unit}_A \times \mathsf{id}_{MB}) = \tau_{A,B}$. Similarly, we can define a right-to-left product distribution $\psi'_{A,B} = \tau'_{A,B} \bullet \tau_{MA,B}$. A strong monad is said to be *commutative* if the product distributions coincide, i.e. $\psi_{A,B} = \psi'_{A,B}$. Examples of commutative monads are the identity monad, the exception monad and the environment monad [28]. On the contrary, the state monad and the list monad [29] are non-commutative.

4 Monadic Corecursion

In this section we elaborate the notion of monadic corecursion and introduce a recursive functional that behaves accordingly, called monadic anamorphism. One way of approaching to monadic anamorphism is by dualizing the recursion scheme that characterizes *monadic catamorphism* [5,7]. However, we have opted to give instead a direct introduction to this concept by means of an intuitive explanation of its behavior.

A first approximation to the notion of monadic corecursion can be got by

considering the usual corecursion scheme captured by the diagram:

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & \mu G \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle out_G} \\ GA & \xrightarrow[\quad Gf \quad]{} & G\mu G \end{array}$$

but viewing it as a diagram in the category $\mathcal{C}_M$ of an arbitrary monad M. Proceeding that way we are thinking of each arrow as an effect-producing function, getting the somewhat 'imperative' idea of a corecursive process that produces some side-effect along its evaluation. Since category $\mathcal{C}$ 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 such a scheme. The first step is to make the computational effect explicit.

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & M\mu G \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle out^\star_G} \\ M(GA) & \xrightarrow[\quad (Gf)^\star \quad]{} & M(G\mu G) \end{array}$$

It rests to determine who play the rôle of G and out_G, and then we are ready to define the new functional out of this diagram. Recall that the objects of $\mathcal{C}$ and $\mathcal{C}_M$ coincide. So, the data structure generated by such a recursive definition necessarily corresponds to a datatype μF, for some functor F on $\mathcal{C}$. G corresponds thus to some *monadic extension* of F —to be denoted $\widehat{F}$— which on objects coincides with F and that acts on monadic functions.

Following type considerations, it is possible to see that $\widehat{out_F}$ —the lifting of the final coalgebra out_F— is the natural candidate to play the rôle of out_G. Intuitively, this arrow permits to perform single observations to the data structure generated by the corecursive process, just propagating the computational effect. In summary, a monadic corecursive definition will correspond to a function f satisfying this diagram in $\mathcal{C}$:

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & M(\mu F) \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \widehat{out_F}^\star} \\ M(FA) & \xrightarrow[\quad (\widehat{F}f)^\star \quad]{} & M(F\mu F) \end{array} \tag{7}$$

The next subsection discusses the definition and properties of the monadic extension of a (regular) functor. Function $g : A \to M(FA)$ is called a *monadic coalgebra*. In subsection 4.2 we briefly analyze this notion and present alternative forms of coalgebra mappings. The precise definition of monadic anamorphism and some of its calculational laws are presented in Subsection 4.3. The

material treated in Subsections 4.1 and 4.2 holds for an arbitrary category $\mathcal{C}$, except where explicitly otherwise stated.

4.1 The Monadic Extension of a Functor

Earlier we have seen that morphisms in $\mathcal{C}$ can be translated to the Kleisli category by applying the lifting functor $(\widehat{-})$. Now we study how to proceed with functors on $\mathcal{C}$ in order to lift them to the monadic world. For every functor F we derive a construction $\widehat{F}$ that acts on elements of the Kleisli category. In particular, we show an inductive definition of $\widehat{F}$ when F is regular, analogous to those presented in [5,7,27]. In addition, we give a brief account of properties satisfied by $\widehat{F}$. (A more complete presentation can be found in [23].)

The *monadic extension* $\widehat{F} : \mathcal{C}_M \to \mathcal{C}_M$ of a functor $F : \mathcal{C} \to \mathcal{C}$ is a construction such that: on objects $\widehat{F}A = FA$, since recall that the objects of $\mathcal{C}_M$ and $\mathcal{C}$ coincide; 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 $\widehat{F}f : FA \to M(FA)$— in $\mathcal{C}_M$, whose action embeds that of F. 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 such a 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 $\mathcal{C}$ mentioned earlier (see [21,23]). As a result we get that for each A, $\delta^F_A = \widehat{F}\mathsf{id}_{MA}$.

$\widehat{F}$ is called a *lifting* of F when it is a functor on $\mathcal{C}_M$. In that case, the following diagram between functors commute:

$$\begin{array}{ccc} \mathcal{C}_M & \xrightarrow{\widehat{F}} & \mathcal{C}_M \\ {\scriptstyle(\widehat{-})}\uparrow & & \uparrow{\scriptstyle(\widehat{-})} \\ \mathcal{C} & \xrightarrow[F]{} & \mathcal{C} \end{array}$$

Observe that every monadic extension makes this diagram always commute on objects. The following theorem specifies when a functor F has a lifting.

Theorem 4.1 ([21]) *Given a monad M, $\widehat{F} : \mathcal{C}_M \to \mathcal{C}_M$ is a lifting of F iff the natural transformation $\delta^F : FM \Rightarrow MF$ satisfies the equations:*

(8) $\qquad \delta^F_A \circ F\mathsf{unit}_A = \mathsf{unit}_{FA}$

(9) $\quad \mu_{FA} \circ M\delta^F_A \circ \delta^F_{MA} = \delta^F_A \circ F\mu_A$

These equations care for the functoriality axioms for $\widehat{F}$. Equation (8) cares

for the preservation of identities, $\widehat{F}\mathsf{unit}_A = \mathsf{unit}_{FA}$, while (9) makes lifting distribute over composition in the Kleisli category, $\widehat{F}(f \bullet g) = \widehat{F}f \bullet \widehat{F}g$.

Proposition 4.2 *Let M be a strong monad. Then the functor $F = A \times -$, for $A \in \mathcal{C}$, has a lifting with δ^F given by the strength. The same holds for $F = - \times A$, where δ^F is given by the dualization of the strength.*

The conditions for the existence of a lifting can be straightforwardly generalize to the case of multiary functors $F : \mathcal{C}^n \to \mathcal{C}$. The following proposition uses such generalize conditions.

Proposition 4.3 *If the monad M is commutative, then the product functor $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ has a lifting with $\delta^{\times}_{(A,B)} = \psi_{A,B}$.*

When the monad is strong but non-commutative —like e.g. the state monad— the product functor makes (8) hold equally, while (9) fails:

$$\psi_{A,B} \circ (\mathsf{unit}_A \times \mathsf{unit}_B) = \mathsf{unit}_{A \times B}$$

$$\mu_{A \times B} \circ M\psi_{A,B} \circ \psi_{MA,MB} \neq \psi_{A,B} \circ (\mu_A \times \mu_B)$$

that is, the monadic extension $\widehat{\times}$ lacks the preservation of Kleisli composition. Let us see now what happens with the sum functor.

Proposition 4.4 *Let $\mathcal{C}$ be a category with coproduct. Then, the sum functor $+ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ has a lifting with $\delta^{+}_{(A,B)} = [M\mathsf{inl}, M\mathsf{inr}]$.*

It is interesting to see what happens if we consider the monadic extension of the sum functor in **Cpo**. In that case, for $+$ given either by the coalesced sum or the separated sum, $\widehat{+}$ fails in general to be a lifting since it lacks the preservation of identities, i.e.

$$\delta^{+}_{(A,B)} \circ (\mathsf{unit}_A + \mathsf{unit}_B) \neq \mathsf{unit}_{A+B}$$

It is said to be a *semi-functor*. The reason for the failure is because, for the verification of this equation, it is necessary that unit be strict, and this is not the case in general (e.g., unit is not strict for the exception and the state monad). On the other hand, the preservation of Kleisli composition for $\widehat{+}$ holds thanks to the strictness of the multiplication μ (stated in Lemma 3.3).

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

Definition 4.5 Let M be a strong monad. Then,

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

That $\delta^\times$ is defined as ψ means that the monadic effects in product expressions are sequenced from left-to-right; a right-to-left policy can also be specified by using ψ' instead. In the last line F is the bifunctor that induces D. The definition of δ^D is a form of monadic catamorphism.

From Definition 4.5 these typical cases can be calculated:

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

Example 4.6 (i) For the functor $L_A = \underline{1} + \underline{A} \times I$, we have:

$$\delta_B^{L_A} = [\widehat{\mathsf{inl}}, \widehat{\mathsf{inr}} \bullet \tau_{A,B}] : 1 + A \times MB \to M(1 + A \times B)$$

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

$$\delta_B^{R_A} = \tau_{A,B^*} \circ (\mathsf{id}_A \times \delta^{(-)^*}) : A \times (MB)^* \to M(A \times B^*)$$

where $\delta^{(-)^*} = (\!|[\widehat{\mathsf{nil}}, \widehat{\mathsf{cons}} \bullet \psi_{B,B^*}]|\!) : (MB)^* \to MB^*$. Inlining,

$$\delta^{(-)^*}(\mathsf{nil}) = \mathsf{unit}(\mathsf{nil})$$

$$\delta^{(-)^*}(\mathsf{cons}(m, ms)) = m \star \lambda b.\, \delta^{(-)^*}(ms) \star \lambda bs.\, \mathsf{unit}(\mathsf{cons}(b, bs))$$

By means of $\delta^{(-)^*}$ we can collect the results and accumulate the effects produced by the execution of a list of computations from left-to-right, (A right-to-left accumulation is accomplished when $\delta^\times$ is defined as ψ'.)

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

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

In particular,

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

$\widehat{D}f$ is usually called a *monadic map* [5]; its expression is the result of applying map-cata-fusion (see law (3)) to $\delta^D \circ Df$.

Example 4.7 Let us see the extensions of the functors in Example 4.6.

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

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

Let us discuss some properties of the defined extensions.

Theorem 4.8 *Let $\mathcal{C}$ be a category with product and coproduct such that every regular functor on it has an initial algebra. Let M be a commutative monad. Then, all regular functors have a lifting.*

The proof consists of the verification of equations (8) and (9) for all $\delta's$ in Definition 4.5 (see [27,23]). The proofs for the product and sum functors are given by Propositions 4.3 and 4.4. Note that commutativity is only required for the product functor. We can achieve weaker results if we eliminate this assumption.

Proposition 4.9 *Let $\mathcal{C}$ be as above. Let M be a strong monad. Then,*

(i) *The extension of every regular functor preserves identities.*

(ii) *All regular functors containing only product expressions of the form $A \times -$ or $- \times A$, for some object A, have a lifting.*

Case (i) is a consequence of the fact that commutativity is never used for the verification of equation (8). For case (ii), recall that as stated in Proposition 4.2 the functors $A \times -$ and $- \times A$ can be lifted for any strong monad.

Again, it is interesting to analyze what happens when we consider monadic extensions of functors on **Cpo**. Recall that $\widehat{+}$ is a semi-functor in **Cpo**. The monadic extension of type functors may also fail to preserve identities. In the facts above, the validity of equation (8) for a type functor D,

$$\delta^D_A \circ D\,\mathsf{unit}_A = (\!|Min_{F_A} \circ \widehat{F}(\mathsf{unit}, \mathsf{id})|\!)_{F_A} = \mathsf{unit}_{DA}$$

relies on initiality, since unit satisfies the characteristic equation of the catamorphism. In **Cpo**, however, this equality holds if, in addition, both unit and the catamorphism coincide in their behaviour on $\perp$, and this is not the case in general. In fact, while on the one hand unit is not necessarily strict, on the other hand $(\!|Min_{F_A} \circ \widehat{F}(\mathsf{unit}, \mathsf{id})|\!)_{F_A}$ turns out to be strict for a large class of functors, as so is the algebra $Min_{F_A} \circ \widehat{F}(\mathsf{unit}, \mathsf{id})$. For instance, a typical case is when $F = F_1 + F_2$. Then, $Min_{F_A} \circ \widehat{F}(\mathsf{unit}, \mathsf{id}) = Min_{FA} \circ [M\mathsf{inl} \circ \widehat{F_1}(\mathsf{unit}, \mathsf{id}), M\mathsf{inr} \circ \widehat{F_2}(\mathsf{unit}, \mathsf{id})]$. Recall that in_F is strict as it is an isomorphism. Thus, in order Min_{F_A} to be strict, M needs to be a strictness-preserving functor —i.e. a functor that maps a strict function f to a strict function Mf. Fortunately, as shown by Freyd [6], this automatically holds for every functor on **Cpo** that happens to be locally continuous.[5] Since, in addition, every case analysis is strict by definition, it follows that the algebra $Min_{F_A} \circ \widehat{F}(\mathsf{unit}, \mathsf{id})$ is strict. On the other hand, concerning the presevation of Kleisli composition we can state the following fact.

Proposition 4.10 *Let M be a stong monad and let $\mathcal{C}$ be* **Cpo**. *Then,*

(i) *If M is commutative then all regular functors extend to (at least) semi-functors.*

(ii) *All regular functors containing only product expressions of the form $A \times -$ or $- \times A$ extend (at least) to semi-functors.*

[5] The actions M of the monads commonly used in functional programming are locally continuous.

Example 4.11 Consider again the functors L_A and R_A. Suppose that M is the exception or the state monad. (For these monads unit is not strict.) Then, (i) $\widehat{L_A}$ is a semi-functor, as L_A is given in terms of a sum; and (ii) $\widehat{R_A}$ is a semi-functor, as so is $\widehat{\mathsf{List}}$.

4.2 Monadic Coalgebras

Consider a functor F and a monad M. A *monadic F-colagebra* is a pair (A, g) consisting of an object A (the carrier) and an arrow $g : A \to M(FA)$ (the operation). Like (plain) coalgebras, the arrow g may be thought of as a *structure*, now returning a computation instead of simply a value. The functor F plays again the rôle of *signature* of the structure. Using the monadic extension $\widehat{F}$, one might then say that a monadic coalgebra $A \to M(FA)$ (in $\mathcal{C}$) is a $\widehat{F}$-coalgebra $A \to \widehat{F}A$ in $\mathcal{C}_M$. (Recall that on objects, $\widehat{F}A = FA$.)

A structure preserving mapping between two monadic coalgebras should be an arrow between their carriers that preserves their structures and is *compatible* with their monadic effects. After regarding monadic coalgebras as coalgebras in $\mathcal{C}_M$, the concept of coalgebra cohomomorphism arises as a natural candidate to play the rôle of the desired notion of structure-preserving mapping. We thus say that an arrow $f : A \to MB$ is a *monadic F-cohomomorphism* between two monadic coalgebras $g : A \to M(FA)$ and $g' : B \to M(FB)$ if $g' \bullet f = \widehat{F}f \bullet g$. So defined, a monadic cohomomorphism is an arrow that itself produces a monadic effect, which is compatible with that of the monadic coalgebras.

There is a particular class of structure-preserving mappings which are given by pure mappings between the carriers of monadic coalgebras. We say that an arrow $f : A \to B$ is a *pure cohomomorphism* between $g : A \to M(FA)$ and $g' : B \to M(FB)$ if the following diagram commutes.

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle g'} \\ M(FA) & \xrightarrow[M(Ff)]{} & M(FB) \end{array}$$

Note that two monadic coalgebras are connected by a morphism of this kind only when the monadic effects produced by them coincide on f-related inputs. That is, when for any $a \in A$ and $b \in B$ such that $f(a) = b$, the monadic effects produced by $g(a)$ and $g'(b)$ are the same. This is because the arrow $M(Ff)$ maps a value of type FA to a value of type FB, simply propagating the monadic effect.

The following laws are basic facts relating both cohomomorphism versions. Let g, g' and g'' be monadic coalgebras. If $\widehat{F}$ preserves identities, then

(10) $g' \circ f = M(Ff) \circ g \quad \Rightarrow \quad g' \bullet \widehat{f} = \widehat{F}\widehat{f} \bullet g$

The assumption about $\widehat{F}$ makes the equality $\widehat{Ff} = \widehat{F}\widehat{f}$ hold, and this is used in the proof of the law. The following two laws state restricted forms of composition between monadic cohomomorphisms which are always doable.

$$(11)\quad \left.\begin{array}{l} g' \circ f = M(Ff) \circ g \\ g'' \bullet f' = \widehat{F} f' \bullet g' \end{array}\right\} \;\Rightarrow\; g'' \bullet (f' \circ f) = \widehat{F}(f' \circ f) \bullet g$$

$$(12)\quad \left.\begin{array}{l} g' \bullet f = \widehat{F} f \bullet g \\ g'' \circ f' = M(Ff') \circ g' \end{array}\right\} \;\Rightarrow\; g'' \bullet (\widehat{f'} \bullet f) = \widehat{F}(\widehat{f'} \bullet f) \bullet g$$

Only when $\widehat{F}$ is a lifting or a semi-functor the following kind of composition can be considered:

$$(13)\quad \left.\begin{array}{l} g' \bullet f = \widehat{F} f \bullet g \\ g'' \bullet f' = \widehat{F} f' \bullet g' \end{array}\right\} \;\Rightarrow\; g'' \bullet f' \bullet f = \widehat{F}(f' \bullet f) \bullet g$$

The following laws establish the relationship between coalgebra mappings and the monadic cohomomorphism versions. Let g and g' be two (plain) coalgebras. Then,

(14) $g' \circ f = Ff \circ g \;\Rightarrow\; \widehat{g'} \circ f = M(Ff) \circ \widehat{g}$

(15) $g' \circ f = Ff \circ g \;\Rightarrow\; \widehat{g'} \bullet \widehat{f} = \widehat{F}\widehat{f} \bullet \widehat{g}$

Law (15) holds provided that $\widehat{F}$ preserves identities.

4.3 Monadic Anamorphism

Recall diagram (7) given earlier in this section. Observe that, in particular, this diagram expresses that a monadic corecursive function is a monadic cohomomorphism between g and the lifting of the final coalgebra. It is worth noting, however, that there may exist multiple functions fulfilling this diagram. In other words, $\widehat{out_F}$ is not *final*. The following counterexample gives evidences of this fact.

Example 4.12 Let M be the maybe monad $MA = A + 1$, and let F be the functor $S_A = \underline{A} \times I$ that captures the signature of streams over A. Consider the natural transformation $\mathsf{fail}_{A,B} = \lambda a.\mathsf{inr}(\bot) : A \to MB$ that fails for every value of type A. It satisfies an absorption property, namely that for any $k : A \to MB$ and $k' : B \to MC$, $k' \bullet \mathsf{fail}_{A,B} = \mathsf{fail}_{A,C} = \mathsf{fail}_{B,C} \bullet k$. In addition, it holds that $\widehat{S_A}\,\mathsf{fail} = \mathsf{fail}$. Hence,

$$\widehat{S_A}\,\mathsf{fail}_{B,A^\infty} \bullet g = \mathsf{fail}_{B,A\times A^\infty} = \widehat{out_{S_A}} \bullet \mathsf{fail}_{B,A^\infty}$$

which shows that $\mathsf{fail}_{B,A^\infty}$ satisfies diagram (7) for any g.

Now consider $g = \mathsf{unit} \circ \langle h, t\rangle : B \to M(A \times B)$ for $h : B \to A$ and $t : B \to B$. Since out_{S_A} is the final stream coalgebra, there exists a unique cohomomorphism from $\langle h, t\rangle$ to out_{S_A} given by the anamorphism $[\![\langle h, t\rangle]\!]_{S_A}$:

$B \to A^\infty$. Proposition 4.2 tells us that $\widehat{S_A}$ is a lifting (note that the maybe monad is strong). Therefore by (15) it follows that the lifting of the anamorphism is a monadic cohomomorphism between the liftings of $\langle h, t\rangle$ and out_{S_A}, and thereby, like fail, it satisfies diagram (7).

We will then adopt the *least* solution to diagram (7) as the definition of monadic anamorphism. To calculate it we consider the functional

$$\phi(f) = \widehat{in_F} \bullet \widehat{F} f \bullet g$$

that arises from reversing $\widehat{out_F}$ in the diagram. This is possible because $\widehat{out_F}$ is an isomorphism —with inverse $\widehat{in_F}$ w.r.t. Kleisli composition— since so is out_F and lifting $(\widehat{-})$ is a functor. (Recall that functors preserve isomorphisms.)

Definition 4.13 Let F be a locally continuous functor and let M be a strong monad. The **monadic anamorphism** for $g : A \to M(FA)$, denoted by $[\![g]\!]^M_F : A \to M\mu F$, is defined as $\mathsf{fix}(\phi)$.

Example 4.14 Consider the functor $S_A = \underline{A} \times I$. Since $\widehat{S_A} f = \tau \circ (\mathsf{id}_A \times f)$, then the monadic anamorphism $[\![g]\!]^M_{S_A}$ for $g : B \to M(A \times B)$ is the least function satisfying the equation $f = \widehat{\mathsf{scons}} \bullet (\tau \circ (\mathsf{id} \times f)) \bullet g$, i.e.

$$f(b) = g(b) \star \lambda(a, b').\, f(b') \star \lambda s.\, \mathsf{unit}(\mathsf{scons}(a, s))$$

In Example 4.12 we saw that, when M is the maybe monad and $g = \mathsf{unit} \circ \langle h, t\rangle$, both fail and $\mathsf{unit} \circ [\![\langle h, t\rangle]\!]_{S_A}$ are solutions to this recursive equation. However, neither of them correspond to $[\![g]\!]^M_{S_A}$, which, in contrast, is given by the completely undefined function $\lambda b.\, \bot$. The reason why the monadic anamorphism yields no response at all can be observed in the chain of computations that the iterative unfolding of g generates:

$$g(b) \star \lambda(a_1, b_1).\, g(b_1) \star \lambda(a_2, b_2).\, g(b_2) \star \lambda(a_3, b_3).\, \cdots$$

Since for every $x \in B$ the computation $g(x) = \mathsf{unit}(h(x), t(x))$ succeeds, the iterative unfolding proceeds infinitely. This means that we should wait infinite time to be able to resolve the $\star$'s in this expression and to extract the stream of results $(a_1, a_2, \ldots)$. This kind of "resolution in the infinite" is precisely what function $\mathsf{unit} \circ [\![\langle h, t\rangle]\!]_{S_A}$ models, but does not correspond to the computational behaviour. This shows the inconvenience of using the maybe monad in combination with the generation of infinite data structures.

Now, we turn to the discussion of calculational laws for monadic anamorphism. Because a monadic anamorphism is, in particular, a monadic cohomomorphism, it can be composed with other monadic morphisms only in the forms presented in Subsection 4.2. As first law, we present what we call *pure fusion*. It states that a pure cohomomorphism followed by a monadic anamorphism is again a monadic anamorphism.

$$(16)\ g \circ f = M(Ff) \circ g' \quad \Rightarrow \quad [\![g]\!]^M_F \circ f = [\![g']\!]^M_F$$

It is proved by a simple fixed point induction. When $\widehat{F}$ is a lifting or a semi-

functor, we can also state a *mana-fusion-law*.

(17) $g \bullet f = \widehat{F} f \bullet g' \quad \Rightarrow \quad [\![(g)]\!]^M_F \bullet f = [\![(g')]\!]^M_F$

which is also verified by a simple fixpoint induction.

Finally, we present a law called *mana-map-fusion*, which corresponds to ana-map-fusion (see (4)) in the monadic case. Suppose that M is locally continuous. Then, for $f : A \to B$ and $g : C \to F_A C$,

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

This law is got as an instance of Proposition 5.3, to be presented in the next section, and that states the result of composing a monadic anamorphism with a catamorphism. Indeed, recall that by definition a type functor is given by a catamorphism (see Section 2).

5 Monadic Hylomorphism

Having developed a monadic extension of anamorphism, it seems natural to investigate the notion of monadic hylomorphism too. The introduction of such a notion turns out to be not only of theoretical interest, but also of practical relevance, as it permits to achieve new cases of deforestation which are by now impossible to be considered.

Similarly to monadic anamorphism, we can proceed to introduce monadic hylomorphism by regarding each component of a hylomorphism as being of monadic nature. However, 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 [23].)

Definition 5.1 Given a monadic coalgebra $g : B \to M(FB)$ and an algebra $h : FA \to A$ we define the **monadic hylomorphism** as the function $(\!|h, g|\!)_F : B \to MA$ given by $(\!|h, g|\!)_F = \mathsf{fix}(\lambda f.\widehat{h} \bullet \widehat{F} f \bullet g)$.

Example 5.2 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 $(\!|h, g|\!)_{L_A} : B \to MC$ is given by:

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

Inlining:

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

Let us now discuss some properties. First of all, observe that monadic anamorphism is a particular case of monadic hylomorphism: $[\![(g)]\!]^M_F = (\!|in_F, g|\!)_F$. On the other hand, in the special case that the monadic coalgebra is given by

$\widehat{g}$ for $g : B \to FB$, the monadic hylomorphism reduces to a (plain) hylomorphism as the following calculation shows.

$$\langle\!|h, \widehat{g}|\!\rangle_F = \mathsf{fix}(\lambda f.\widehat{h^\star} \circ (\widehat{F}f)^\star \circ \widehat{g}) = \mathsf{fix}(\lambda f.\widehat{h^\star} \circ \delta^F \circ Ff \circ g) = [\![\widehat{h} \bullet \delta^F, g]\!]_F$$

So, in particular, $(\!|\widehat{h} \bullet \delta^F_A|\!)_F = \langle\!|h, \widehat{out_F}|\!\rangle_F$. However, in general, $(\!|\widehat{h} \bullet \delta^F_A|\!)_F \neq \widehat{(\!|h|\!)_F}$ and thus $\langle\!|in_F, \widehat{out_F}|\!\rangle_F = [\![\widehat{out_F}]\!]^M_F \neq \mathsf{unit}$. They become equalities only when unit is strict. (Of course, they are automatically valid in $\mathbf{Cpo}_\perp$.)

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.

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

Thus, we can transform any composition of this kind into a monolithic 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. In the next section, we shall see that 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 factorization is proved by the following proposition.

Proposition 5.3 *Suppose that functor M is locally continuous. Let $h : FA \to A$ be an algebra and $g : B \to M(FB)$ a monadic coalgebra. Then, $\langle\!|h, g|\!\rangle_F = M(\!|h|\!)_F \circ [\![g]\!]^M_F$.*

Proof. By fixpoint induction with predicate $P(f, f_1, f_2) \equiv f = Mf_2 \circ f_1$. Define that $\psi(f) = \widehat{h} \bullet \widehat{F}f \bullet g$, $\chi(f) = h \circ Ff \circ out_F$, and $\phi(f) = \widehat{in_F} \bullet \widehat{F}f \bullet g$.

Base Case: We have to prove that:

$$B \xrightarrow{\perp} MA \;=\; B \xrightarrow{\perp} M\mu F \xrightarrow{M\perp} MA$$

As mentioned earlier, every locally continuous functor M is strictness-preserving (see [6]). Hence, $M\perp : M\mu F \to MA$ is a strict function and its composition with $\perp : B \to M\mu F$ is the bottom morphism $\perp : B \to MA$.

Inductive Case: Assume that $f = Mf_2 \circ f_1$. Recall that, for every f, $\widehat{f^\star} = Mf$. Thus, $M\chi(f_2) \circ \phi(f_1) = Mh \circ M(Ff_2) \circ Mout_F \circ Min_F \circ (\widehat{F}f_1)^\star \circ g$. Observe that $Mout_F \circ Min_F = M(out_F \circ in_F) = \mathsf{id}$. Also, $M(Ff_2) \circ (\widehat{F}f_1)^\star = (M(Ff_2) \circ \widehat{F}f_1)^\star$. By naturality of δ^F we get that $M(Ff_2) \circ \widehat{F}f_1 = \widehat{F}(Mf_2 \circ f_1)$ and by induction hypothesis $\widehat{F}(Mf_2 \circ f_1) = \widehat{F}f$. Summing up, we have that $M\chi(f_2) \circ \phi(f_1) = Mh \circ (\widehat{F}f)^\star \circ g$ which is equal to $\psi(f)$.

Therefore, it follows that $\mathsf{fix}(\psi) = M\,\mathsf{fix}(\chi) \circ \mathsf{fix}(\phi)$. □

Fusion laws can be established in combination with algebra and monadic coalgebra mappings. For algebras h and h' and monadic coalgebras g and g',

(19) f strict $\wedge$ $f \circ h = h' \circ Ff$ $\Rightarrow$ $\widehat{f} \bullet (\!|h, g|\!)_F = (\!|h', g|\!)_F$

(20) $g \circ f = M(Ff) \circ g'$ $\Rightarrow$ $(\!|h, g|\!)_F \circ f = (\!|h, g'|\!)_F$

Law (19) holds provided that M is locally continuous.

To conclude, we present a kind of Acid Rain Theorem for this form of monadic hylomorphism. We will say that $\boldsymbol{T} : (A \to FA) \to (A \to MGA)$ is a *monadic coalgebra transformer* whenever for every $f : A \to B$, and coalgebras $g : A \to FA$ and $g' : B \to FB$, it holds that if $g' \circ f = Ff \circ g$ then $\boldsymbol{T}(g') \circ f = MGf \circ \boldsymbol{T}(g)$.

Theorem 5.4 (Acid Rain)

Cata-mhylo fusion: *Let $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ be an algebra transformer and let M be locally continuous. For $h : FA \to A$ strict and $g : B \to MGB$,*

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

Mhylo-ana fusion: *Let $\boldsymbol{T} : (A \to FA) \to (A \to MGA)$ be a monadic coalgebra transformer. For $h : GA \to A$ and $g : B \to FB$,*

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

Proof. The proof of cata-mhylo-fusion is similar to that of cata-hylo-fusion (see Theorem 2.2), but relying on the application of law (19) instead. The proof of mhylo-ana-fusion is as follows. By definition of monadic coalgebra transformer we get that $[\!(g)\!]_F : B \to \mu F$ is a pure cohomomorphism between the monadic G-coalgebras $\boldsymbol{T}(g) : B \to MGB$ and $\boldsymbol{T}(out_F) : \mu F \to MG\mu F$. Therefore, by applying law (20) we arrive at the desired result. □

An application of cata-mhylo-fusion will be mentioned in the presentation of monadic parsers in Subsection 6.2. An application of mhylo-ana-fusion is sketched in the following example.

Example 5.5 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. It is also possible to give a generic (or polytypic [9]) version of zip, called $\mathsf{pzip} : DA \times DB \to M(D(A \times B))$, which zips two terms of type DA and DB, resp. (D is induced by bifunctor F.) M is the maybe monad; its presence 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 $\mathsf{pzip} = [\!(\mathsf{pzstep})\!]^M_{F_{A\times B}}$ for $\mathsf{pzstep} = \mathsf{fzip} \circ (out_{F_A} \times out_{F_B})$, where $\mathsf{fzip} : F(A, X) \times F(B, Y) \to M(F_{A\times B}(X \times Y))$ is certain natural transformation defined by induction on the structure of bifunctor F (see [9,23] for a definition). Now suppose that the data structures are generated, resp., by anamorphisms $[\!(g_1)\!]_{F_A} : X \to DA$ and $[\!(g_2)\!]_{F_B} : Y \to DB$. Consider the

straightforward generalization of mhylo-ana-fusion for the case of simultaneously generated data structures:

$$\langle\!|h,\, \boldsymbol{T}(out_{F_1}, out_{F_2})|\!\rangle_G \circ ([\![g_1]\!]_{F_1} \times [\![g_2]\!]_{F_2}) = \langle\!|h,\, \boldsymbol{T}(g_1, g_2)|\!\rangle_G$$

Recall that $[\![\mathsf{pzstep}]\!]^M = \langle\!|in_F, \mathsf{pzstep}|\!\rangle$. By defining $\boldsymbol{T}(g_1, g_2) = \mathsf{fzip} \circ (g_1 \times g_2)$ we can write that $\mathsf{pzstep} = \boldsymbol{T}(out_{F_A}, out_{F_B})$. Therefore, by applying the generalized fusion law

$$\mathsf{pzip} \circ ([\![g_1]\!] \times [\![g_2]\!]) = [\![\mathsf{fzip} \circ (g_1 \times g_2)]\!]^M_{F_{A\times B}} : X \times Y \to M(D(A \times B))$$

we eliminate the generation of the two intermediate data structures.

6 Applications

The aim of this section is to illustrate the use of the monadic functionals and some of their calculational laws. The first example describes a novel formulation of graph traversal algorithms (such as DFS or BFS) based on monadic corecursion. The second one deals with monadic parsing and shows that every monadic parser can be expressed as a monadic hylomorphism.

6.1 Graph Traversals

By *graph traversals* we understand functions that take a list of roots (entry points to a graph) and return a list containing the vertices met along the way. For the formulation of such a class of functions we consider a graph representation that we establish now. Recall that a *directed graph* is a pair $G = (V, E)$ where V is the set of *vertices* and $E \subseteq V \times V$ is the set of *arcs* of the graph. Two vertices are said to be *adjacent* if there is an arc connecting them. There are several ways in which a graph can be represented in order to compute with it. Two standard ways are adjacency matrices and adjacency lists. The representation we consider is close to the latter. Indeed, we will assume that a graph is given by an *adjacency list function* $\mathsf{adj} : V \to V^*$ which for each vertex returns its adjacency list. This gives a sufficiently abstract representation that at the same time is useful for algorithmic purposes.

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. Thus, 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:

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

Our aim is to construct a monadic corecursive formulation of graph traversal in which the set of visited nodes is manipulated within the state monad.

Operationally speaking, a reason for using the state monad might be because we want to consider an "imperative" representation for sets. For example, 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* (i.e by an array with destructive updates). In that case the set operations represent primitives that operate over the array directly. To be able to handle these imperative operations in a functional setting a well-established technique is to encapsulate them in a monadic ADT based on the state monad [29]:

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

which in addition to unit and $\star$ possesses the operations:

$$\emptyset^M : MA \to A \qquad \uplus^M : V \to M1 \qquad \lhd^M : V \to M\,\mathsf{Bool}$$

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 [25,29]). 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).[6]

Based on this ADT, we can now 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 the following monadic anamorphism:

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

Operationally speaking, given an initial list of roots vs, $\mathsf{graphtrav}$ first allocates an empty set, then applies $\mathsf{gtrav}(vs)$ to it yielding a list of vertices and a final state of the set, and finally de-allocates the set and returns the list.

In each iteration, the action of the monadic coalgebra gopen begins with an exploration of the current list of roots vs in order to find an element in it that had not been reached before. To this end it removes from the front of vs all vertices u that qualify as visited (i.e. those for which $u \lhd s = \mathsf{true}$) until either an unvisited vertex is met or the end of the list is reached. This task is performed by function $\mathsf{mdropS} : V^* \to MV^*$ defined by:

$$\begin{aligned} \mathsf{mdropS}(\mathsf{nil}) &= \mathsf{unit}(\mathsf{nil}) \\ \mathsf{mdropS}(\mathsf{cons}(v, vs)) &= \lhd^M(v) \star \lambda b.\,\mathbf{if}\ b\ \mathbf{then}\ \mathsf{mdropS}(vs) \\ &\qquad\qquad \mathbf{else}\ \mathsf{unit}(\mathsf{cons}(v, vs)) \end{aligned}$$

[6] Note that the introduction of the monadic ADT also makes sense in case we consider a pure functional representation of sets. We have made reference to the imperative solution only for the sake of illustration.

Once mdropS was applied, we then proceed to visit the vertex (if any) at the head of the input list and to mark it (inserting it in the set). A new 'state' of the list of roots is also computed. For this we use a function, called $policy : V \times V^* \to V^*$, which encapsulates the administration policy utilized for the list of roots. In this form we can achieve a formulation parametric in the strategy followed by the traversal. In summary,

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

By definition of monadic anamorphism,

$$\mathsf{gtrav} = \mathsf{fix}(\lambda f.[\widehat{\mathsf{nil}}, \widehat{\mathsf{cons}}^\star \circ \tau \circ (\mathsf{id}_V \times f)]^\star \circ \mathsf{gopen})$$

Thus, inlining we have:

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

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, at each stage, after dropping from the front of the stack all visited vertices with mdropS, the top v is removed and replaced by its adjacency list $\mathsf{adj}(v)$. That is,

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

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

On the other hand, 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. Now, at each stage, after dropping visited vertices with mdropS, the front v of the queue is removed and its adjacency list $\mathsf{adj}(v)$ concatenated at the end of the queue. That is,

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

In this case let us call bf to the resulting instance of gtrav.

Representation Change.

As we have just seen, a breadth-first traversal manages the list of roots as a queue. However, operationally speaking, it is well-known that using a list for representing a queue is quite inefficient. In fact, an element is enqueued

by appending it at the end of the list, and this takes time proportional to the length of the list. To eliminate this inefficiency we will apply the *pure-fusion law* shown earlier to transform function bf into an equivalent monadic anamorphism that makes use of a better queue representation.

Suppose we are given an ADT $\mathcal{Q}(A)$ of queues over A, which comes equipped with these operations: $\mathsf{empty} : \mathcal{Q}(A)$ (the empty queue), $\mathsf{enq} : A \times \mathcal{Q}(A) \to \mathcal{Q}(A)$ (inserts a new element), $\mathsf{front} : \mathcal{Q}(A) \to A$ (returns the front element), $\mathsf{isnull} : \mathcal{Q}(A) \to \mathsf{Bool}$ (tests whether a queue is empty), and $\mathsf{deq} : \mathcal{Q}(A) \to \mathcal{Q}(A)$ (removes the front element). Using this ADT we organize the list of roots waiting for attention as a pair $(vs, q) \in V^* \times \mathcal{Q}(V^*)$, such that vs is the adjacency list being currently attended and q is a queue containing adjacency lists waiting for activation. When the list vs empties, a new list is then taken from the queue q. With this new representation we construct a new monadic coalgebra

$$\mathsf{qopen} : V^* \times \mathcal{Q}(V^*) \to M(1 + V \times (V^* \times \mathcal{Q}(V^*)))$$

defined as follows:

$$\begin{array}{l}
\mathsf{qopen}(\ell, q) = \mathsf{mdropS}(\ell) \star \lambda \ell'. \\
\quad \textbf{case } \ell' \textbf{ of} \\
\quad\quad \begin{array}{lll}
\mathsf{nil} & \to & \textbf{if } \mathsf{isnull}(q) \textbf{ then } \mathsf{unit}(\mathsf{inl}(\bot)) \\
 & & \textbf{else } \mathsf{qopen}(\mathsf{front}(q), \mathsf{deq}(q)) \\
\mathsf{cons}(v, vs) & \to & \uplus^M(v) \star \lambda x. \\
 & & \mathsf{unit}(\mathsf{inr}(v, (vs, \mathsf{enq}(\mathsf{adj}(v), q))))
\end{array}
\end{array}$$

Now, consider the function $\mathsf{change} : V^* \times \mathcal{Q}(V^*) \to V^*$ defined as

$$\mathsf{change}(\ell, q) = \ell +\!\!+ \mathsf{q2list}(q)$$

where q2list maps a queue $\langle \ell_1, \ldots, \ell_n \rangle$ to a list $\ell_1 +\!\!+ \cdots +\!\!+ \ell_n$. It is not hard to see that change is a pure cohomomorphism between the monadic coalgebras qopen and gopen. Therefore, if we now apply pure-fusion (equation (16)), then we achieve an improved version of the breadth-first traversal given also by a monadic anamorphism:

$$V^* \times \mathcal{Q}(V^*) \xrightarrow{\mathsf{change}} V^* \xrightarrow{\mathsf{bf}} MV^* \quad \rightsquigarrow \quad [\![(\mathsf{qopen})]\!]^M_{L_V}$$

Search Procedures

One might be interested in performing some calculation with the outcome of a graph traversal. For example, to apply a catamorphism $(\!|h|\!)_{L_V}$ to it.

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

But recall that, $\mathsf{graphtrav} = \emptyset^M \circ \mathsf{gtrav}$. So, by applying naturality of $\emptyset^M$, i.e.

$$f \circ \emptyset^M = \emptyset^M \circ Mf$$

for every f, we can push the catamorphism inside 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 $\mathsf{filter}(p) = (\!|\mathsf{fil}|\!)_{L_V} : V^* \to V^*$ that removes the elements of a list that do not satisfy a prediate $p : V \to \mathsf{Bool}$, where

$$\mathsf{fil}(x) = \begin{cases} \mathsf{nil} & \text{if} \quad x = \mathsf{inl}(\bot) \\ \mathsf{cons}(v, \ell) & \text{if} \quad x = \mathsf{inr}(v, \ell) \text{ and } p(v) \\ \ell & \text{if} \quad x = \mathsf{inr}(v, \ell) \text{ and } \neg p(v) \end{cases}$$

Then,

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

represents a search procedure that explores a graph in determinate order with the aim at finding all vertices fulfilling a given predicate. Like before, by specifying a concrete traversal strategy typical search procedures such as *depth-first search* or *breadth-first search* are accomplished.

6.2 Monadic Parsing

The parsing technique called recursive descent is very popular in functional programming. By means of it a *functional parser* for a language $\mathcal{L}$ is constructed by replacing its grammar by a collection of mutually recursive functions, each corresponding to one of the syntactic categories (nonterminals) of the grammar. For each syntactic category S, the goal of the corresponding function parser_S is to analyze a sequence of input symbols —usually called tokens— that form a string in the language $\mathcal{L}(S)$, and to return some representation for the recognized string. Let T stand for the set of tokens. We consider that parsers are functions of this type:

$$\mathsf{Parser}\ A = [T^* \to (A \times T^*)^*]$$

That is, a parser takes a string of tokens and yields a list with all the alternative manners in which the input string can be parsed. A parser may either fail or succeed to recognize a given string. Failure is represented by the empty list of results, meaning that there is no way to parse the input string. Each alternative parsing is composed by a value of type A, representing the parsed input, together with the remaining unparsed suffix of the input string. In the parser literature, the outcome of a parser is usually given by a parse tree, which describes the structure of the recognized string. However, functional parsers are typically presented as functions that return values of any kind.

The reason why parsers return the remaining unprocessed string is because they may call other parsers or themselves recursively in order to parse

substructures. The body of a grammar production $S ::= X_1 \cdots X_n$, where each X_i is either a terminal or a syntactic category, can be thought of as a sequence of goals that must be fulfilled in order to deduce that an input string belongs to the syntactic category S. This sequence of goals is resolved by calling the respective parser function for each X_i in the order they appear and then composing them with help of a combinator for *sequencing*. A goal X_i corresponding to a terminal is satisfied only if this terminal is just the next symbol in the input string. This task is performed by the elementary parser $\mathsf{tok} : T \to \mathsf{Parser}\ T$,

$$\mathsf{tok}(t)(\mathsf{nil}) = \mathsf{nil} \qquad \mathsf{tok}(t)(\mathsf{cons}(t', s')) = \textbf{if } t = t' \textbf{ then } [(t, s')] \textbf{ else } \mathsf{nil}$$

where the notation $[x]$ stands for the singleton list $\mathsf{cons}(x, \mathsf{nil})$. Grammars usually have various alternative productions for each syntactic category, i.e. $S ::= X_1 \cdots X_m | \cdots | Y_1 \cdots Y_n$. In a functional parser the choice for which production to apply is represented by a combinator for *alternation*.

This amounts to see that the logical structure of a functional parser is given by the context-free grammar of the language. In fact, just like grammars in BNF notation, we can build up parsers from other parsers by using combinators such as sequencing and alternation. The aim of the present example is to give a formal explanation of this fact with help of the monadic recursive functionals introduced in previous sections. Our ultimate goal is to give a formal characterization of the recursive structure of recursive descent parsers. To the best of our knowledge, this the first attempt to characterize the structure of recursive descent parsers.

As Wadler observed [28,29], functional parsers can be structured using the so-called *parser monad*. In the monadic approach, the combinators for sequencing and alternation are given as primitive operations which permits to focus on the relevant structure of parsers. This fact will help us to clearly identify the two phases that actually conform the definition of a functional parser, namely (i) *syntax analysis*, by which a string is recognized and a parse tree generated, and (ii) the application of *semantic actions*, by which a value (outcome of the parser) is calculated from the just produced parse tree. (This in turn raises a connection between functional parsers and *attribute grammars*.)

Following, we briefly summarize the main results we achieve. We recognize that the syntax analysis phase of a parser can be expressed as a monadic corecursive function on the datatype representing the *concrete syntax* (i.e. the datatype of parse trees). Joining this fact with the fact that semantic actions are usually defined by induction over the structure of parse trees (i.e. they correspond to a *catamorphism*), we obtain that the application of semantics actions after a syntax analyzer yields a monadic hylomorphism, avoiding therefore the generation and immediate consumption of parse trees, and such that it corresponds to a typical functional parser.

The Parser Monad

According to the type definition given above, parsers can be regarded as a kind of state transformers whose state is represented by the input string of tokens. The definition of the parser monad [28,29,18] is the following:

$$MA = \mathsf{Parser}\ A \qquad \mathsf{unit}(a) = \lambda s.\,[(a, s)] \qquad p \star f = \mathsf{concat} \circ \mathsf{List}(\bar{f}) \circ p$$

where $\bar{f}$ denotes the uncurry of $f : A \to MB$, i.e. $\bar{f}(x, y) = f(x)(y)$, and $\mathsf{concat} = (\!|[\mathsf{nil}, +\!\!+]|\!)_{L_A}$ is the traditional function that flattens a list of list into a list [1]. For each $a \in A$, the parser $\mathsf{unit}(a)$ does not consume any input, and always succeeds returning a. The bind operator $\star$ corresponds to the combinator for *sequencing*. Given an input string s, a parser $p \star f$ first applies parser p to s, yielding a list of alternative parsings $[(a_1, s_1), \ldots, (a_n, s_n)]$. Each parsing (a_i, s_i) is then mapped with $\bar{f}$, resulting a new list $[f(a_1)(s_1), \ldots, f(a_n)(s_n)]$ each of its elements is itself a list of parsings. Finally, these lists are joined together into a list $f(a_1)(s_1) +\!\!+ \cdots +\!\!+ f(a_n)(s_n)$ by concat.

The parser monad is a special case of a *monad with a zero and a plus* [19,18]. That is, it is a monad equipped with $\mathsf{zero} : MA$ and $\oplus : MA \times MA \to MA$ such that, for each type A, the triple $(MA, \oplus, \mathsf{zero})$ forms a monoid structure: $\mathsf{zero} \oplus p = p$, $p \oplus \mathsf{zero} = p$ and $p \oplus (q \oplus r) = (p \oplus q) \oplus r$. For the parser monad,

$$\mathsf{zero} = \lambda s.\,\mathsf{nil} \qquad\qquad p \oplus q = \lambda s.\,p(s) +\!\!+ q(s)$$

The parser zero fails for any input. The operator $\oplus$ corresponds to the combinator for *alternation*. For a string s, the parser $p \oplus q$ applies both p and q to s and appends all parsings yielded by them. The parser zero is indeed a zero of $\star$: $\mathsf{zero} \star f = \mathsf{zero}$ and $p \star \lambda a.\,\mathsf{zero} = \mathsf{zero}$. In addition, $\star$ distributes through $\oplus$ on the left: $(p \oplus q) \star f = (p \star f) \oplus (q \star f)$. In terms of the Kleisli star this says that: $f^\star \circ \oplus = \oplus \circ (f^\star \times f^\star)$.

The following two parsers will be useful later.

$$\mathsf{item}(s) = \begin{cases} \mathsf{nil} & \text{if } s = \mathsf{nil} \\ [(t, s')] & \text{if } s = \mathsf{cons}(t, s') \end{cases} \qquad\qquad q \rhd p = q \star \lambda a.\ \textbf{if } p(a) \textbf{ then } \mathsf{unit}(a) \textbf{ else } \mathsf{zero}$$

The parser $\mathsf{item} : MT$ returns the first token in the input string and fails if the input is empty. By means of the operator $\rhd : MA \times (A \to \mathsf{Bool}) \to MA$ we can filter the results of a parser with a predicate.

Our running example will be a simple language of arithmetic expressions with this *concrete syntax* specification:

$$\begin{aligned} \mathit{exp} &::= \mathit{term}\ \text{`+'}\ \mathit{exp} \mid \mathit{term} \\ \mathit{term} &::= \mathit{factor}\ \text{`*'}\ \mathit{term} \mid \mathit{factor} \\ \mathit{factor} &::= \text{`('}\ \mathit{exp}\ \text{`)'} \mid \mathit{numeral} \end{aligned}$$

For the sake of simplicity, let us assume that each numeral $\underline{n}$ comes given by the natural number n it represents. The set T of terminal symbols corresponding

to this grammar is thus defined as $T = \{\text{`('}, \text{`)'}, \text{`+'}, \text{`*'}\} \cup \mathbb{N}$. From the grammar we can construct the following monadic parser, which recognizes an expression and returns the natural number that arises from evaluating it. Let $\mathsf{add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $\mathsf{prod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

$$\mathsf{parser}_e \ : \ M\mathbb{N}$$
$$\mathsf{parser}_e = (\mathsf{parser}_t \star \lambda t.\, \mathsf{tok}(\text{`+'}) \star \lambda a.\, \mathsf{parser}_e \star \lambda e.\, \mathsf{unit}(\mathsf{add}(t, e))) \oplus \mathsf{parser}_t$$

$$\mathsf{parser}_t \ : \ M\mathbb{N}$$
$$\mathsf{parser}_t = (\mathsf{parser}_f \star \lambda f.\, \mathsf{tok}(\text{`*'}) \star \lambda p.\, \mathsf{parser}_t \star \lambda t.\, \mathsf{unit}(\mathsf{prod}(f, t))) \oplus \mathsf{parser}_f$$

$$\mathsf{parser}_f \ : \ M\mathbb{N}$$
$$\mathsf{parser}_f = (\mathsf{tok}(\text{`('}) \star \lambda l.\, \mathsf{parser}_e \star \lambda e.\, \mathsf{tok}(\text{`)'}) \star \lambda r.\, \mathsf{unit}(e)) \oplus (\mathsf{item} \rhd (\in \mathbb{N}))$$

Syntax Analysis

The technique to be described here is completely general, as it can be used for any context-free language. Our exposition, however, will essentially focus on the language of arithmetic expressions presented above.

By *syntax analysis* we understand the process by which strings of tokens are recognized and returned in the form of parse trees. To construct a syntax analyzer for a language, we need to give a datatype representation for the concrete syntax, as it specifies the definition of parse trees. For the language of arithmetic expressions these are the datatype declarations:

$$\begin{array}{ll} \mathsf{Exp} & = \mathsf{addop}(\mathsf{Term}, \mathsf{Plus}, \mathsf{Exp}) \mid \mathsf{term}\ \mathsf{Term} \\ \mathsf{Term} & = \mathsf{prodop}(\mathsf{Factor}, \mathsf{Mult}, \mathsf{Term}) \mid \mathsf{factor}\ \mathsf{Factor} \\ \mathsf{Factor} & = \mathsf{par}(\mathsf{Left}, \mathsf{Exp}, \mathsf{Right}) \mid \mathsf{num}\ \mathbb{N} \end{array} \qquad \begin{array}{l} \mathsf{Plus} = \mathsf{plus} \\ \mathsf{Mult} = \mathsf{mult} \\ \mathsf{Left} = \mathsf{left} \\ \mathsf{Right} = \mathsf{right} \end{array}$$

The reason for introducing datatypes Plus, Mult, Left and Right for the terminals {'(','),'+','*'} is because parse trees structurally represent *all* details of recognized strings, inclusive terminal symbols. As we shall see, the presence of these datatypes for terminals turns out to be determinant for achieving a corecursive formulation of the syntax analyzer, as they force the occurrence of calls to the parsers for the terminals exactly in the places they are required.

A syntax analyzer for a language is composed by one function for each syntactic category and each terminal. Each of these functions is given by a monadic parser that yields parse trees of the corresponding type. For instance, $\mathsf{syntax}_e : M\mathsf{Exp}$. However, the trick we will use to achieve a corecursive formulation of the syntax analyzer consists of regarding these functions as functions from the unit type. So, for example, $\mathsf{syntax}_e : 1 \to M\mathsf{Exp}$.[7] The (recursive) behaviour of these component functions is guided by the (recursive) structure

[7] The usefulness of considering the unit type will become clear later.

of the type of parse trees they construct. Indeed, each of them can be expressed as a monadic anamorphism. Moreover, they are mutually recursive, since so are the parse tree types. Consequently, this makes altogether seven mutually-recursive monadic anamorphisms.

To build the monadic anamorphisms we need to identify the functors that capture the signature of the parse tree types. Recall that when various datatypes are defined by simultaneous recursion, their functors reflect this fact by having so many variables as involved datatypes (see e.g. [4, Section 3d]). In the special case of arithmetic expressions, the functors are on seven variables. Let us write $\vec{A} = (A_e, A_t, A_f, A_p, A_m, A_l, A_r)$ for short. Then,

$$F_e\,\vec{A} = A_t \times A_p \times A_e + A_t \qquad F_p\,\vec{A} = 1 \qquad F_l\,\vec{A} = 1$$
$$F_t\,\vec{A} = A_f \times A_m \times A_t + A_f \qquad F_m\,\vec{A} = 1 \qquad F_r\,\vec{A} = 1$$
$$F_f\,\vec{A} = A_l \times A_e \times A_r + \mathbb{N}$$

Of course, the datatypes Plus, Times, Left, and Right are neither recursive nor depend on other types. However, if we consider them as part of the simultaneous recursion definition, then we can assign them a variable position in all functors. Their presence as variables rather than as constant types within functors enables us to (automatically) force calls to their corresponding syntax analysis functions in the corecursive formulation. Obviously, the functions corresponding to these non-recursive types are also non-recursive, but even though they are indeed monadic anamorphisms. Omitting the unit type,

$$\mathsf{syntax}_p = \mathsf{tok}(\text{`+'}) \star \lambda p.\,\mathsf{unit}(\mathsf{plus}) \qquad \mathsf{syntax}_l = \mathsf{tok}(\text{`('}) \star \lambda l.\,\mathsf{unit}(\mathsf{left})$$
$$\mathsf{syntax}_m = \mathsf{tok}(\text{`*'}) \star \lambda m.\,\mathsf{unit}(\mathsf{mult}) \qquad \mathsf{syntax}_r = \mathsf{tok}(\text{`)'}) \star \lambda r.\,\mathsf{unit}(\mathsf{right})$$

Now we address the definition of the three recursive analyzers. We begin with the analyzer for whole expressions syntax_e. It can be expressed as a monadic anamorphism $[\![(g_e)]\!]^M_{F_e}$ on certain coalgebra g_e. This means that it satisfies the following diagram:

$$\begin{array}{ccc} 1 & \xrightarrow{\quad\mathsf{syntax}_e\quad} & M\mathsf{Exp} \\ {\scriptstyle g_e}\downarrow & & \uparrow{\scriptstyle M[\mathsf{addop},\mathsf{term}]} \\ M(F_e(1,\ldots,1)) & \xrightarrow[(\widehat{F}_e\,\mathsf{syntax})^\star]{} & M(F_e(\mathsf{Exp},\ldots,\mathsf{Right})) \end{array}$$

where syntax stands for the tuple of the seven functions $(\mathsf{syntax}_e, \ldots, \mathsf{syntax}_r)$. Observe that by definition of F_e, $F_e(1,\ldots,1) = 1 \times 1 \times 1 + 1$, and

$$F_e(\mathsf{Exp},\ldots,\mathsf{Right}) = \mathsf{Term} \times \mathsf{Plus} \times \mathsf{Exp} + \mathsf{Term}$$

Also,

$$\widehat{F}_e(f_1,\ldots,f_7) = [\widehat{\mathsf{inl}} \bullet \psi^{(3)} \circ (f_2 \times f_4 \times f_1), \widehat{\mathsf{inr}} \bullet f_2]$$

for $\psi^{(3)} : MA \times MB \times MC \to M(A \times B \times C)$. Consequently,

$$M[\mathsf{addop}, \mathsf{term}] \circ \widehat{F}_e(f_1, \ldots, f_7) = [\widehat{\mathsf{addop}} \bullet \psi^{(3)} \circ (f_2 \times f_4 \times f_1), \widehat{\mathsf{term}} \bullet f_2]$$

The monadic coalgebra is given by

$$g_e = \oplus \circ \langle \widehat{\mathsf{inl}} \circ \Delta_3, \widehat{\mathsf{inr}} \rangle$$

where $\Delta_3 = \langle \mathsf{id}, \mathsf{id}, \mathsf{id} \rangle$. That is, $g_e(\bot) = \mathsf{unit}(\mathsf{inl}(\bot, \bot, \bot)) \oplus \mathsf{unit}(\mathsf{inr}(\bot))$. The carrier of the monadic coalgebra represents a notion of *control* in this case. The occurrences of 1 in the type $1 \times 1 \times 1 + 1$ are used to indicate the positions where the recursive computation has to proceed. These positions represent nothing more that the parsing *goals* of the productions. The product of 1's models the fact that the respective parsing goals will be sequenced. Note also how the existence of two alternative productions for *exp* is modeled by the occurrence of $\oplus$ in g_e. Using the fact that $\star$ distributes through $\oplus$ on the left, we can perform the following calculation for an arbitrary $h = [h_1, h_2]$,

$$h^\star \circ g_e = \oplus \circ (h^\star \times h^\star) \circ \langle \widehat{\mathsf{inl}} \circ \Delta_3, \widehat{\mathsf{inr}} \rangle = \oplus \circ \langle h_1 \circ \Delta_3, h_2 \rangle$$

In summary, we have:

$$\mathsf{syntax}_e = \oplus \circ \langle \widehat{\mathsf{addop}} \bullet \psi^{(3)} \circ \langle \mathsf{syntax}_t, \mathsf{syntax}_p, \mathsf{syntax}_e \rangle, \widehat{\mathsf{term}} \bullet \mathsf{syntax}_t \rangle$$

Omitting the applications to the unit type,

$$\begin{aligned} \mathsf{syntax}_e = {} & \mathsf{syntax}_t \star \lambda t.\, \mathsf{syntax}_p \star \lambda a.\, \mathsf{syntax}_e \star \lambda e.\, \mathsf{unit}(\mathsf{addop}(t, a, e)) \\ & \oplus \mathsf{syntax}_t \star \lambda t.\, \mathsf{unit}(\mathsf{term}(t)) \end{aligned}$$

coinciding with the analyzer one would have directly written by hand.

Likewise, the analyzers for terms and factors, syntax_t and syntax_f, can be expressed as monadic anamorphisms $[\![g_t]\!]^M_{F_t}$ and $[\![g_f]\!]^M_{F_f}$, respectively, where

$$g_t = g_e \qquad \text{and} \qquad g_f = \oplus \circ \langle \widehat{\mathsf{inl}} \circ \Delta_3, \mathsf{item} \rhd (\in \mathbb{N}) \rangle$$

Like above, by formal manipulation we can deduce that:

$$\mathsf{syntax}_t = \oplus \circ \langle \widehat{\mathsf{prodop}} \bullet \psi^{(3)} \circ \langle \mathsf{syntax}_f, \mathsf{syntax}_m, \mathsf{syntax}_t \rangle, \widehat{\mathsf{factor}} \bullet \mathsf{syntax}_f \rangle$$

$$\mathsf{syntax}_f = \oplus \circ \langle \widehat{\mathsf{par}} \bullet \psi^{(3)} \circ \langle \mathsf{syntax}_l, \mathsf{syntax}_e, \mathsf{syntax}_r \rangle, \widehat{\mathsf{num}} \bullet (\mathsf{item} \rhd (\in \mathbb{N})) \rangle$$

Adding Semantic Actions

Suppose that now we want to incorporate semantic actions to a parser, in the sense that we want to compute values from the parse trees generated by a syntax analyzer. In parsing theory this typically corresponds to the association of attributes with each grammar symbol, and semantic rules with each production to compute with the attributes. In our setting, this can be regarded as the definition of a catamorphism. The incorporation of semantic

actions to a syntax analyzer is thus captured by the following composite:

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

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:

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

Like for the syntax analyzer, it is worth noting that, when simplified, the expression of this monadic hylomorphism coincides with that of the monadic parser one would have written by hand. This equation can also be interpreted as stating the following result:

Theorem 6.1 *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.*

Now we can gather the benefits of having structured the syntax analyzer as a monadic anamorphism. First of all, 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 monolithic 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. now they can be the subject of fusion transformations—something that was impossible up to now. 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 [15], the semantic actions of an interpreter / compiler 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 transformer $\boldsymbol{T} : (FA \to A) \to (GA \to A)$. The semantic actions are then actually given on the abstract syntax, i.e. $\mathsf{sem} = (\!|h|\!)_F : \mu F \to A$. As a result we have the following composite:

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

Because $M\mathsf{conv} \circ \mathsf{syntax} = \langle\!|\, \boldsymbol{T}(in_F), g|\!\rangle_G$, we can apply cata-mhylo-fusion (see

Theorem 5.4) 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.

7 Conclusions

This paper investigated two new monadic recursive functionals, whose transformation laws permit to achieve new deforestation cases within monads. The examples presented aimed at showing that these functionals capture definitions that commonly appear in programming practice.

An interesting issue to be investigated is the possibility to integrate monadic hylomorphism and its transformation laws as part of a calculational-based transformation system like is the system HYLO [22]. Roughly speaking, the system HYLO considers hylomorphism as the standard pattern of recursive function definition within programs and automatically transforms programs by applying fusion laws. Concretely, our proposal is to investigate the development of a monadic extension of such a system that considers monadic hylomorphism as the standard form of recursion and applies its transformation laws. Such an extension would naturally embed the resolution of 'pure' cases of deforestation as a special instance —i.e. those that HYLO resolve— since observe that when the underlying monad is the *identity monad*, our monadic recursive functionals reduce to the standard 'pure' ones.

Acknowledgement

This work has been partially supported by a german DAAD scholarship. Diagrams were constructed using Paul Taylor's diagrams package.

References

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

[2] R.S. Bird and O. de Moor. Relational program derivation and context-free language recognition. In *A Classical Mind*, pages 17–35. Prentice Hall, 1994.

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

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

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

[6] P. Freyd. Recursive types reduced to inductive types. In *5th IEEE Symposium on Logic in Computer Science*, pages 498–507, 1990.

[7] 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. Also available from `http://www.ipl.t.u-tokyo.ac.jp/~hu/`.

[8] J. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, Utrecht University, 1993.

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

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

[11] D.J. Lehmann and M.B. Smith. Algebraic specification of data types. *Mathematical Systems Theory*, 14:97–139, 1981.

[12] G. Malcolm. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[13] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science. Springer-Verlag, 1986.

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

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

[16] 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.

[17] 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.

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

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] D. A. Schmidt. *Denotational Semantics. A Methodology for Language Development.* Allyn and Bacon, Inc., Boston Mass., 1986.

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

[27] D. Tuijnman. *A Categorical Approach to Functional Programming.* PhD thesis, Fakultät für Informatik, Universität Ulm, Germany, Januar 1996.

[28] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

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