Fusion of Recursive Programs with Computational Effects

Alberto Pardo

Instituto de Computación
Universidad de la República
Montevideo, Uruguay

pardo@fing.edu.uy

Abstract

Fusion laws permit to eliminate various of the intermediate data structures that are created in function compositions. The fusion laws associated with the traditional recursive operators on datatypes cannot in general be used to transform recursive programs with effects. Motivated by this fact, this paper addresses the definition of two recursive operators on datatypes that capture functional programs with effects. Effects are assumed to be modeled by monads. The main goal is thus the derivation of fusion laws for the new operators. One of the new operators is called *monadic unfold*. It captures programs (with effects) that generate a data structure in a standard way. The other operator is called *monadic hylomorphism*, and corresponds to programs formed by the composition of a monadic unfold followed by a function defined by structural induction on the data structure that the monadic unfold generates.

1 Introduction

A common approach to program design in functional programming is the use of recursive operators on datatypes as a tool for structuring programs [5, 4]. Typical examples of such operators include: map, fold, unfold and primitive recursion. Recursive operators on datatypes encapsulate common patterns of computation and structure programs according to the data structures they traverse or generate. A key feature of these operators is that they can be derived from the categorical interpretation of recursive types. The categorical approach permits then to derivate algebraic laws for the operators, and provides a smooth framework for conducting calculations [20, 23, 7, 16, 5].

Some of the algebraic laws are useful for *deforestation* [37, 35]. These are the so-called *fusion laws* [23]. Deforestation is a program transformation technique by means of which intermediate data structures can be eliminated. In functional programming, the interest in that technique is due to the fact that programs are often constructed as a composition of smaller ones, using intermediate data structures to connect the subprograms. Such compositional style of programming is suitable for program modularization. However, programs developed using that style may be inefficient both in time and space, mainly because of the proliferation of intermediate data structures caused by the use of function composition.

Another way of structuring functional programs is by the *effects* they produce. This can be accomplished by using *monads* [39]. Monads capture in an unified framework a wide variety of computational effects, such as state, exceptions, or input/output. They encapsulate the action of an effect in an abstract data type M. As a result, the occurrences of the effect in a program are explicit, since they are expressed in terms of the abstract data type operations. The effect is also reflected in the type of the program, which is now of the form $A \to MB$. Monads permit

that even imperative features, like destructive update or input/output, can be integrated in a purely-functional language (see [33, 18]).

The compositional style of programming mentioned previously can still be applied to programs with effects. Concerning deforestation, however, the fusion laws for 'pure' programs cannot in general be used for programs with effects. Basically, the problem is that the programs with monadic effects have shapes of recursion that are in general not captured by the recursive operators originally thought for 'pure' programs. Motivated by this fact, in the present paper we address the definition of recursive operators that permit to represent programs with monadic effects. The main goal is then the derivation of fusion laws for such operators, since they provide cases of deforestation for that kind of programs.

Recent works [8, 12, 26] have investigated the interaction between monads and recursion for the specific case of programs that consume a data structure. The interaction in that case is captured by an operator called *monadic fold* —a version of the standard *fold* operator [23, 4]— which represents programs with effects defined by structural induction over the input data structure.

In this paper we are concerned with the dual case. That is, we investigate the class of recursive programs with effects whose structure is dictated by that of the values they produce as output. One of the contributions of the paper is then the introduction of a new operator, called *monadic unfold*, which captures the pattern of recursion of that class of programs. It is a monadic version of the standard *unfold* operator [13, 10].

An example of a function that follows that pattern of recursion is $\mathsf{getLine}$. This function, as defined in the functional language Haskell [6], reads a line from the standard input and stores its characters in a list. The effect in this case is modeled by the IO monad [33] (i.e. the monad of input/output). Computations in that monad denote actions that perform some input/output operation and return a value. Two actions m and m' can be combined in sequence as follows: $m \star \lambda x.\, m'$. The variable x binds the result from m so that it can be used by m'. (The result of the whole expression is that of m'.) The primitive action $\mathsf{getChar}$ reads a character from standard input, and then returns it. Thus, we can define $\mathsf{getLine}$ as follows:

$$\begin{array}{lcl} \mathsf{getLine} & = & \mathsf{getChar} \star \lambda c.\, \mathbf{if}\ c = \mathtt{eol} \\ & & \qquad \mathbf{then}\ \mathsf{unit}(\mathsf{nil}) \\ & & \qquad \mathbf{else}\ \mathsf{getLine} \star \lambda \ell.\, \mathsf{unit}(\mathsf{cons}(c, \ell)) \end{array}$$

An action of the form $\mathsf{unit}(x)$, for some x, performs no input/output and returns the value x. Observe how the structure of $\mathsf{getLine}$'s body is determined by the construction of the list.

In practice, one can often find programs that satisfy the following scheme: first build up a data structure with an unfold, and then, by means of a fold, compute the output value from the data contained in the structure. Functional programs of this kind are called *hylomorphisms* [23]. Another contribution of the paper is thus the introduction of a monadic version of hylomorphism, called *monadic hylomorphism*, which corresponds to the composition of a monadic unfold followed by (the monadic lifting) of a fold. That is, monadic hylomorphisms represent programs that produce some effect during the phase of generation of the intermediate data structure. Programs of this kind are not uncommon in programming practice. A nice property of monadic hylomorphism is that the two phases can be joined together into a single program which avoids the generation of the intermediate data structure.

An example of monadic hylomorphism is function $\mathsf{lenLine}$, which computes the length of a line read from standard input. It can be defined as follows:

$$\mathsf{lenLine} = \mathsf{getLine} \star \lambda \ell.\, \mathsf{unit}(\mathsf{length}(\ell))$$

where length is the function such that $\mathsf{length}(\mathsf{nil}) = 0$ and $\mathsf{length}(\mathsf{cons}(x, \ell)) = 1 + \mathsf{length}(\ell)$. A more efficient version of lenLine can then be obtained by joining together the parts.

$$\begin{array}{lcl} \mathsf{lenLine} & = & \mathsf{getChar} \star \lambda c.\, \mathbf{if}\ c = \mathtt{eol} \\ & & \quad \mathbf{then}\ \mathsf{unit}(0) \\ & & \quad \mathbf{else}\ \mathsf{lenLine} \star \lambda n.\, \mathsf{unit}(1+n) \end{array}$$

The rest of the paper is organized as follows. Section 2 reviews the categorical approach to program calculation. Section 3 presents background material on monads. In Section 4, we study the so-called monadic extension of a functor. In Section 5, we use this last concept for defining the monadic operators. First, we briefly review monadic fold. Then we introduce monadic unfold, and finally monadic hylomorphism. Section 6 deals with two non-trivial applications that can be developed using monadic unfold and hylomorphism. Finally, Section 7 gives some concluding remarks.

2 Preliminaries

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 $\mathcal{C}$. In this setting, a datatype T is understood as a solution to a type equation $X \cong FX$, for an appropriate endofunctor $F : \mathcal{C} \to \mathcal{C}$ that captures the shape (or signature) of the type. This section reviews the relevant concepts concerning the categorical approach to recursive datatypes [19, 21, 15] and its application to program calculation [20, 23, 7, 16, 5]. We show then how recursive operators and their calculational properties are derived from elementary categorical constructions.

An interesting feature of those constructions is that they are parameterized by the signature of the datatypes involved. Thus, as a natural consequence, the resulting operators are given by generic definitions on families of datatypes. Specific versions of the operators for particular datatypes may then be obtained by instantiation. This form of parametricity over type constructors has been the basis for recent developments in *generic* (or *polytypic*) *programming* [3, 17], and is precisely the approach we will follow throughout.

In this section (and in various other parts of the paper) the default category $\mathcal{C}$ will stand for **Cpo**, the category of *pointed cpos* —i.e. complete partial orders possessing a least element $\bot$— and continuous functions. The main reason for working in **Cpo** is that various of the recursive operators we are going to consider make only sense in a domain-theoretic setting, since they need to be defined as least fixed points of recursive equations. This is the case of hylomorphism [23], and the monadic versions of unfold and hylomorphism we will introduce.

As usual, a function $f : A \to B$ between pointed cpos is said to be *strict* if it preserves the least element, i.e. $f(\bot_A) = \bot_B$. The final object of **Cpo** is given by the singleton set $\{\bot\}$ and will be written as 1. By $\mathbf{Cpo}_\bot$ we denote the subcategory of **Cpo** formed by pointed cpos and strict continuous functions.

2.1 Functors

In this paper we will only consider a restricted class of datatypes, called *regular datatypes*. These are datatypes whose declarations contain no function spaces and have recursive occurrences with the same arguments from left-hand sides. Their signatures are given by so-called *regular functors*, which are presented next.

We start introducing what we consider basic functors: $I : \mathcal{C} \to \mathcal{C}$ stands for the identity functor; it is defined by the identity on objects and arrows. $\underline{A}^n : \mathcal{C}^n \to \mathcal{C}$ denotes the n-ary constant functor that maps n-tuples of objects to the object A, and n-tuples of functions to the

identity on A, id_A; when $n = 1$ we simply write $\underline{A}$. For $n \geq 2$, we write $\Pi_i^n : \mathcal{C}^n \to \mathcal{C}$ to denote the i-th projection functor from a n-ary product category.

The product bifunctor $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as cartesian product:

$$A \times B = \{(a, b) \mid a \in A,\ b \in B\}$$

Like in **Set** (the category of sets and total functions), cartesian product is a categorical product in **Cpo**. 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 $\langle f, g \rangle : C \to A \times B$. Product associativity is a natural isomorphism denoted by $\alpha_{A,B,C} : A \times (B \times C) \to (A \times B) \times C$.

The sum bifunctor $+ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as *separated sum*:

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

The sum inclusions $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ are defined by $\mathsf{inl}(a) = (0, a)$ and $\mathsf{inr}(b) = (1, b)$. The separated sum fails to be a coproduct in **Cpo** (see [21]); actually **Cpo** does not have coproducts. This means that, given two continuous functions $f : A \to C$ and $g : B \to C$, case analysis, defined as a strict function $[f, g] : A + B \to C$, is not the unique mediating morphism between $A + B$ and C that satisfies the equations $[f, g] \circ \mathsf{inl} = f$ and $[f, g] \circ \mathsf{inr} = g$. Its *fusion law* includes a strictness requirement:

$$f \text{ strict} \quad \Rightarrow \quad f \circ [g, h] = [f \circ g, f \circ h]$$

In a functional programming notation, a case analysis $[f, g]$ is often written as:

$$\lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ \mathsf{inl}(a) \to f(a);\ \mathsf{inr}(b) \to g(b)$$

Generalizations of product and sum to n components can be obtained in an obvious way.

The composition of a functor $F : \mathcal{C}^n \to \mathcal{C}$ with n functors $G_1, \ldots, G_n$ (all of the same arity) is written $F\langle G_1, \ldots, G_n \rangle$ (or $F\langle G_i \rangle$ for short); when $n = 1$ we omit the brackets. It stands for the functor that maps $A \mapsto F(G_1 A, \ldots, G_n A)$. We write $F \dagger G$ for $\dagger\langle F, G \rangle$ when $\dagger \in \{\times, +\}$.

Regular functors are functors built from identities, constants, projections, products, sums, type functors and composition. They can be inductively defined by the following grammar:

$$F ::= I \mid \underline{A}^n \mid \Pi_i^n \mid \times \mid + \mid D \mid F\langle F, \ldots, F \rangle$$

D stands for type functors, which correspond to fixed points of parameterized (regular) functors; they are introduced in Subsection 2.4. A functor F on **Cpo** is said to be *locally continuous* when its operation on functions is continuous. In particular, it can be verified that all regular functors are locally continuous.

2.2 Fold

Given an endofunctor $F : \mathcal{C} \to \mathcal{C}$, an *F-algebra* is a pair consisting of an object A (called the carrier) and a morphism $h : FA \to A$ (called the operation). An *algebra map*, or *homomorphism*, between two algebras $h : FA \to A$ and $k : FB \to B$ is an arrow $f : A \to B$ between the carriers which commutes with the operations: $f \circ h = k \circ Ff$. F-algebras and their homomorphisms form a category. We shall refer to it as $\mathbf{Alg}(F)$. Composition and identities in this category are inherited from $\mathcal{C}$.

The *canonical* solution to a domain equation $X \cong FX$ for a locally continuous endofunctor F on **Cpo** is specified by a pointed cpo μF together with an isomorphism $in_F : F\mu F \to \mu F$ (see [1], for example). The datatype μF contains partial, finite, as well as infinite values. The algebra in_F is a strict function that encodes the *constructors* of the datatype. We shall denote its inverse by $out_F : \mu F \to F\mu F$.

There exists a least homomorphism between in_F and any other algebra $h : FA \to A$ (see [1]). That homomorphism gives rise to a recursive operator, usually called *fold* [4] (or *catamorphism* [23]), which captures function definition by structural recursion. We shall denote it by $(\!|h|\!)_F : \mu F \to A$. Fold corresponds to the following least fixed point:

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

It is possible to verify that, if the algebra h is strict, then so is $(\!|h|\!)_F$ [7]. The following equation holds by definition of homomorphism.

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

It states that fold consumes the input data structure, replacing the constructors by the target algebra h.

There may exist multiple homomorphisms between in_F and another algebra h. This means that in_F is weak initial in $\mathbf{Alg}(F)$. That is the reason why to define fold as the least homomorphism between in_F and an algebra h. In every category for which an initial algebra in $\mathbf{Alg}(F)$ exists for every regular functor F, fold can be defined as the unique homomorphism between the initial algebra and any other algebra h [20, 7, 16]. Examples of categories where that holds are **Set** and $\mathbf{Cpo}_\perp$ [21, 7, 1].

Before illustrating specific instances of fold for some datatypes, we explain how parameterized datatypes are modeled. By fixing the first argument of a bifunctor $F : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ one can get a unary functor $F(A, -)$, to be written F_A, such that $F_A\, B = F(A, B)$ and $F_A\, f = F(\mathsf{id}_A, f)$. The functor F_A induces a parameterized datatype $DA = \mu F_A$ whose constructors $in_{F_A} : F_A(DA) \to DA$ are given by the canonical solution to the recursive equation $X \cong F(A, X)$.

Example 2.1 (i) The functor $L_A = \underline{1} + \underline{A} \times I$ captures the signature of lists with elements over A. Lists are usually declared in a functional programming language as follows:

$$\mathsf{list}(A) = \mathsf{nil} \mid \mathsf{cons}(A \times \mathsf{list}(A))$$

We will often write A^* for $\mathsf{list}(A)$. The datatype of lists then corresponds to the canonical solution of the equation $X \cong 1 + A \times X$. The constructors $\mathsf{nil} : 1 \to A^*$ and $\mathsf{cons} : A \times A^* \to A^*$ are given as part of the isomorphic algebra $[\mathsf{nil}, \mathsf{cons}] : 1 + A \times A^* \to A^*$. For any strict algebra $h = [h_1, h_2] : 1 + A \times B \to B$, fold is a strict function $f = (\!|h|\!)_{L_A} : A^* \to B$ such that,

$$f(\mathsf{nil}) = h_1 \qquad\qquad f(\mathsf{cons}(a, \ell)) = h_2(a, f(\ell))$$

It corresponds to the standard foldr operator used in functional programming [4].

(ii) Leaf-labelled binary trees,

$$\mathsf{tree}(A) = \mathsf{leaf}\ A \mid \mathsf{join}(\mathsf{tree}(A) \times \mathsf{tree}(A))$$

are obtained from the recursive equation $X \cong B_A X$, where $B_A = \underline{A} + I \times I$. The constructors are given within the algebra $[\mathsf{leaf}, \mathsf{join}] : A + \mathsf{tree}(A) \times \mathsf{tree}(A) \to \mathsf{tree}(A)$. The fold operator $f = (\!|h|\!)_{B_A}$, for $h = [h_1, h_2]$, is such that

$$f(\mathsf{leaf}(a)) = h_1(a) \qquad\qquad f(\mathsf{join}(t, u)) = h_2(f(t), f(u))$$

□

Folds enjoy many useful laws for program calculation. One of them is the *identity law*

$$(\!|in_F|\!)_F = \mathsf{id}_{\mu F} \tag{1}$$

which states that fold on the constructors is equal to the identity. Another possible reading for this law is that $\mathsf{id}_{\mu F}$ is the least endomorphism of in_F [19, 1].

A law that plays an important role in calculations is the so-called *fusion law* [20, 23, 7, 5, 4]:

$$f \text{ strict } \wedge \ f \circ h = h' \circ Ff \ \Rightarrow \ f \circ (\!|h|\!)_F = (\!|h'|\!)_F \tag{2}$$

It states that the composition of a strict homomorphism with a fold is again a fold.

2.3 Unfold

Given a functor F, a *F-coalgebra* is a pair consisting of an object A (the carrier) and an arrow $g : A \to FA$ (the operation). The functor F plays again the role of signature of the structure. A coalgebra map, or *homomorphism*, between two coalgebras $g : A \to FA$ and $g' : B \to FB$ is an arrow $f : A \to B$ such that $g' \circ f = Ff \circ g$. Just like algebras, coalgebras and their homomorphisms form a category $\mathbf{Coalg}(F)$.

The coalgebra $out_F : \mu F \to F\mu F$, inverse of in_F, is a strict function that encodes the destructors of the datatype. It turns out to be *final* in $\mathbf{Coalg}(F)$ [7]. This means that there exists a unique homomorphism from any coalgebra $g : A \to FA$ to out_F. The unique homomorphism gives rise to an operator, called *unfold* [13, 10] (or *anamorphism* [23]) and denoted by $[\!(g)\!]_F : A \to \mu F$, which satisfies the equation:

$$out_F \circ [\!(g)\!]_F = F\,[\!(g)\!]_F \circ g$$

Equivalently,

$$[\!(g)\!]_F = in_F \circ F\,[\!(g)\!]_F \circ g$$

The last equation states that unfold recursively builds up a data structure by decomposing its argument using coalgebra g.

Unfold can be defined in the same way in any other category for which a final coalgebra exists for every regular functor. For example, in **Set** and $\mathbf{Cpo}_\perp$ [20, 21, 7].

Example 2.2 (i) The functor $S_A = \underline{A} \times I$ captures the signature of the datatype of streams A^∞, formed by infinite (and partial) sequences of elements over A. Every stream coalgebra $g = \langle h, t\rangle : B \to A \times B$ is the pairing of two functions $h : B \to A$ and $t : B \to B$. The final coalgebra $\langle \mathsf{head}, \mathsf{tail}\rangle : A^\infty \to A \times A^\infty$ gives the destructors, while its inverse, $\mathsf{scons} : A \times A^\infty \to A^\infty$, the constructor of streams. The unfold operator is the unique function $f = [\!(g)\!]_{S_A} : B \to A^\infty$ such that $\mathsf{head} \circ f = h$ and $\mathsf{tail} \circ f = f \circ t$. Equivalently,

$$f = \mathsf{scons} \circ \langle h, f \circ t\rangle$$

(ii) Consider again $L_A = \underline{1} + \underline{A} \times I$. Given any coalgebra $g : B \to 1 + A \times B$, the unfold operator for lists is the unique function $f = [\!(g)\!]_{L_A} : B \to A^*$ such that

$$\begin{array}{lll} f(b) = & \textbf{case } g(b) \textbf{ of} & \\ & \quad \mathsf{inl}(\perp) & \to \ \mathsf{nil} \\ & \quad \mathsf{inr}(a, b') & \to \ \mathsf{cons}(a, f(b')) \end{array}$$

□

Calculational laws for unfold can be derived using finality of out_F. For instance, the *identity law* states that unfold on the destructors is equal to the identity.

$$[\!(out_F)\!]_F = \mathsf{id}_{\mu F} \tag{3}$$

There is also a corresponding *fusion law* [23, 7],

$$g \circ f = Ff \circ g' \ \Rightarrow \ [\!(g')\!]_F \circ f = [\!(g)\!]_F \tag{4}$$

which states that the composition of an unfold with a homomorphism yields an unfold.

2.4 Type Functors

Let $DA = \mu F_A$ be a parameterized datatype induced by a bifunctor F. D can be made into a functor $D : \mathcal{C} \to \mathcal{C}$, called a *type functor* [7, 5], by defining its action $Df : DA \to DB$ on an arrow $f : A \to B$, equally by a fold or an unfold.

$$Df = (\!|in_{F_B} \circ F(f, \mathsf{id}_{DB})|\!)_{F_A} = [\!(F(f, \mathsf{id}_{DA}) \circ out_{F_A})\!]_{F_B} \tag{5}$$

Example 2.3 For lists, the action on arrows $\mathsf{list}(f) = (\!|[\mathsf{nil}, \mathsf{cons} \circ (f \times \mathsf{id})]|\!)_{L_A}$ corresponds to the well-known map function [4]:

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

□

The following example shows that type functors may be used for defining other datatypes.

Example 2.4 Rose trees with elements over A are multiway branching trees declared by

$$\mathsf{rose}(A) = \mathsf{fork}\ (A \times \mathsf{list}(\mathsf{rose}(A)))$$

Their signature is captured by the functor $R_A = \underline{A} \times \mathsf{list}$. For an algebra $h : A \times B^* \to B$, fold is the function $f = (\!|h|\!)_{R_A} : \mathsf{rose}(A) \to B$ such that:

$$f(\mathsf{fork}(a, \ell)) = h(a, \mathsf{list}(f)(\ell))$$

while for $g = \langle g_1, g_2 \rangle : B \to A \times B^*$, unfold is the function $f = [\!(g)\!]_{R_A} : B \to \mathsf{rose}(A)$ such that,

$$f = \mathsf{fork} \circ \langle g_1, \mathsf{list}(f) \circ g_2 \rangle$$

□

The following are standard laws on type functors, called *map-fold-fusion* and *unfold-map-fusion*, respectively. For $f : A \to B$,

$$(\!|h|\!)_{F_B} \circ Df = (\!|h \circ F(f, \mathsf{id})|\!)_{F_A} \tag{6}$$

$$Df \circ [\!(g)\!]_{F_A} = [\!(F(f, \mathsf{id}) \circ g)\!]_{F_B} \tag{7}$$

2.5 Hylomorphism

Fold and unfold express standard ways of consuming and generating data structures respectively. Now we look at functions given by the composition of a fold with an unfold. They capture the idea of structuring a computation by an (implicit) intermediate data structure.

Given an algebra $h : FA \to A$ and a coalgebra $g : B \to FB$, the *hylomorphism* [23, 7] (or *hylo* for short) is the function $[\![h, g]\!]_F : B \to A$ defined by

$$[\![h, g]\!]_F \ = \ B \xrightarrow{\ [\!(g)\!]_F\ } \mu F \xrightarrow{\ (\!|h|\!)_F\ } A \tag{8}$$

Note that this definition makes only sense in a setting where the data structures produced by unfolds are the same as those consumed by folds, as it is the case of **Cpo**.

Applying the identity laws (1) and (3), it is immediate to see that folds and unfolds are themselves hylos:

$$(\!|h|\!)_F = [\![h, out_F]\!]_F \qquad\qquad [\!(g)\!]_F = [\![in_F, g]\!]_F$$

Equivalently, hylos can be given by a fixed point definition:

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

This alternative definition shows that we can always transform the composition of a fold with an unfold into a single function that avoids the construction of the intermediate data structure. Expanding this definition, we obtain the equation

$$[\![h, g]\!]_F = h \circ F\,[\![h, g]\!]_F \circ g$$

which expresses the shape of recursion that comes with each datatype.

Two well-known examples of hylos are quicksort and merge sort. In both sorting methods there is an implicit data structure, a different kind of binary tree in each one, that determines the shape of the call-trees (see [2]).

Example 2.5 Consider again the functor $L_A = \underline{1} + \underline{A} \times I$. For $g : B \to 1 + A \times B$ and $h = [h_1, h_2] : 1 + A \times C \to C$, the hylomorphism for lists is given by,

$$[\![h, g]\!]_{L_A} = \mathsf{fix}(\lambda f.[h_1, h_2 \circ (\mathsf{id}_A \times f)] \circ g)$$

It can can be functionally expressed as follows:

$$\begin{array}{lll} [\![h, g]\!](b) = & \textbf{case } g(b) \textbf{ of} & \\ & \quad \mathsf{inl}(\bot) & \to\ h_1 \\ & \quad \mathsf{inr}(a, b') & \to\ h_2(a, [\![h, g]\!](b')) \end{array}$$

□

The expressiveness of hylos is very rich. In practice, almost every interesting recursive function, even a fixed-point operator (see [24]), can be expressed as a hylo.

The following *fusion laws* [7],

$$f \text{ strict } \wedge\ f \circ h = h' \circ Ff \quad \Rightarrow \quad f \circ [\![h, g]\!]_F = [\![h', g]\!]_F \tag{9}$$

$$g \circ f = Ff \circ g' \quad \Rightarrow \quad [\![h, g]\!]_F \circ f = [\![h, g']\!]_F \tag{10}$$

are a direct consequence of (8) and the fusion laws (2) and (4) for fold and unfold, respectively.

Hylos have been demonstrated useful and effective for the application of deforestation techniques (see [35, 31]). This has been motivated by the existence of two laws, usually referred to as the *Acid Rain Theorem* [35, 31], which, under certain conditions, permit to fuse the composition of hylos with folds and unfolds, yielding new hylos. We present this theorem below, but first we need to introduce the notion of transformer.

A function $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ that converts each F-algebra into a G-algebra on the same carrier is said to be a *transformer* [7] 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, $\boldsymbol{T}$ is a transformer if every homomorphism between two F-algebras happens to be also a homomorphism between the respective G-algebras.

Obviously, a similar notion can be defined for coalgebras as well. A function $\boldsymbol{T} : (A \to FA) \to (A \to GA)$ is said to be a *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 2.6 (Acid Rain)

hylo-fold fusion: *Let* $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ *be a transformer.*

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

unfold-hylo fusion: *Let* $\boldsymbol{T} : (A \to FA) \to (A \to GA)$ *be a transformer.*

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

Proof. We only show the proof of hylo-fold fusion; the other case is analogous. Suppose $h : FA \to A$ strict. Recall that every fold is a homomorphism of algebras. Thus, by definition of 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, observe that $(\!|h|\!)_F$ is strict, since by hypothesis h is strict. Therefore, by (9) we arrive at the desired result. □

Since folds and unfolds are particular cases of hylos, one can write a specialization of this theorem in terms of folds and unfolds solely.

$$h \text{ strict} \;\wedge\; \boldsymbol{T} \text{ transformer} \quad \Rightarrow \quad (\!|h|\!)_F \circ (\!|\boldsymbol{T}(in_F)|\!)_G = (\!|\boldsymbol{T}(h)|\!)_G$$

$$\boldsymbol{T} \text{ transformer} \quad \Rightarrow \quad [\!(\boldsymbol{T}(out_F))\!]_G \circ [\!(g)\!]_F = [\!(\boldsymbol{T}(g))\!]_G$$

On the other hand, one can also obtain a generalization of this theorem, by substituting the fold and the unfold by respective hylos. Indeed, that is the version of the theorem commonly found in the literature (see e.g. [31] and references therein).

3 Monads

In this work we are interested in studying recursive programs with effects. Monads permit to uniformly express a wide variety of computational effects, such as exception handling, state-based computations, parsing or nondeterminism, in a functional setting. Moggi [27] proposed monads as a mechanism for structuring denotational semantics descriptions of programming languages. Afterwards, Wadler [38, 39] introduced monads into functional programming, by showing that the same technique can be applied for structuring functional programs that produce effects. In the last years, the structuring method based on monads has become a style of programming widely adopted by the Haskell community [6].

This section reviews as much of monads as we need. The concept of a monad originates from category theory and can be defined in various ways. We start with the formulation in terms of so-called Kleisli triples, which appears as the most suitable for programming purposes.

Definition 3.1 ([27]) A **Kleisli triple** $(M, \mathsf{unit}, -^\star)$ over a category $\mathcal{C}$ 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: $\mathsf{unit}_A^\star = \mathsf{id}_{MA}$, $f^\star \circ \mathsf{unit}_A = f$, and $f^\star \circ g^\star = (f^\star \circ g)^\star$. □

The use of monads introduces an explicit distinction between the notions of *value* and *computation*. Computations delivering values of type A are regarded as objects of type MA. Under this interpretation, unit_A can be understood as an operation that turns a value into a computation that yields this value without producing any effect at all. The extension operator gives a way of composing monadic functions, passing the effect around: The Kleisli (or monadic) composition of two functions $f : A \to MB$ and $g : B \to MC$ is defined as follows:

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

Now we can assign a meaning to the Kleisli triple laws. The first two laws amount to say that unit is a left and right identity with respect to Kleisli composition, whereas the last one expresses that composition is associative. In other words, the Kleisli triple laws just express that monadic morphisms form a category.

Definition 3.2 ([27]) 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. □

In the context of functional programming, a monad is often presented as 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 typically called *bind*. The Kleisli triple laws translate to corresponding equations in terms of unit and $\star$ (see [39], for example). An expression $m \star f$ corresponds to $f^\star(m)$. We will keep both notations for *bind*, using each one where it better suits. The infix notation turns out to be preferable for writing functional programs in monadic style, as it gives a graphical idea of the sequencing that exists between computations. In this sense, expressions of the form $m \star \lambda v.\, m'$ are usual, and 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.

Example 3.3 (i) Of all monads, the *identity monad* is the simplest one: $M = I$; $\mathsf{unit}_A = \mathsf{id}_A$; 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 specific exception signaled by a value of type E. The corresponding unit and extension operator are given by:

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

where $f : A \to B + E$. That is, unit takes a value and returns a computation that always succeeds. The extension operator may be thought of as a kind of *strict* function application that propagates the exception if it is the case. In the particular case of having a unique exception value, i.e. when E is given by the unit type 1, the exception monad is sometimes referred to as the *maybe monad* [38, 26].

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

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

where $f : A \to [S \to B \times S]$. That is, unit takes a value and returns a computation that yields this value without modifying the state. The bind operator sequences two computations so that the state and value resulting from the first are supplied to the second.

Various proposals have shown how the state monad can also be used as an effective tool for encapsulating actual imperative features, such as mutable variables, destructive data structures, or I/O, while retaining fundamental properties like referential transparency (see [33, 18]). They achieve this goal by hiding the *real* state in an abstract data type based on the monad that is equipped with operations that internally access the real state. Such technique can be applied either when the state is internal or external to the program. I/O is a typical case of the latter.□

Given a monad M over $\mathcal{C}$, we can define a *lifting functor* $\widehat{(-)} : \mathcal{C} \to \mathcal{C}_M$ as the identity on objects, and $\widehat{f} = \mathsf{unit}_B \circ f : A \to MB$, 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 form an adjunction $(\widehat{-}) \dashv U : \mathcal{C}_M \to \mathcal{C}$, with $\eta = \mathsf{unit}$ and $\epsilon = \mathsf{id}$.

An alternative definition of monad is the following.

Definition 3.4 ([27]) A **monad** over $\mathcal{C}$ is a triple (M, unit, μ) formed 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*) which obey 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 monad are equivalent. In fact, from the Kleisli triple components one can define the action of M on an arbitrary arrow $f : A \to B$ by $Mf = (\mathsf{unit}_B \circ f)^\star$, and the multiplication by $\mu_A = id^\star_{MA}$. 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 property, specific to monads over **Cpo**, will be used later.

Lemma 3.5 *The multiplication μ of every monad over* **Cpo** *is a strict operation.*

It seems to us that this property should be known. However, we haven't seen it in the literature, so we prove it here anyway.

Proof. By Definition 3.4, $\mu_A \circ \mathsf{unit}_{MA} = \mathsf{id}_{MA}$ for each object A. The following property holds in **Cpo**: if $A \to B \to C$ is a strict function, then so is $B \to C$ (see [9] for a proof). Therefore, μ_A is strict as so is the identity. □

Example 3.6 The *list monad* permits to describe computations that yield a list of results:

$$MA = A^* \qquad \mathsf{unit}_A(a) = \mathsf{cons}(a, \mathsf{nil}) \qquad \mu_A = \mathsf{concat}$$

where $\mathsf{concat} = (\!|[\mathsf{nil}, +\!\!+]|\!) : (A^*)^* \to A^*$ concatenates a list of lists into a list and $+\!\!+ : A^* \times A^* \to A^*$ concatenates two lists to form a longer list. (These are standard functions in functional programming [4].) Computations of this kind may be thought of as offering a choice of values, which can be used to model a form of nondeterminism. Since $f^\star = \mu_B \circ Mf = \mathsf{concat} \circ \mathsf{list}(f)$ and concat is a fold, by applying map-fold-fusion, law (6), we get that: $f^\star = (\!|[\mathsf{nil}, +\!\!+ \circ (f \times \mathsf{id})]|\!)_{L_A}$. That is, $f^\star(\mathsf{nil}) = \mathsf{nil}$ and $f^\star(\mathsf{cons}(a, \ell)) = f(a) +\!\!+ f^\star(\ell)$. □

In subsequent sections we will assume that all monads we deal with are *strong*. The notion of strong monad is derived from that of strong functor. An endofunctor $M : \mathcal{C} \to \mathcal{C}$ on a category $\mathcal{C}$ with product is called *strong* if it comes equipped with a natural transformation $\tau_{A,B} : A \times MB \to M(A \times B)$, called a *strength*, satisfying these equations:

$$M\,\pi_2 \circ \tau_{1,A} = \pi_2 \qquad \tau_{A,B\times C} \circ (\mathsf{id}_A \times \tau_{B,C}) \circ \alpha_{A,B,MC} = M\alpha_{A,B,C} \circ \tau_{A\times B,C}$$

A monad (M, unit, μ) is called *strong* [27] if M is a strong functor, and in addition, $\mathsf{unit} : I \Rightarrow M$ and $\mu : M^2 \Rightarrow M$ are strong natural transformations,[1] which is expressed by the equations:

$$\mathsf{unit}_{A\times B} = \tau_{A,B} \circ (\mathsf{id}_A \times \mathsf{unit}_B) \tag{11}$$

$$\mu_{A\times B} \circ M\tau_{A,B} \circ \tau_{A,MB} = \tau_{A,B} \circ (\mathsf{id}_A \times \mu_B) \tag{12}$$

For example, in a cartesian closed category (like **Set** or **Cpo**) there is a one-to-one correspondence between strengths and enrichments of functors. An endofunctor F is $\mathcal{C}$-enriched (over the

[1] A natural transformation $\sigma : F \Rightarrow G$ between strong functors F and G is called *strong* if it is compatible with the strengths. That is, $\sigma_{A\times B} \circ \tau^F_{A,B} = \tau^G_{A,B} \circ (\mathsf{id}_A \times \sigma_B)$.

exponentiation) when there is a natural transformation $\mathsf{fmap}_{A,B} : [A \to B] \to [FA \to FB]$ that internalizes the action of F on arrows from A to B. Thus, in this setting, a strong monad is equivalent to a monad with an enriched functor. The equations (11) and (12) then say that $\mathsf{unit} : I \Rightarrow M$ and $\mu : M^2 \Rightarrow M$ are $\mathcal{C}$-enriched natural transformations.

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 transformation $\psi_{A,B} : MA \times MB \to M(A \times B)$, given by $\psi_{A,B} = \tau_{A,B} \bullet \tau'_{A,MB}$, which describes how the monad distributes over the product [27]. It says that a pair of computations may be joined as a new computation by first evaluating the first argument and then the second. Functionally expressed,

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

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. $\psi_{A,B} = \psi'_{A,B}$. Monads like identity or state reader [39] (also called environment monad) are commutative. The maybe monad is commutative in **Set**, for example, but not in **Cpo**. Indeed, suppose that we want to join two computations m and m' on this monad, such that m diverges and m' fails. Then, the computation that results from $\psi(m, m')$ will diverge, whereas the one given by $\psi'(m, m')$ will fail. Examples of non-commutative monads are exception (with multiple error values), state and list.

4 Monadic Extensions of Functors

In the previous section we saw that morphisms in a category $\mathcal{C}$ can be mapped to the Kleisli category by the action of the lifting functor $\widehat{(-)} : \mathcal{C} \to \mathcal{C}_M$. In this section, we focus our attention on functors on $\mathcal{C}$. For each functor F, we describe how to derive a construction, denoted by $\widehat{F}$, which acts on elements of the Kleisli category. We call such a construction a *monadic extension of* F. In the particular case that F is a regular functor, we see that a monadic extension $\widehat{F}$ can be defined by induction on the structure of F. (Analogous definitions to the given here can be found in [8, 12, 36]). We present also the main properties that monadic extensions satisfy (see [32] for further details).

The usefulness of monadic extensions will become clear in Section 5, when we address the definition of recursive operators that represent functions with effects. There, we will see that some properties of monadic extensions play a relevant role in the calculational theory of these operators.

A *monadic extension* of a functor $F : \mathcal{C} \to \mathcal{C}$ is a construction $\widehat{F} : \mathcal{C}_M \to \mathcal{C}_M$ 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 $f : A \to MB$, it yields an arrow $\widehat{F}f : \widehat{F}A \to M(\widehat{F}B)$ —actually $\widehat{F}f : FA \to M(FB)$— in $\mathcal{C}_M$, whose action embodies that of F. Observe that every monadic extension satisfies the equality $\widehat{F}(\widehat{-}) = (\widehat{-})F$ on objects.

Monadic extensions $\widehat{F} : \mathcal{C}_M \to \mathcal{C}_M$ are in one-to-one correspondence with natural transformations $\delta^F : FM \Rightarrow MF$ —often called *distribution laws*— that perform the distribution of a monad over a functor [29, 32]. In effect, for each distribution law δ^F, we can define the action of a monadic extension $\widehat{F}$ on arrows $f : A \to MB$ as follows:

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

Conversely, a distribution law δ^F can be derived from a monadic extension $\widehat{F}$ by using the adjunction between $\mathcal{C}_M$ and $\mathcal{C}$ mentioned in Section 3; namely, $\delta^F_A = \widehat{F}\mathsf{id}_{MA}$.

It is easy to verify that the following equalities hold for every monadic extension:

$$M\!Fg \circ \widehat{F}f = \widehat{F}(Mg \circ f) \tag{13}$$

$$\widehat{F}g \circ Ff = \widehat{F}(g \circ f) \tag{14}$$

A monadic extension $\widehat{F}$ is called a *lifting* of F when it happens to be a functor on $\mathcal{C}_M$.

Theorem 4.1 ([29]) *Given a monad (M, unit, μ) and a functor F on $\mathcal{C}$, $\widehat{F} : \mathcal{C}_M \to \mathcal{C}_M$ is a lifting of F iff the natural transformation $\delta^F : FM \Rightarrow MF$ satisfies these equations:*

$$\delta^F_A \circ F\mathsf{unit}_A = \mathsf{unit}_{FA} \tag{15}$$

$$\mu_{FA} \circ M\delta^F_A \circ \delta^F_{MA} = \delta^F_A \circ F\mu_A \tag{16}$$

These equations ensure that the functoriality axioms for $\widehat{F}$ hold. Equation (15) considers the preservation of identities, $\widehat{F}\mathsf{unit}_A = \mathsf{unit}_{FA}$, while (16) makes lifting distribute over composition in the Kleisli category, $\widehat{F}(f \bullet g) = \widehat{F}f \bullet \widehat{F}g$.

Now let us analyse the behaviour of some specific functors. We start with functors involving product.

Proposition 4.2 ([29]) *Let category $\mathcal{C}$ be cartesian. Let M be a strong monad over $\mathcal{C}$ and A an object of $\mathcal{C}$. Then, the functor $F = A \times -$ 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 of Theorem 4.1 can be generalized for multiary functors $F : \mathcal{C}^n \to \mathcal{C}$ in a straightforward manner (see [36, 32]). The following proposition uses that generalization.

Proposition 4.3 *Let category $\mathcal{C}$ be cartesian. Let M be a commutative monad. Then, the product functor $\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ has a lifting with $\delta^{\times}_{(A,B)} = \psi_{A,B}$. (Another lifting of the product can be obtained by using the right-to-left product distribution ψ'.)*

A proof of this proposition can be found in [36, 32]. When the monad is strong but non-commutative —like e.g. the state monad— the product functor makes (15) still hold, while (16) fails:

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

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

In other words, the monadic extension $\widehat{\times}$ on a strong non-commutative monad lacks the preservation of Kleisli composition.

Now, let us see what happens with the sum functor.

Proposition 4.4 ([36]) *Let $\mathcal{C}$ be a category with coproduct $+$. Then, the sum functor $+ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ has a lifting with $\delta^{+}_{(A,B)} = [M\mathsf{inl}, M\mathsf{inr}]$.*

Note 4.5 An interesting case obviously left out by Proposition 4.4 is that of the sum functor in **Cpo**. The reason is that this is a category without coproduct. Even though, something can be said about the sum functor we adopted for **Cpo**, namely the separated sum. To define the monadic extension of the separated sum, $\widehat{+}$, one uses the same distribution law as in Proposition 4.4, i.e. $\delta^{+}_{(A,B)} = [M\mathsf{inl}, M\mathsf{inr}]$. However, unlike in Proposition 4.4, the monadic extension so defined might not be a lifting, since it may lack the preservation of identities. In fact, it may happen that

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

The problem here is that this inequation becomes an equality only when unit is a strict function. Unfortunately, not every monad satisfies this property. For example, monads like exception or state have non-strict unit.

On the other hand, one can see that $\widehat{+}$ preserves Kleisli compositions provided that M is a strictness-preserving functor.[2] In other words, the following equation holds for the separated sum:

$$\mu_{A+B} \circ M\delta^{+}_{(A,B)} \circ \delta^{+}_{(MA,MB)} = \delta^{+}_{(A,B)} \circ (\mu_A + \mu_B)$$

The proof of this equation uses the fact that the multiplication, μ, of every monad over **Cpo** is a strict function (see Lemma 3.5). A construction like $\widehat{+}$ that preserves compositions but not necessarily identities is called a *semi-functor* [11]. □

Now we consider regular functors in general. First, we analyze the distribution laws and monadic extensions corresponding to regular functors under the assumption that the underlying category has product, coproduct and initial algebra for every regular functor. Regular functors on **Cpo**, on the other hand, are discussed at the end of the section.

Definition 4.6 Let M be a strong monad over $\mathcal{C}$. For each regular functor F we can define a distribution law $\delta^F : FM \Rightarrow MF$ by induction on the structure of F as follows.

$$\begin{array}{lll}
\delta^{I}_{A} & = \mathsf{id}_{MA} & \delta^{+}_{(A,B)} = [M\mathsf{inl}, M\mathsf{inr}] \\
\delta^{\underline{C}}_{A} & = \mathsf{unit}_C & \delta^{F\langle G_i\rangle}_{A} = \delta^{F}_{(G_iA)} \circ F(\delta^{G_1}_{A}, \ldots, \delta^{G_n}_{A}) \\
\delta^{\Pi^n_i}_{(A_1,\ldots,A_n)} & = \mathsf{id}_{MA_i} & \delta^{D}_{A} = (\!|Min_{F_A} \circ \delta^{F}_{(A,DA)}|\!)_{FMA} \\
\delta^{\times}_{(A,B)} & = \psi_{A,B} &
\end{array}$$

□

Analogous definitions can be found in [8, 12, 36]. We used (G_iA) as an abbreviation for $(G_1A, \ldots, G_nA)$. That $\delta^{\times}$ is defined as ψ means that monadic effects in products are sequenced from left-to-right; a right-to-left policy can also be specified by using ψ' instead of ψ. The definition of δ^D is a form of monadic fold (F is the bifunctor that induces D); δ^D_A is also known as *passive traversal* [28] and function thread [3].

From Definition 4.6, these typical cases can be calculated:

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

Example 4.7

1. For $L_A = \underline{1} + \underline{A} \times I$, $\delta^{L_A}_{X} = [\widehat{\mathsf{inl}}, \widehat{\mathsf{inr}} \bullet \tau_{A,X}] : 1 + A \times MX \to M(1 + A \times X)$.
2. For $B_A = \underline{A} + I \times I$, $\delta^{B_A}_{X} = [\widehat{\mathsf{inl}}, \widehat{\mathsf{inr}} \bullet \psi_{X,X}] : 1 + MX \times MX \to M(1 + X \times X)$.
3. For $R_A = \underline{A} \times \mathsf{list}$,

$$\delta^{R_A}_{X} = \psi_{A,X^*} \circ (\mathsf{unit} \times \delta^{\mathsf{list}}) = \tau_{A,X^*} \circ (\mathsf{id}_A \times \delta^{\mathsf{list}}) : A \times (MX)^* \to M(A \times X^*)$$

where $\delta^{\mathsf{list}} = (\!|[\widehat{\mathsf{nil}}, \widehat{\mathsf{cons}} \bullet \psi_{X,X^*}]|\!) : (MX)^* \to MX^*$. That is,

[2] A functor M is called *strictness-preserving* if its action on arrows maps a strict function f to a strict function Mf. For example, as shown by Freyd [9], every locally continuous functor on **Cpo** is strictness-preserving.

$$\begin{aligned}\delta^{\mathsf{list}}(\mathsf{nil}) &= \mathsf{unit}(\mathsf{nil})\\ \delta^{\mathsf{list}}(\mathsf{cons}(m,\ell)) &= m \star \lambda x.\, \delta^{\mathsf{list}}(\ell) \star \lambda xs.\, \mathsf{unit}(\mathsf{cons}(x,xs))\end{aligned}$$

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

□

From Definition 4.6 we can derive a monadic extension for every regular functor:

$$\begin{array}{ll} \widehat{I}f = f & f \mathbin{\widehat{+}} g = [M\mathsf{inl} \circ f, M\mathsf{inr} \circ g] \\ \widehat{\underline{C}}f = \mathsf{unit}_C & \widehat{F\langle G_i\rangle}f = \widehat{F}(\widehat{G}_1 f, \ldots, \widehat{G}_n f) \\ \widehat{\Pi}_i^n(f_1,\ldots,f_n) = f_i & \widehat{D}f = (\!|M in_{F_B} \circ \widehat{F}(f, \mathsf{id})|\!)_{F_A} \\ f \mathbin{\widehat{\times}} g = \psi \circ (f \times g) & \end{array}$$

In particular,

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

The extension $\widehat{D}$ is usually called a *monadic map* [8], it is also known as *active traversal* [28]. Its expression is obtained by applying map-fold-fusion, law (6), to the composition $\delta^D \circ Df$.

Example 4.8 The distributions laws of Example 4.7 give rise to these extensions:

1. $\widehat{L_A}f = [\widehat{\mathsf{inl}}, \widehat{\mathsf{inr}} \bullet \tau \circ (\mathsf{id} \times f)]$.
2. $\widehat{B_A}f = [\widehat{\mathsf{inl}}, \widehat{\mathsf{inr}} \bullet \psi \circ (f \times f)]$. That is,

$$\begin{array}{rll} (\widehat{B_A}f)(x) = & \textbf{case } x \textbf{ of} & \\ & \mathsf{inl}(a) & \to\ \mathsf{unit}(\mathsf{inl}(a)) \\ & \mathsf{inr}(y,y') & \to\ f(y) \star \lambda z.\, f(y') \star \lambda z'.\, \mathsf{unit}(\mathsf{inr}(z,z')) \end{array}$$

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

$$\begin{aligned}\widehat{\mathsf{list}}(f)(\mathsf{nil}) &= \mathsf{unit}(\mathsf{nil})\\ \widehat{\mathsf{list}}(f)(\mathsf{cons}(a,\ell)) &= f(a) \star \lambda b.\, \widehat{\mathsf{list}}(f)(\ell) \star \lambda \ell'.\, \mathsf{unit}(\mathsf{cons}(b,\ell'))\end{aligned}$$

□

Theorem 4.9 *Let $\mathcal{C}$ be a category with product, coproduct, and such that there is an initial algebra for every regular functor. If M is a commutative monad, then every regular functor extends to a lifting.*

The proof of this theorem is by induction on the structure of regular functors. It consists of the verification of equations (15) and (16) for the distribution laws in Definition 4.6 (see [36, 32] for details of the proof). The cases of the product and sum functors are given by Propositions 4.3 and 4.4.

Since commutativity of the monad is only used for the product functor, slightly weaker results can be stated after eliminating this assumption.

Proposition 4.10 *Let category $\mathcal{C}$ be as above and let M be a strong monad. Then,*

1. *The extension of every regular functor preserves identities.*

2. *All regular functors containing only product expressions of the form $A \times -$ or $- \times A$, for some object A, extend to a lifting.*

Proof. Case (1) follows from the fact that commutativity is never used in the verification of equation (15). For (2), recall that the functors $A \times -$ and $- \times A$ have a lifting for any strong monad (see Proposition 4.2). The proof for the remainder regular functors proceeds like in Theorem 4.9. □

We conclude this section with a short analysis of monadic extensions for regular functors on **Cpo**. (See [32] for further details.) Observe that the distribution laws given in Definition 4.6 can be reused for regular functors on **Cpo**, even though the assumptions that the underlying category possesses coproduct and initial algebra for each regular functor do not hold in this setting. In fact, these assumptions did not have effect on the definition of the distribution laws. However, they were essential for asserting the properties we deduced for the corresponding monadic extensions. Consequently, in order to achieve similar results on **Cpo** we need to substitute the invalid assumptions by others.

Before asserting general results we analyze the case of some specific functors. For example, the results stated in the Propositions 4.2 and 4.3 still apply in this setting, since cartesian product is a categorical product in **Cpo**. The case of the sum functor in **Cpo** was already discussed in Note 4.5. There, we saw that the monadic extension of the separated sum preserves identities provided that the monad possesses a strict unit, and preserves Kleisli compositions whenever the functor M of the monad is strictness-preserving.

It is possible to verify that essentially the same holds for the extensions of type functors. We give an idea of this fact by means of a concrete example. Consider the monadic extension of the list type functor, $\widehat{\mathsf{list}}(f)$, described in Example 4.8. It is a strict fold because its target algebra is strict. Thus, in particular, $\widehat{\mathsf{list}}(\mathsf{unit})$ is strict. However, unit might not be strict. When that is the case it follows that $\widehat{\mathsf{list}}(\mathsf{unit}) \neq \mathsf{unit}$. On the contrary, when unit is strict, it is not hard to verify that $\widehat{\mathsf{list}}(\mathsf{unit})$ and unit are the same function.

Summing up, we can state the following.

Proposition 4.11 ([32]) *Let M be a strong monad over* **Cpo** *with strict* unit *and strictness preserving functor. Then, the monadic extension of every regular functor preserves identities.*

The proof is by induction on the structure of regular functors. It uses the fact that, under the same hypothesis, the monadic extension of every regular functor is a strictness-preserving operation (see [32] for details).

Concerning the preservation of Kleisli composition, the following can be stated.

Proposition 4.12 *Let M be a strong monad over* **Cpo** *with strictness-preserving functor. Then,*

1. *If M is commutative then the monadic extension of every regular functor preserves Kleisli compositions.*

2. *The monadic extension of every regular functor containing only product expressions of the form $A \times -$ or $- \times A$ preserves Kleisli compositions.*

Example 4.13 Let M be a strong monad over **Cpo** with strictness-preserving functor and non-strict unit. Consider the extensions in Example 4.8.

1. $\widehat{L_A}$ is a semi-functor, as L_A is given by a sum.

2. $\widehat{B_A}$ does not preserve identities because B_A is given by a sum. To preserve compositions, M needs to be commutative because B_A contains a product expression.

3. Note that R_A can be expressed as the functor composition $(A \times -)\,\mathsf{list}$. Therefore, $\widehat{R_A}$ is a semi-functor, as so is $\widehat{\mathsf{list}}$.

□

5 Merging Recursion and Monads

Sections 2 and 3 presented two different trends in program design. One of them is data-driven, in the sense that programs are constructed according to the structure of the values they generate or consume. The other one is effects-driven, and is based on the use of monads as structuring device. In this section we show that these two structuring mechanisms can be combined, giving rise to monadic extensions of the recursive operators presented in Sections 2.

Research on monadic recursive operators was initiated by Fokkinga [8] with the study of monadic maps and folds. From a more pragmatical perspective, Meijer and Jeuring [26] presented various examples of these operators and showed concrete applications of their laws. In Subsection 5.1 we shortly review monadic folds.

Subsections 5.2 and 5.3 contain our main contributions to the subject in question. There, we introduce two new operators that correspond to monadic versions of unfold and hylomorphism respectively. We also study the relevant aspects of their calculational theory. The fusion laws for these operators turn out to be of fundamental importance, since they enclose new practical cases of deforestation. Various of the functions that take part in those laws will be required to possess a structure that is only captured by the new operators or by any of their associated concepts.

Throughout this section the way in which monads encapsulate effects demonstrate extremely helpful for analyzing recursive functions with effects. Monads permit to focus on the relevant structure of functions disregarding details about the specific effect that a function produces.

In this section we will assume again that the default category $\mathcal{C}$ is **Cpo**.

5.1 Monadic Fold

Various of the concepts we saw in Section 2 can be lifted to the Kleisli category. One of them is the concept of algebra. A *monadic F-algebra* is a pair consisting of an object A (the carrier) and an arrow $h : FA \to MA$ (the operation). A homomorphism between monadic algebras $h : FA \to MA$ and $h' : FB \to MB$ is an arrow $f : A \to MB$ such that $f \bullet h = h' \bullet \widehat{F} f$.

By definition, every homomorphism $f : \mu F \to MA$ between $\widehat{in_F}$ —the lifting of the datatype constructors— and any other monadic algebra $h : FA \to MA$ satisfies the following equation:

$$f \bullet \widehat{in_F} = h \bullet \widehat{F} f \tag{17}$$

The *monadic fold* operator [8] is then defined as the least homomorphism between $\widehat{in_F}$ and h. We shall denote it by $(\!|h|\!)^M_F : \mu F \to MA$. The existence of a least homomorphism between $\widehat{in_F}$ and any h follows from the fact that (17) reduces to an equation of a homomorphism between two normal algebras. Indeed, since $\widehat{F} f = \delta^F_A \circ Ff$ and $f \bullet \widehat{in_F} = f \circ in_F$, equation (17) rewrites to:

$$f \circ in_F = (h \bullet \delta^F_A) \circ Ff$$

Thus, every homomorphism $f : \mu F \to MA$ between the monadic algebras $\widehat{in_F}$ and h is a homomorphism between the (normal) algebras in_F and $h \bullet \delta^F_A : F(MA) \to MA$, and vice-versa. Since a least homomorphism between in_F and any other algebra always exists, the desired

property follows. In other words, every monadic fold is a special case of a fold.

$$(\!|h|\!)^M_F = (\!|h \bullet \delta^F_A|\!)_F \tag{18}$$

In a category, such as **Set** or $\mathbf{Cpo}_\perp$, for which an initial algebra exists for every regular functor F, monadic fold is defined as the unique homomorphism between the lifting of the initial algebra and any other monadic algebra h. That is the way monadic fold is usually presented in the literature [8, 12]. The uniqueness of that homomorphism in this setting is also a consequence of the fact we just proved about equation (17).

Example 5.1 Consider the datatype of lists. Given a monadic algebra $h = [h_1, h_2] : 1+A\times B \to MB$, the monadic fold is a function $f = (\!|h|\!)^M_{L_A} : A^* \to MB$ such that

$$f(\mathsf{nil}) = h_1 \qquad\qquad f(\mathsf{cons}(a,\ell)) = f(\ell) \star \lambda x.\, h_2(a,x)$$

□

A monadic fold $(\!|h|\!)^M_F$ is strict if $h : FA \to MA$ and δ^F_A are strict functions and the functor M of the monad is strictness-preserving. The proof of this property follows directly from (18) and the fact that a fold is strict when so is its target algebra. The verification that the algebra $h \bullet \delta^F_A$ is strict is straightforward.

A similar property states that a monadic fold $(\!|\widehat{h}|\!)^M_F$, with $h : FA \to A$, is strict if h and δ^F are strict and the functor M of the monad is strictness-preserving. Observe that this property reduces to the previous one only when the unit of the monad is strict.

An *identity law* for monadic fold does not hold in general. This means that it is not always true that

$$(\!|\widehat{in_F}|\!)^M_F = \mathsf{unit}$$

For example, this equation is not satisfied when δ^F is strict, M is strictness-preserving, but the unit of the monad is non-strict: while $(\!|\widehat{in_F}|\!)^M_F$ is strict by the previous property (because so is in_F), unit is non-strict by hypothesis. On the other hand, it is possible to prove that the identity law holds for every regular functor F when M is strictness-preserving and the unit of the monad is strict. We do not need to require that δ^F be strict for every regular functor, since that follows from strictness of unit (see [32] for details).

Further laws for monadic fold can be found in [8, 12, 32].

5.2 Monadic Unfold

A definition of monadic unfold can be obtained by dualizing the recursion scheme that characterizes monadic fold. However, we prefer not to proceed in this way, but to introduce monadic unfold by means of an intuitive explanation of its behaviour. The development that follows can be conducted in any category for which a final coalgebra exists for every regular functor. Let M be an arbitrary monad.

Roughly speaking, a *monadic unfold* is a function that behaves like an unfold, but with the additional feature of producing an effect. A first approximation to its definition can be got by lifting to the Kleisli category $\mathcal{C}_M$ the recursion pattern that characterizes unfold. This means to consider the diagram:

$$\begin{array}{ccc} A & \xrightarrow{\;f\;} & D \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle out_G} \\ GA & \xrightarrow[Gf]{} & GD \end{array}$$

and view it as a diagram in the Kleisli category. Proceeding that way, we are actually thinking of each arrow as an effect-producing function, getting the somewhat 'imperative' idea of a recursive process that produces some (implicit) side-effect while it generates a data structure.

Being $\mathcal{C}$ the 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 what such a scheme means. The first step towards the description in $\mathcal{C}$ is to make the computational effect explicit.

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & MD \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle out_G^\star} \\ M(GA) & \xrightarrow[(Gf)^\star]{} & M(GD) \end{array}$$

Now, it rests to determine what plays the role of G and out_G. Since the objects of $\mathcal{C}$ and $\mathcal{C}_M$ coincide, the data structure D generated by such a recursive definition necessarily corresponds to a datatype of $\mathcal{C}$ with signature given by a functor F. This means that G is a *monadic extension* $\widehat{F}$ of F, as it acts on elements of $\mathcal{C}_M$. Hence, $Gf = \widehat{F}f$ and $GX = \widehat{F}X = FX$. Following type considerations, it is immediate to see that $\widehat{out_F}$, the lifting of the final coalgebra out_F, is the natural candidate to play the role of out_G. Intuitively, this arrow permits to perform single observations to the data structure generated by the recursive process, just propagating the computational effect.

In summary, a monadic unfold will correspond to a function f satisfying this diagram in $\mathcal{C}$:

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & MD \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \widehat{out_F}^\star} \\ M(FA) & \xrightarrow[(\widehat{F}f)^\star]{} & M(FD) \end{array} \tag{19}$$

As we will see below, the solution to this diagram may not be unique, and this will force us to introduce monadic unfold by a fixed point definition. Before addressing this point in detail, let us introduce the notion of monadic coalgebra.

A *monadic F-coalgebra* is a pair formed by an object A (the carrier) and an arrow $g : A \to M(FA)$ (the operation). Like 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 role of *signature*. In view of the definition and choosing a monadic extension $\widehat{F}$, one might then say that a monadic coalgebra $A \to M(FA)$ (in $\mathcal{C}$) is actually a $\widehat{F}$-coalgebra $A \to \widehat{F}A$ when viewed as an arrow in $\mathcal{C}_M$.

A structure preserving mapping between two monadic coalgebras is an arrow between their carriers that preserves their structures, and is *compatible* with their monadic effects. A *monadic coalgebra map* or *homomorphism* between two monadic coalgebras $g : A \to M(FA)$ and $g' : B \to M(FB)$ is an arrow $f : A \to MB$ such that $g' \bullet f = \widehat{F}f \bullet g$. Homomorphisms are closed under composition whenever $\widehat{F}$ preserves Kleisli composition. On the other hand, unit is the identity homomorphism provided that $\widehat{F}$ preserves identities. Moreover, when $\widehat{F}$ preserves identities, the lifting of any homomorphism between two normal coalgebras is an homomorphism between the liftings of the coalgebras. That is, for coalgebras g and g',

$$\widehat{F}\mathsf{unit} = \mathsf{unit} \quad \wedge \quad g' \circ f = Ff \circ g \quad \Rightarrow \quad \widehat{g'} \bullet \widehat{f} = \widehat{F}\widehat{f} \bullet \widehat{g} \tag{20}$$

Now we are in conditions to show that the solution to diagram (19) may not be unique, i.e. there may exist various homomorphisms between the monadic coalgebras g and $\widehat{out_F}$. The following counterexample gives evidences of this fact.

Example 5.2 Recall diagram (19). Let M be the maybe monad $MA = A + 1$, and let F be the functor $S_A = \underline{A} \times I$. Recall from Example 2.2 that S_A captures the signature of streams with elements over A. Thus, the datatype D corresponds to A^∞. Consider the natural transformation

$$\mathsf{fail}_{A,B} = A \xrightarrow{!_A} 1 \xrightarrow{\mathsf{inr}} MB$$

where $!_A$ denotes the unique arrow to the final object 1. That natural transformation represents the function that fails for every input. Functionally expressed, $\mathsf{fail}_{A,B} = \lambda a.\mathsf{inr}(\bot)$. It satisfies an absorption property, namely, for any k and k', $k' \bullet \mathsf{fail} = \mathsf{fail} = \mathsf{fail} \bullet k$.

Next, we show that $\widehat{S_A}\,\mathsf{fail} = \mathsf{fail}$. The proof we present is as general as possible. It can be carried out in any category for which a distribution law $d : A \times (B + C) \to A \times B + A \times C$, inverse of the map $[\mathsf{id}_A \times \mathsf{inl}, \mathsf{id}_A \times \mathsf{inr}]$, exists. A category with product and coproduct for which d exists is said to be *distributive* (**Set** is the typical example). In the particular case of **Cpo**, even though the separated sum is not a coproduct, a distribution law d can be defined as the following strict function:

$$\begin{aligned} d(a, x) = \ & \mathbf{case}\ x\ \mathbf{of} \\ & \mathsf{inl}(b) \ \to\ \mathsf{inl}(a, b) \\ & \mathsf{inr}(c) \ \to\ \mathsf{inr}(a, c) \end{aligned}$$

Observe that, since $[\mathsf{id}_A \times \mathsf{inl}, \mathsf{id}_A \times \mathsf{inr}] \circ \mathsf{inr} = \mathsf{id}_A \times \mathsf{inr}$, it holds that $d \circ (\mathsf{id}_A \times \mathsf{inr}) = \mathsf{inr}$. In this setting, the strength of the maybe monad can then be given by $\tau_{A,B} = (\mathsf{id} + \pi_2) \circ d : A \times (B + 1) \to A \times B + 1$.

Now, we are ready to assert the desired equality:

$$\begin{aligned} \widehat{S_A}\,\mathsf{fail}_{B,C} &= \tau_{A,C} \circ (\mathsf{id}_A \times \mathsf{fail}_{B,C}) \\ &= (\mathsf{id} + \pi_2) \circ d \circ (\mathsf{id}_A \times \mathsf{inr}) \circ (\mathsf{id}_A \times !_B) \\ &= (\mathsf{id} + \pi_2) \circ \mathsf{inr} \circ (\mathsf{id}_A \times !_B) \\ &= \mathsf{inr} \circ \pi_2 \circ (\mathsf{id}_A \times !_B) \\ &= \mathsf{inr} \circ !_B \circ \pi_2 \\ &= \mathsf{inr} \circ !_{A \times B} \\ &= \mathsf{fail}_{A \times B, A \times C} \end{aligned}$$

Therefore, by using the absorption property, we have that

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

which shows that $\mathsf{fail}_{B,A^\infty}$ satisfies diagram (19) for any g.

Now consider the particular case that $g = \mathsf{unit} \circ \langle h, t \rangle : B \to M(A \times B)$ for $h : B \to A$ and $t : B \to B$. Since out_{S_A} is a final coalgebra, the unfold $[\![\langle h, t \rangle]\!]_{S_A} : B \to A^\infty$ is the unique homomorphism from $\langle h, t \rangle$ to out_{S_A}. In addition, from Proposition 4.2 we know that $\widehat{S_A}$ is a lifting. Thus, by (20) it follows that the lifting of the unfold, $\mathsf{unit} \circ [\![\langle h, t \rangle]\!]_{S_A}$, is a homomorphism between the liftings of $\langle h, t \rangle$ and out_{S_A}, and thereby it satisfies diagram (19) for that g.

Therefore, we have found a case where (at least) two different solutions to (19) exist. □

The existence of multiple solutions to diagram (19) forces us to define monadic unfold in a domain-theoretic setting. We then adopt the *least* solution to (19) as the definition of monadic unfold in **Cpo**. In other words, it is defined as the least homomorphism between the monadic coalgebras $g : A \to M(FA)$ and $\widehat{out_F}$. We shall denote it by $[\![g]\!]^M_F : A \to M\mu F$. It is easy to verify that a least homomorphism always exists.

The least homomorphism can be obtained by considering the least fixed point of the functional

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

This functional arises from reversing $\widehat{out_F}$ in diagram (19), which can be done because $\widehat{out_F}$ is an isomorphism (with inverse $\widehat{in_F}$ w.r.t. Kleisli composition). Indeed, it is the result from mapping the isomorphism out_F with functor $(\widehat{-})$, and functors preserve isomorphisms.

Definition 5.3 For $g : A \to M(FA)$, the **monadic unfold** $[\![g]\!]^M_F : A \to M\mu F$ is defined by $\mathsf{fix}(\phi)$. □

Example 5.4 For the datatype of lists, it can be verified that

$$(\mathsf{unit} \circ [\mathsf{nil}, \mathsf{cons}]) \bullet \widehat{L_A} f = [\widehat{\mathsf{nil}}, \widehat{\mathsf{cons}} \bullet (\psi \circ (\mathsf{unit} \times f))]$$

Thus, for $g : B \to M(1 + A \times B)$, the monadic unfold is the following function of type $B \to MA^*$:

$$[\![g]\!]^M_{L_A} = \mathsf{fix}(\phi) \qquad \text{with} \qquad \phi(f) = [\widehat{\mathsf{nil}}, \widehat{\mathsf{cons}} \bullet (\psi \circ (\mathsf{unit} \times f))] \bullet g$$

It can be functionally expressed as follows:

$$\begin{aligned} [\![g]\!]^M(b) = g(b) \star \lambda x.\ \mathbf{case}\ x\ \mathbf{of} & \\ \mathsf{inl}(\bot) &\to \mathsf{unit}(\mathsf{nil}) \\ \mathsf{inr}(a, b') &\to [\![g]\!]^M(b') \star \lambda \ell.\, \mathsf{unit}(\mathsf{cons}(a, \ell)) \end{aligned}$$

□

Example 5.5 For the case of streams, the monadic unfold $[\![g]\!]^M_{S_A}$, for $g : B \to M(A \times B)$, is the function given by

$$[\![g]\!]^M_{S_A} = \mathsf{fix}(\phi) \qquad \text{with} \qquad \phi(f) = \widehat{\mathsf{scons}} \bullet (\tau \circ (\mathsf{id} \times f)) \bullet g$$

That is,

$$[\![g]\!]^M(b) = g(b) \star \lambda(a, b').\, [\![g]\!]^M(b') \star \lambda s.\, \mathsf{unit}(\mathsf{scons}(a, s))$$

In Example 5.2 we saw that, when M is the maybe monad and $g = \mathsf{unit} \circ \langle h, t \rangle$, the functions $\mathsf{fail}_{B,A^\infty}$ and $\mathsf{unit} \circ [\![\langle h, t \rangle]\!]_{S_A}$ are valid solutions to (19). However, neither of them correspond to the monadic unfold, which is given by the completely undefined function $\lambda b. \bot : B \to MA^\infty$. The reason why in this case the monadic unfold 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. However, it does not correspond to the expected computational behaviour. This example shows the inconvenience of using the maybe monad in combination with the generation of infinite data structures. □

Example 5.6 The function $\mathsf{zip} : A^* \times B^* \to (A \times B)^*$ is well-known in functional programming [4]. It takes a pair of lists and returns a list of pairs of elements at corresponding positions. Jeuring and Jansson [17] give a polytypic version of zip, that they call $\mathsf{pzip} : DA \times DB \to M(D(A \times B))$, which zips two terms of type DA and DB, respectively, where D is a type functor induced by a regular bifunctor F and $MA = A + 1$ is the maybe monad. The maybe monad is necessary to control that the two terms being zipped have the same shape.

The function pzip can be represented as a monadic unfold

$$\mathsf{pzip} = [\!(\mathsf{pzstep})\!]^M_{F_{A\times B}}$$

with $\mathsf{pzstep} = \mathsf{fzip} \circ (out_{F_A} \times out_{F_B})$. The function

$$\mathsf{fzip} : F(A,X) \times F(B,Y) \to M(F_{A\times B}(X \times Y))$$

is a natural transformation that inspects the outermost constructors of two terms and proceeds accordingly. Its definition is given by induction on the structure of F (see [17, 32]). □

An *identity law* for monadic unfold holds provided that M is strictness-preserving and the unit of the monad is strict.

$$M \text{ strictness-preserving} \;\wedge\; \mathsf{unit}\text{ strict} \;\Rightarrow\; [\!(\widehat{out_F})\!]^M_F = \mathsf{unit} \tag{21}$$

Actually, this law states exactly the same as the homonymous one for monadic fold. Indeed, observe that the monadic unfold $[\!(\widehat{out_F})\!]^M_F$ and the monadic fold $(\!|\widehat{in_F}|\!)^M_F$ are the same function.

Recall that homomorphisms between monadic coalgebras are closed under composition whenever $\widehat{F}$ preserves compositions. When that is the case, we can state a *fusion-law*,

$$M \text{ strictness-preserving} \;\wedge\; g \bullet f = \widehat{F}f \bullet g' \;\Rightarrow\; [\!(g)\!]^M_F \bullet f = [\!(g')\!]^M_F \tag{22}$$

which can be verified by a simple fixpoint induction.

Homomorphisms between two monadic coalgebras are arrows that themselves may produce monadic effects, compatible with that of the monadic coalgebras. It is also possible to define a class of structure-preserving mappings that are given by *pure* functions between the carriers of monadic coalgebras. We say that an arrow $f : A \to B$ is a *pure homomorphism* between $g : A \to M(FA)$ and $g' : B \to M(FB)$ if $g' \circ f = M(Ff) \circ g$. Note that two monadic coalgebras are connected by a homomorphism of this kind when their monadic effects coincide on f-related inputs. That is, 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. The arrow $M(Ff)$ simply maps a value of type FA to a value of type FB, propagating the monadic effect.

Associated with pure homomorphisms we can state a law that we call *pure fusion*.

$$g \circ f = M(Ff) \circ g' \;\Rightarrow\; [\!(g)\!]^M_F \circ f = [\!(g')\!]^M_F \tag{23}$$

It can also be proved straightforwardly by fixpoint induction.

Finally, we present a law, called *munfold-map-fusion*, that relates monadic unfolds with type functors. For $f : A \to B$,

$$M \text{ strictness-preserving} \;\wedge\; M(Df) \circ [\!(g)\!]^M_{F_A} = [\!(M(F(f,\mathsf{id})) \circ g)\!]^M_{F_B} \tag{24}$$

This law is an instance of Proposition 5.9 (to be presented in the next subsection), since, by definition, the action on arrows of a type functor is given by a fold (see Subsection 2.4).

Example 5.7 Recall function $\mathsf{pzip} : DA \times DB \to M(D(A \times B))$ from Example 5.6. We can define a polytypic function $\mathsf{pzipWith}(f) : DA \times DB \to M(DZ)$ that maps each pair in the pzip of two data structures with function $f : A \times B \to Z$ as follows:

$$\mathsf{pzipWith}(f) \;=\; DA \times DB \xrightarrow{\mathsf{pzip}} M(D(A \times B)) \xrightarrow{M(Df)} M(DZ)$$

Since pzip is a monadic unfold, by (24) we then obtain that $\mathsf{pzipWith}(f)$ is a monadic unfold too.

$$\mathsf{pzipWith}(f) = [\!(MF(f,\mathsf{id}) \circ \mathsf{pzstep})\!]^M_{F_Z}$$

□

5.3 Monadic Hylomorphism

The most general version of monadic hylomorphism one can think of consists of the (Kleisli) composition of a monadic fold with a monadic unfold. Given a monadic algebra $h : FA \to MA$ and a monadic coalgebra $g : B \to M(FB)$, let us denote it by $[\![h, g]\!]^M_F : B \to MA$. Thus,

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

This operator represents functions that produce effects in the phase of construction as well as in the phase of consumption of the intermediate data structure. For example, a typical case may be the execution of input/output operations during both recursive processes. The drawback with functions of this kind is that they cannot be always transformed into a single function that avoids the construction of the intermediate data structure of type μF. The problem is that, in order to apply such a transformation, we need that $\widehat{F}$ preserve compositions, which is not satisfied by the monadic extension of every regular functor on the state monad, for example.

Motivated by this fact, we will consider instead a restricted version of monadic hylomorphism, defined as the composition of a monadic unfold with (the lifting of) a fold.[3]

Definition 5.8 Given an algebra $h : FA \to A$ and a monadic coalgebra $g : B \to M(FB)$, we define the **monadic hylomorphism** as a function $\langle\!|h, g|\!\rangle^M_F : B \to MA$ such that

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

□

Observe that functions of this kind only produce an effect in the phase of construction of the intermediate data structure. Monadic hylos are typical in programming practice. For example, as we shall see in Subsection 6.2, they correspond to the application of semantic actions (specified by a fold) to the parse trees generated by a parser (a monadic unfold).

By the identity law of fold, (1), it follows that monadic unfold is an instance of this operator:

$$[\![g]\!]^M_F = \langle\!|in_F, g|\!\rangle^M_F$$

The following is a factorization property similar to that of hylos. It states that any monadic hylo can be transformed into a monolithic function that avoids the generation of the intermediate data structure.

Proposition 5.9 *If M is a strictness-preserving functor, then*

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

Proof. The proof is 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:

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

The function $M\perp : M\mu F \to MA$ is strict because M is strictness-preserving. Hence, its composition with $\perp : B \to M\mu F$ is bottom $\perp : B \to MA$.

Inductive Case: Assume that $f = Mf_2 \circ f_1$. Recall that, for every f, $\widehat{f^\star} = (\mathsf{unit} \circ f)^\star = Mf$. Since $Mout_F \circ Min_F = M(out_F \circ in_F) = \mathsf{id}$, we have that

[3] Another possible variant of monadic hylomorphism can be obtained by composing an unfold with a monadic fold. It reduces to a normal hylomorphism, since every monadic fold is a fold. This variant will not be considered in the paper.

$$M\chi(f_2) \circ \phi(f_1) = Mh \circ M(Ff_2) \circ (\widehat{F}f_1)^\star \circ g$$

Observe that $M(Ff_2) \circ (\widehat{F}f_1)^\star = (M(Ff_2) \circ \widehat{F}f_1)^\star$. By equation (13), $M(Ff_2) \circ \widehat{F}f_1 = \widehat{F}(Mf_2 \circ f_1)$, and by induction hypothesis $\widehat{F}(Mf_2 \circ f_1) = \widehat{F}f$. Summing up,

$$M\chi(f_2) \circ \phi(f_1) = Mh \circ M(Ff_2) \circ (\widehat{F}f_1)^\star \circ g = Mh \circ (\widehat{F}f)^\star \circ g = \psi(f)$$

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

Example 5.10 For the datatype of lists, given $g : B \to M(1 + A \times B)$ and $h = [h_1, h_2] : 1 + A \times C \to C$, the monadic hylo $\langle\!|h, g|\!\rangle^M_{L_A} : B \to MC$ is given by:

$$\langle\!|h, g|\!\rangle^M_{L_A} = \mathsf{fix}(\lambda f.\ [\widehat{h_1}, \widehat{h_2}^\star \circ \psi \circ (\mathsf{unit} \times f)]^\star \circ g)$$

It can be functionally expressed as follows,

$$\begin{aligned}\langle\!|h, g|\!\rangle^M(b) = \ & g(b) \star \lambda x.\ \mathbf{case}\ x\ \mathbf{of} \\ & \mathsf{inl}(\bot) \ \to\ \mathsf{unit}(h_1) \\ & \mathsf{inr}(a, b') \ \to\ \langle\!|h, g|\!\rangle^M(b') \star \lambda c.\, \mathsf{unit}(h_2(a, c))\end{aligned}$$

□

The following fusion laws can be established in combination with homomorphisms of algebras and monadic coalgebras.

$$f \text{ strict } \wedge\ f \circ h = h' \circ Ff \quad \Rightarrow \quad Mf \circ \langle\!|h, g|\!\rangle^M_F = \langle\!|h', g|\!\rangle^M_F \tag{25}$$

$$g \circ f = M(Ff) \circ g' \quad \Rightarrow \quad \langle\!|h, g|\!\rangle^M_F \circ f = \langle\!|h, g'|\!\rangle^M_F \tag{26}$$

In (25), M needs to be strictness-preserving. These laws follow directly from Definition 5.8 and the fusion laws (2) and (23) for fold and monadic unfold, respectively.

To conclude, we present an Acid Rain Theorem for monadic hylo. We will need a notion of transformer between coalgebras and monadic coalgebras. A function $\boldsymbol{T} : (A \to FA) \to (A \to MGA)$ is said to be a *transformer* when for every $f : A \to B$, and coalgebras $g : A \to FA$ and $g' : B \to FB$, if $g' \circ f = Ff \circ g$ then $\boldsymbol{T}(g') \circ f = MGf \circ \boldsymbol{T}(g)$.

Theorem 5.11 (Acid Rain)

mhylo-fold fusion: *Let M be strictness-preserving and let $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ be a transformer.*

$$h \text{ strict } \ \Rightarrow\ \ M(\!|h|\!)_F \circ \langle\!|\, \boldsymbol{T}(in_F), g|\!\rangle^M_G = \langle\!|\, \boldsymbol{T}(h), g|\!\rangle^M_G$$

unfold-mhylo fusion: *Let $\boldsymbol{T} : (A \to FA) \to (A \to MGA)$ be a transformer.*

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

Proof. The proof of mhylo-fold-fusion is similar to that of hylo-fold-fusion presented in Theorem 2.6, but now relying on the application of law (25). The proof of unfold-mhylo-fusion is as follows. By definition of transformer we get that $[\!(g)\!]_F : B \to \mu F$ is a pure homomorphism between the monadic G-coalgebras $\boldsymbol{T}(g) : B \to MGB$ and $\boldsymbol{T}(out_F) : \mu F \to MG\mu F$. Therefore, by (26) we arrive at the desired result. □

An application of mhylo-fold-fusion will be shown in Subsection 6.2. Since every monadic unfold is a special case of a monadic hylo, we can write a specialization of unfold-mhylo-fusion in terms of monadic unfold.

$$[\!(\, \boldsymbol{T}(out_F))\!]^M_G \circ [\!(g)\!]_F = [\!(\, \boldsymbol{T}(g))\!]^M_G \tag{27}$$

Example 5.12 Recall again the function pzip from Example 5.6. Suppose that the data structures being zipped by pzip are generated, resp., by unfolds $[\![(g_1)]\!]_{F_A} : X \to DA$ and $[\![(g_2)]\!]_{F_B} : Y \to DB$. Thus, we have the following situation:

$$X \times Y \xrightarrow{[\![(g_1)]\!]_{F_A} \times [\![(g_2)]\!]_{F_B}} DA \times DB \xrightarrow{\mathsf{pzip}} M(D(A \times B))$$

The construction of the data structures may be avoided if we fuse both parts. When we try to work out this case with a fusion law for unfold, we observe that there is no law in Section 2 that applies (even considering generalizations for simultaneously generated data structures). The reason is that pzip does not meet the structure of any of the functions that take part in those laws. The desired deforestation can then be performed when we use the fact that pzip is representable by a monadic unfold, and look at the laws for this operator. We need to consider the following generalization of law (27) for the case of simultaneously generated data structures:

$$[\![\, \boldsymbol{T}(out_{F_1}, out_{F_2})]\!]^M_G \circ ([\![(g_1)]\!]_{F_1} \times [\![(g_2)]\!]_{F_2}) = [\![\, \boldsymbol{T}(g_1, g_2)]\!]^M_G$$

Define that $\boldsymbol{T}(g_1, g_2) = \mathsf{fzip} \circ (g_1 \times g_2)$. That way, $\mathsf{pzstep} = \boldsymbol{T}(out_{F_A}, out_{F_B})$. Therefore, by using this law the composition above can be transformed into a monadic unfold that does not generate the data structures.

$$\mathsf{pzip} \circ ([\![(g_1)]\!] \times [\![(g_2)]\!]) = [\![(\mathsf{fzip} \circ (g_1 \times g_2))]\!]^M_{F_{A \times B}}$$

□

6 Examples

The aim of this section is to illustrate two non-trivial examples of monadic unfolds and hylos as well as some uses of their calculational laws. In the first example, we show that the core of a graph traversal algorithm (such as DFS or BFS) is given by a monadic unfold. The second one shows that the recursive structure of a monadic parser (a form of recursive descent parser) is characterized by monadic hylo. Further examples can be found in [32].

6.1 Graph Traversals

By *graph traversal* we understand a function that takes a list of roots (entry points to a graph) and returns a list containing the vertices met along the way. We consider that a graph G is given by an *adjacency list function* $\mathsf{adj} : V \to V^*$ which returns the adjacency list for each vertex. This representation of graphs is sufficiently abstract, but at the same time useful for algorithmic purposes.

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

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

We show that it is possible to give a formulation of graph traversal in terms of a monadic unfold that handles the set of visited nodes in a 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. That representation permits $O(1)$ time insertions and lookups when implemented by a *mutable array* (i.e. by an array with destructive updates). In that case, the operations on sets correspond to primitives that work over the mutable array directly. To be able to handle such imperative operations in a functional setting, a well-established technique is to encapsulate them in an abstract data type based on the state monad [39]:

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

In addition to unit and $\star$, it possesses these operations:

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

given by

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

These monadic operations guarantee safe, in-place update whenever they manipulate the set in a *single-threaded* manner, i.e not duplicating it (see [34, 39]). To ensure this 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).[4]

Based on this abstract data type, we can define graph traversal as a function $\mathsf{graphtrav} : V^* \to V^*$ given by

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

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

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

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 the function $\mathsf{mdropS} : V^* \to MV^*$ which is defined by:

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

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

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

[4]Note that the introduction of the abstract data type still makes sense when we adopt a purely functional representation for sets. We have made reference to the imperative solution only for the sake of illustration.

and thus,

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

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

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

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,

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

Let us call breadth-first to the instance of gtrav obtained in this case.

Representation Change.

As we have just seen, a breadth-first traversal handles the list of roots as a queue. However, operationally speaking, it is well-known that the list representation of queues is quite inefficient. In fact, when an element is enqueued, it is appended at the end of the list, and this takes time proportional to the length of the list. To eliminate this inefficiency, with help of pure fusion, law (23), we will calculate a new formulation of breadth-first traversal on a supposed better representation of queues (see [30] for fast functional implementations of queues; also [4].)

Suppose we are given an abstract data type $\mathcal{Q}(A)$ 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 $(\ell, q) \in V^* \times \mathcal{Q}(V^*)$, such that ℓ is the adjacency list currently active (i.e. the one being inspected) and q is a queue containing the adjacency lists waiting for activation. When the list ℓ empties, a new list is 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 by

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

Now, consider the function $\mathsf{change} : V^* \times \mathcal{Q}(V^*) \to V^*$ that translates from one representation of queues to the other. It is defined by

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

where $\mathsf{q2list}$ 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 homomorphism between the monadic coalgebras qopen and gopen. Therefore, by applying pure fusion we obtain a new monadic unfold that corresponds to the desired version of breadth-first traversal on the alternative representation of queues.

$$[\![\mathsf{qopen}]\!]^M_{L_V} = \mathsf{breadth\text{-}first} \circ \mathsf{change}$$

Search Procedures

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

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

Since $\mathsf{graphtrav} = \emptyset^M \circ \mathsf{gtrav}$, by naturality of $\emptyset^M$, i.e.

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

for every f, we can push the catamorphism into the monad,

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

and because gtrav is a monadic unfold,

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

Consider the case in that the fold is the function $\mathsf{filter}(p) = (\!|\mathsf{fil}|\!)_{L_V} : V^* \to V^*$ that removes all the elements of a list that do not satisfy a predicate $p : V \to \mathsf{bool}$, with

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

In that case, the composition

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

represents a *search procedure* that explores a graph in determining order with the aim at finding all vertices fulfilling a given predicate. Like before, by specifying concrete traversal strategies, typical search procedures like *depth-first search* or *breadth-first search* can be 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. On the other hand, when a parser succeeds, 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, type A usually corresponds to the type of parse trees, which describe the structure of the recognized strings. However, functional parsers are typically presented as functions that return values of an arbitrary A.

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 from 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.

As observed by Wadler [38, 39], functional parsers can be structured using the so-called *parser monad*. In the monadic approach, the combinators for sequencing and alternation are given by primitive operations of the monad. This permits to focus on the relevant structure of parsers.

The purpose of this example is to give a formal characterization of the recursive structure of recursive descent parsers with help of monadic unfolds and hylomorphisms. To the best of our knowledge, this constitutes the first attempt to give such a characterization.

Following we summarize the main results we achieve. We regard the definition of a functional parser as the composition of two phases, namely (i) *syntax analysis*, process by which a string is recognized and a parse tree generated, and (ii) *semantic actions*, process by which values of any kind are calculated with the parse trees that result from the syntax analysis. Our first result is to recognize that the syntax analysis phase can be expressed by a monadic unfold on the datatype representing the *concrete syntax* of the language (i.e. the datatype of parse trees). Semantic actions are usually defined by induction over the structure of parse trees (i.e. by a fold). Thus, the application of semantic actions after the syntax analysis phase can be expressed by a monadic hylomorphism. In other words, the shape of recursion followed by a functional parser is dictated by the signature of parse trees, even though the trees never appear explicitly.

The Parser Monad

The *parser monad* [38, 39, 25] is defined by

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

where $\bar{f}$ denotes the uncurry of $f : A \to MB$, i.e. $\bar{f}(x, y) = f(x)(y)$. 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 $[f(a_1)(s_1), \ldots, f(a_n)(s_n)]$ whose elements are themselves lists 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 particular case of a *monad with a zero and a plus* [26, 25]. That is, it is 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.

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

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:

$$\begin{array}{rl} \text{(Expressions)} & \mathit{exp} \;::=\; \mathit{term}\ \text{`+'}\ \mathit{exp} \mid \mathit{term} \\ \text{(Terms)} & \mathit{term} ::=\; \mathit{fact}\ \text{`*'}\ \mathit{term} \mid \mathit{fact} \\ \text{(Factors)} & \mathit{fact} \;::=\; \text{`('}\ \mathit{exp}\ \text{`)'} \mid \mathit{num} \end{array}$$

where *num* stands for the set of numerals. We will assume that each numeral $\underline{n}$ is given by the natural number n that it represents. The set of terminal symbols corresponding to this grammar is thus defined as $T = \{\text{`+'}, \text{`*'}, \text{`('}, \text{`)'}\} \cup \mathbb{N}$.

From this grammar, we can construct the following monadic parser, which recognizes expressions and returns the natural number that arises from evaluating them. The parsers for the terminals $\{\text{`+'}, \text{`*'}, \text{`('}, \text{`)'}\}$ are given in terms of the elementary parser tok. The parser for numerals is given as a combination of item and filter $\rhd$ with predicate $p(x) = x \in \mathbb{N}$ (written as $(\in \mathbb{N})$). We write $\mathsf{add} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $\mathsf{prod} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ to denote addition and product.

$$\begin{array}{l} \mathsf{parser}_e \;:\; M\mathbb{N} \\ \mathsf{parser}_e = (\mathsf{parser}_t \star \lambda t.\, \mathsf{tok}(\text{`+'}) \star \lambda a.\, \mathsf{parser}_e \star \lambda e.\, \mathsf{unit}(\mathsf{add}(t, e))) \oplus \mathsf{parser}_t \\ \\ \mathsf{parser}_t \;:\; M\mathbb{N} \\ \mathsf{parser}_t = (\mathsf{parser}_f \star \lambda f.\, \mathsf{tok}(\text{`*'}) \star \lambda p.\, \mathsf{parser}_t \star \lambda t.\, \mathsf{unit}(\mathsf{prod}(f, t))) \oplus \mathsf{parser}_f \\ \\ \mathsf{parser}_f \;:\; M\mathbb{N} \\ \mathsf{parser}_f = (\mathsf{tok}(\text{`('}) \star \lambda l.\, \mathsf{parser}_e \star \lambda e.\, \mathsf{tok}(\text{`)'}) \star \lambda r.\, \mathsf{unit}(e)) \oplus (\mathsf{item} \rhd (\in \mathbb{N})) \end{array}$$

Syntax Analysis

The technique to be described next is completely general, in the sense that it can be used for any context-free language. Our exposition, however, will essentially focus on the language of arithmetic expressions.

By *syntax analysis* we understand the process by which strings of tokens are recognized and returned in the form of parse trees. Thus, to build 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 we have these datatype declarations:

$$\begin{array}{lcl@{\qquad}lcl}
\mathsf{Exp} & = & \mathsf{sum}(\mathsf{Term},\mathsf{Plus},\mathsf{Exp}) \mid \mathsf{term}(\mathsf{Term}) & \mathsf{Mult} & = & \mathsf{multOp} \\
\mathsf{Term} & = & \mathsf{mult}(\mathsf{Factor},\mathsf{Mult},\mathsf{Term}) \mid \mathsf{factor}(\mathsf{Factor}) & \mathsf{Left} & = & \mathsf{left} \\
\mathsf{Factor} & = & \mathsf{brackets}(\mathsf{Left},\mathsf{Exp},\mathsf{Right}) \mid \mathsf{num}(\mathsf{Num}) & \mathsf{Right} & = & \mathsf{right} \\
\mathsf{Plus} & = & \mathsf{plusOp} & \mathsf{Num} & = & \mathbb{N}
\end{array}$$

The reason for introducing a datatype for each token is because parse trees structurally represent *all* details of recognized strings, inclusive connectives, operators and parenthesis. As we shall see below, the presence of these datatypes for the tokens turns out to be determinant for achieving a formulation of the syntax analyzer in terms of a monadic unfold, as they force invocations to the parsers for the tokens 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 $\mathsf{syntax}_X : M\mathsf{D}_X$ that yields parse trees of the corresponding type D_X. For instance, for expressions, $\mathsf{syntax}_e : M\mathsf{Exp}$. However, to achieve a formulation of the syntax analyzer in terms of monadic unfold, we need to use a trick consisting of regarding these functions as functions from the unit type. So, for example, $\mathsf{syntax}_e : 1 \to M\mathsf{Exp}$. (The usefulness of considering the unit type will become clear later.) The logical structure of each function syntax_X is dictated by the recursive structure of the parse trees it returns (i.e. by the structure of D_X). Following this observation, we will express each one as a monadic unfold. It is interesting to note that these functions turn out to be mutually recursive, since so are the parse tree types. Therefore, in the present example we will have eight mutually-recursive monadic unfolds.

To build up the monadic unfolds, we need to identify the functors that capture the signature of the parse tree types. When various datatypes are defined by simultaneous recursion, their functors reflect this fact by having so many variables as involved datatypes (see e.g. [7, 14]). Concretely, in this example the functors are on eight variables. Let us write $\vec{A} = (A_e, A_t, A_f, A_p, A_m, A_l, A_r, A_n)$ for short. Then,

$$\begin{array}{lll@{\qquad}l}
F_e\,\vec{A} = A_t \times A_p \times A_e + A_t & F_p\,\vec{A} = 1 & & \\
F_t\,\vec{A} = A_f \times A_m \times A_t + A_f & F_m\,\vec{A} = 1 & & F_r\,\vec{A} = 1 \\
F_f\,\vec{A} = A_l \times A_e \times A_r + A_n & F_l\,\vec{A} = 1 & & F_n\,\vec{A} = \mathbb{N}
\end{array}$$

Of course, the datatypes for the tokens are neither recursive nor depend on others. However, if we consider them as part of the simultaneous recursion definition, then we can assign to each one a variable position in all functors. Their presence as variables rather than as constant types within functors permits to (automatically) force invocations to their corresponding parser functions. The parser functions corresponding to these non-recursive types are obviously non-recursive, but even though they can be seen as monadic unfolds. Omitting the unit type,

$$\begin{array}{ll@{\qquad}ll}
\mathsf{syntax}_p = \mathsf{tok}(\text{`+'}) \star \lambda p.\,\mathsf{unit}(\mathsf{plusOp}) & & \mathsf{syntax}_r = \mathsf{tok}(\text{`)'}) \star \lambda r.\,\mathsf{unit}(\mathsf{right}) \\
\mathsf{syntax}_m = \mathsf{tok}(\text{`*'}) \star \lambda m.\,\mathsf{unit}(\mathsf{multOp}) & & \mathsf{syntax}_n = \mathsf{item} \rhd (\in \mathbb{N}) \\
\mathsf{syntax}_l = \mathsf{tok}(\text{`('}) \star \lambda l.\,\mathsf{unit}(\mathsf{left}) & &
\end{array}$$

Now we address the definition of the parser functions for the syntactic categories. These are indeed mutually recursive. Let us begin with the analyzer for whole expressions, syntax_e. It can be expressed as a monadic unfold $[\![(g_e)]\!]^M_{F_e}$ on certain monadic coalgebra g_e. This means that it is the least function satisfying this diagram:

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

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

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

Also,

$$\widehat{F}_e(f_1,\ldots,f_8) = [\widehat{\mathsf{inl}}\bullet\psi^{(3)}\circ(f_2\times f_4\times f_1), \widehat{\mathsf{inr}}\bullet f_2]$$

where $\psi^{(n)} : MA_1\times\cdots\times MA_n \to M(A_1\times\cdots\times A_n)$ is the n-ary generalization of the product distribution for the monad ($\psi^{(2)}=\psi$). Therefore,

$$M[\mathsf{sum},\mathsf{term}]\circ\widehat{F}_e(f_1,\ldots,f_8) = [\widehat{\mathsf{sum}}\bullet\psi^{(3)}\circ(f_2\times f_4\times f_1), \widehat{\mathsf{term}}\bullet f_2]$$

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*. 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, i.e. the parsing *goals* of the productions. In this sense, the monadic coalgebra behaves as a *trigger*. The product of 1's models the fact that the parsing goals need to be sequenced. Note also how the existence of two alternative productions for *exp* in the concrete syntax definition 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 $h=[h_1,h_2]$,

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

Summing up, we have that:

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

Omitting the applications to the unit type,

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

which coincides 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 unfolds $[\![(g_t)]\!]^M_{F_t}$ and $[\![(g_f)]\!]^M_{F_f}$, respectively, where

$$g_t = g_f = g_e$$

Like above, by formal manipulation we can deduce that:

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

$$\mathsf{syntax}_f = \oplus\circ\langle\widehat{\mathsf{brackets}}\bullet\psi^{(3)}\circ\langle\mathsf{syntax}_l,\mathsf{syntax}_e,\mathsf{syntax}_r\rangle, \widehat{\mathsf{num}}\bullet\mathsf{syntax}_n\rangle$$

Adding Semantic Actions

Now we want to incorporate semantic actions to a parser, in the sense of computing 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. In our setting, this can be regarded as the definition of a fold. The application of semantic actions after a syntax analyzer is what is normally called a monadic parser:

$$\mathsf{parser} \;=\; 1 \xrightarrow{\;\mathsf{syntax}\;} M\,\mathsf{D} \xrightarrow{\;M\,\mathsf{semantics}\;} MA$$

Since the syntax analyzer is given by a monadic unfold $[\![(g)]\!]^M$ and the semantic actions by a fold $(\!|h|\!)$, their composition is given by a monadic hylomorphism:

$$\mathsf{parser} = \langle\!|h, g|\!\rangle^M \tag{28}$$

Using Proposition 5.9, the function parser can be transformed into a function that does not generate the parse trees. When inlined and simplified, that function coincides with the expression of the monadic parser one would have written by hand. Another interpretation for (28) is the following:

> *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 from expressing a syntax analyzer as a monadic unfold and a parser as a monadic hylomorphism. First of all, this says that we can construct a parser in a modular way. That is, we can develop separately each phase of the parser and at the end join them together into a single function that performs both tasks, but avoids the generation of parse trees. In addition, the representation of monadic parsers in terms of monadic hylomorphism permits to perform formal reasoning with them. For example, now they can be the subject of fusion transformations. Fusion transformations are often necessary for the semantics phase, because the semantic actions usually represent complex operations of an interpreter or a compiler for the given language.

As shown by Meijer [22], the semantic actions of an interpreter/compiler can be developed in a modular way by using a calculational approach like the followed in this paper. However, Meijer's starting-point is the *abstract syntax* of the language. Therefore, to be able to couple Meijer's development with the result of a syntax analyzer, we need to convert from parse trees to *abstract syntax trees*. Roughly speaking, consider that F is the signature of the abstract syntax and G the one of the concrete syntax. The function $\mathsf{c2a} : \mu G \to \mu F$, that maps from concrete syntax to abstract syntax, can be specified by a fold $(\!|\, \boldsymbol{T}(in_F)\,|\!)_G$ whose target algebra is defined in terms of a transformer $\boldsymbol{T} : (FA \to A) \to (GA \to A)$. The semantic actions are now specified on the abstract syntax, i.e. $\mathsf{sem} = (\!|h|\!)_F : \mu F \to A$. With these components we can build the following composite:

$$1 \xrightarrow{\;\mathsf{syntax}\;} M\,\mu G \xrightarrow{\;M\,\mathsf{c2a}\;} M\mu F \xrightarrow{\;M\,\mathsf{sem}\;} MA$$

Observe that

$$M\,\mathsf{c2a} \circ \mathsf{syntax} = \langle\!|\, \boldsymbol{T}(in_F), g|\!\rangle_G^M$$

Thus, we can now apply the mhylo-fold-fusion law from Theorem 5.11,

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

obtaining that the complete interpreter/compiler is given by a monadic hylo, which can in turn be transformed into a single function that, not only avoids the construction of parse trees, but also the one of abstract syntax trees.

7 Concluding Remarks

In this paper, we addressed the definition of recursive operators capable of representing functions with effects, and studied their associated calculational theory. The examples presented aimed at showing that those operators capture functions commonly used in practice.

A possible direction for future research is the design and implementation of a transformation system that automatically performs deforestation on programs with effects by essentially using the Acid Rain Theorem corresponding to monadic hylomorphism. Such a system would constitute an excelent framework where testing the effectiveness of fusion laws on real programs. An experience in this line is being carried out in the context of a transformation system, called HYLO [31], which performs deforestation on purely-functional recursive programs by applying the Acid Rain Theorem corresponding to hylomorphism. Our proposal is therefore the development of a system analogous to HYLO, but based on monadic hylomorphism. That system would be a conservative extension of HYLO, in the sense that it could still apply deforestation to purely-functional programs. The reason is that any purely-functional program can be viewed as a monadic program on the identity monad.

Acknowledgements. I would like to thank one of the anonymous TCS referees for helpful suggestions and Gustavo Betarte for proofreadings and comments. This work has been partially supported by a DAAD scholarship. Diagrams were drawn using Paul Taylor's macros.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[2] L. Augusteijn. Sorting Morphisms. In *Advanced Functional Programming,* LNCS 1608. Springer-Verlag, 1999.

[3] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming - An Introduction -. In *Advanced Functional Programming,* LNCS 1608. Springer-Verlag, 1999.

[4] R. Bird. *Introduction to Functional Programming using Haskell,* 2nd edition. Prentice-Hall, UK, 1998.

[5] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, UK, 1997.

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

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

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

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

[10] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *Proc. 3rd. ACM SIGPLAN International Conference on Functional Programming.* ACM, September 1998.

[11] R. Hoofman. The Theory of Semi-Functors. *Mathematical Structures in Computer Science*, 3(1):93–128, 1993.

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

[13] G. Hutton. Fold and Unfold for Program Semantics. In *Proc. 3rd. ACM SIGPLAN International Conference on Functional Programming.* ACM, September 1998.

[14] H. Iwasaki, Z. Hu, and M. Takeichi. Towards Manipulation of Mutually Recursive Functions. In *3rd. Fuji International Symposium on Functional and Logic Programming (FLOPS'98).* World Scientific, April 1998. Available from `http://www.ipl.t.u-tokyo.ac.jp/~hu/`.

[15] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, 62:222–259, 1997.

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

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

[18] S. Peyton Jones and J. Launchbury. Lazy functional state threads. In *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'94)*, pages 24–35, 1994.

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

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

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

[22] E. Meijer. More Advice on Proving a Compiler Correct: Improve a Correct Compiler. In *PHOENIX Seminar and Workshop on Declarative Programming,* volume 91 of Workshops in Computer Science. Springer-Verlag, 1992.

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

[24] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of Functional Programming Languages and Computer Architecture'95*, pages 324–333, 1995.

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

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

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

[28] E. Moggi, G. Bellè, and C.B. Jay. Monads, shapely functors and traversals. Technical Report DISI-TR-98-06, Università di Genova, 1998.

[29] 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.

[30] C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, 1995.

[31] 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.

[32] A. Pardo. A Calculational Approach to Recursive Programs with Effects, forthcoming PhD Thesis. Technische Universität Darmstadt, 1999.

[33] 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.

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

[35] A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Proceedings of Functional Programming Languages and Computer Architecture'95*, 1995.

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

[37] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

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

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