# Towards Merging Recursion and Comonads

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

**Abstract**

Comonads are mathematical structures that account naturally for effects that derive from the context in which a program is executed. This paper reports ongoing work on the interaction between recursion and comonads. Two applications are shown that naturally lead to versions of a *comonadic fold* operator on the product comonad. Both versions capture functions that require extra arguments for their computation and are related with the notion of strong datatype.

## 1 Introduction

One of the main features of recursive operators derivable from datatype definitions is that they impose a structure upon programs which can be exploited for program transformation. Recursive operators structure functional programs according to the data structures they traverse or generate and come equipped with a battery of algebraic laws, also derivable from type definitions, which are used in program calculations [24, 11, 5, 15]. Some of these laws, the so-called *fusion laws*, are particularly interesting in practice since they enclose specific cases of deforestation, a program transformation technique that permits to remove intermediate data structures from programs. Functional programs can also be structured according to the effects they produce. This can be possibe by using *monads* [33] as structuring device. Monads are well-known mathematical structures with wide application in programming [30, 20] and formal semantics [26, 31, 32].

Previous works [12, 17, 25, 28] have studied how to combine both structuring mechanisms, giving rise to recursive operators that deal with effects modeled by monads. A result that can be concluded from those works is that encapsulating effects with monads leads to a smooth framework for reasoning about programs with effects. In fact, monads permit to focus on the relevant structure of programs disregarding details about the specific effect that a program produces.

*Comonads* are mathematical structures, dual to monads in a categorical sense, that have been used almost entirely in semantics (see e.g. [6, 7, 31, 32]). In recent years, however, there has been a growing interest in investigating the usefulness that comonads may have in programming. Some results in this concern are given by Kieburtz [21], who argues that comonads account naturally for effects that derive from program context.

This paper reports ongoing research concerning the interaction between *comonads* and *recursion*. Like with monads, the aim of this work is the study of recursive operators with effects, now modeled by comonads. Among our goals is the derivation of algebraic laws for such operators, in particular, fusion laws, since they enclose deforestation cases for new recursion patterns. In addition to discussing general aspects of comonads, we focus our study on a particular comonad, namely, the *product comonad*. This comonad permits us to represent two common classes of recursive definitions: (i) functions with extra fixed parameters, and (ii) functions with accumulating parameters. For each of these classes, we introduce an structural recursive operator, a variant of the traditional fold operator, that deals with the parameters within the product comonad. We also present calculational laws associated with these operators. In those laws it is possibe to observe that, like monads, comonads permit to focus on the relevant structure of programs, hiding details about the particular effect.

The remainder of this paper is organized as follows. Section 2 is about datatype theory. Section 3 introduces comonads and discusses distributive laws between functors and comonads. Distributive laws are the basis for the definition of comonadic extensions of functors, which are the constructions that capture the shape of recursion of comonadic operators. In section 4 we review previous work on recursive functions with (fixed) parameters [10, 29], but now from a comonadic point of view. This leads to the definition of a *fold with parameters*, which constitutes our first example of a comonadic fold. In Section 5, we show how a definition of a *fold with accumulating parameters*, another form of comonadic fold, can be obtained from that of fold with parameters by including accumulations as part of the comonadic extension of the base functor. Finally, in Section 6 we draw some conclusions and describe future work.

# 2 Recursive Datatypes

This section presents the mathematical framework the paper is based on and fixes some notation. We describe the essentials of the category-theoretic explanation of inductive and coinductive datatypes, the definition of structural recursive functions in that setting and some of their algebraic laws. Further details on these topics can be found in e.g. [22, 11, 5, 1, 15].

## 2.1 Preliminaries

In the categorical approach to recursive types, types are modeled by objects of a category $\mathcal{C}$, and functions (operations, programs) are modelled by morphisms of this category. In this setting, type constructors correspond to endofunctors on $\mathcal{C}$ (i.e. functors from $\mathcal{C}$ to $\mathcal{C}$). We shall assume that $\mathcal{C}$ is a category with finite products $(\times, 1)$ and finite coproducts $(+, 0)$, where $0/1$ denotes the initial/final object of $\mathcal{C}$. The leading example of such a category is **Set**, the category of sets and total functions.

The unique arrow from $A$ to 1 is written $!_A$. We write $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ to denote the product projections. The pairing of two arrows $f : C \to A$ and $g : C \to B$ is denoted by $\langle f, g\rangle : C \to A \times B$. Product associativity is denoted by $\alpha_{A,B,C} : A \times (B \times C) \to (A \times B) \times C$. The coproduct inclusions are written $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$. For $f : A \to C$ and $g : B \to C$, case analysis is the unique morphism $[f, g] : A + B \to C$ with $[f, g] \circ \mathsf{inl} = f$ and $[f, g] \circ \mathsf{inr} = g$. Product and coproduct can be made into bifunctors $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$ by defining their action on arrows (see e.g. [5]). It is also straightforward to obtain their generalizations to $n$ components.

A recursively defined datatype $T$ is understood as a solution (a fixed point) of an equation $X \cong FX$, for an appropriate endofunctor $F : \mathcal{C} \to \mathcal{C}$ that captures the shape (or signature) of the type. In this paper we will consider datatypes with signatures given by so-called *polynomial functors* [1]. The following is an inductive definition of this class of functors:

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

$I : \mathcal{C} \to \mathcal{C}$ stands for the identity functor. $\underline{A}^n : \mathcal{C}^n \to \mathcal{C}$ denotes the $n$-ary constant functor. It maps $n$-tuples of objects to the object $A$, and $n$-tuples of functions to the identity on $A$; when $n = 1$ we simply write $\underline{A}$. $\Pi_i^n : \mathcal{C}^n \to \mathcal{C}$ (with $n \geq 2$) denotes the $i$-th projection functor from a $n$-ary product category. $F\langle G_1, \ldots, G_n\rangle$ (or $F\langle G_i\rangle$ for short) denotes the composition of $F : \mathcal{C}^n \to \mathcal{C}$ with the functors $G_1, \ldots, G_n$ (all of the same arity); when $n = 1$ we omit 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 $\dagger \in \{\times, +\}$.

We shall often suppress the subscripts on natural transformations where the objects involved are clear.

## 2.2 Inductive Types

*Inductive types* are least fixpoints of (covariant) functors. They correspond to initial functor-algebras, a generalization of the usual notion of term algebras over a given signature.

Let $F : \mathcal{C} \to \mathcal{C}$ be a functor. An $F$-*algebra* is an arrow $h : FA \to A$, called the operation. The object $A$ is called the carrier of the algebra. A morphism of algebras, or $F$-*homomorphism*, between $h : FA \to A$ and $k : FB \to B$ is an arrow $f : A \to B$ such that $f \circ h = k \circ Ff$. The category of $F$-algebras is formed by considering $F$-algebras as objects and $F$-homomorphisms as morphisms. The initial object of this category, if it exists, gives the inductive type whose signature is captured by $F$, and encodes the constructors of that type. We shall denote the initial algebra by $in_F : F\,\mu F \to \mu F$. A well-known result states that any initial algebra is an isomorphism.

Initiality permits to associate an operator with each inductive type, which is used to represent functions defined by structural recursion on that type. This operator, usually called *fold* [3] (or *catamorphism* [24]), corresponds to the unique homomorphism that exists to any $F$-algebra $h : FA \to A$ from the initial one. We will denote it by $(\!|h|\!)_F : \mu F \to A$. For being fold an homomorphism the following equation holds:

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

**Example 2.1** Consider a datatype for natural numbers,

$$\mathbb{N} = \mathsf{zero} \mid \mathsf{succ}\ \mathbb{N}$$

Its signature is captured by a functor $K : \mathcal{C} \to \mathcal{C}$ such that $KA = 1 + A$ and $Kf = \mathsf{id}_1 + f$. Every $K$-algebra is a case analysis $[h_1, h_2] : 1 + A \to A$, with $h_1 : 1 \to A$ and $h_2 : A \to A$; in particular, the initial algebra $[\mathsf{zero}, \mathsf{succ}] : 1 + \mathbb{N} \to \mathbb{N}$ where $\mathsf{zero} : 1 \to \mathbb{N}$ and $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. ($\mathbb{N}$ stands for $\mu K$.) For each algebra $h = [h_1, h_2]$, fold is the unique arrow $f = (\!|h|\!)_K : \mathbb{N} \to A$ such that $f \circ \mathsf{zero} = h_1$ and $f \circ \mathsf{succ} = h_2 \circ f$. □

Lists, trees as well as many other datatypes are usually parameterised. The signature of such datatypes is captured by a bifunctor $F : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. By fixing the first argument of a bifunctor $F$ 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 parameterised inductive datatype $D_\mu A = \mu F_A$, least solution to the equation $X \cong F(A, X)$, whose constructors are given as part of the initial algebra $in_{F_A} : F_A(D_\mu A) \to D_\mu A$.

**Example 2.2** Lists with elements over $A$ are usually declared 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 signature of lists is captured by the functor $L_A = \underline{1} + \underline{A} \times I$. The initial algebra is $[\mathsf{nil}, \mathsf{cons}] : 1 + A \times A^* \to A^*$ with $\mathsf{nil} : 1 \to A^*$ and $\mathsf{cons} : A \times A^* \to A^*$. For any algebra $h = [h_1, h_2] : 1 + A \times B \to B$, fold is the unique arrow $f = (\!|h|\!)_{L_A} : A^* \to B$ such that, $f(\mathsf{nil}) = h_1$ and $f(\mathsf{cons}(a, \ell)) = h_2(a, f(\ell))$. It corresponds to the standard $\mathsf{foldr}$ operator used in functional programming [3]. □

**Example 2.3** Consider a datatype for binary trees

$$\mathsf{btree}(A) = \mathsf{empty} \mid \mathsf{node}\ (\mathsf{btree}(A) \times A \times \mathsf{btree}(A))$$

Its signature is captured by the parameterised functor $B_A = \underline{1} + I \times \underline{A} \times I$. The constructors form the initial algebra $[\mathsf{empty}, \mathsf{node}] : 1 + \mathsf{btree}(A) \times A \times \mathsf{btree}(A) \to \mathsf{btree}(A)$. For each algebra $h = [h_1, h_2] : B_A C \to C$, the fold operator is given by the unique arrow $f = (\!|h|\!)_{B_A} : \mathsf{btree}(A) \to C$ that satisfies these equations: $f(\mathsf{empty}) = h_1$ and $f(\mathsf{node}(t, a, u)) = h_2(f(t), a, f(u))$. □

Now we discuss some of the standard calculational properties that fold enjoys. The first law we present is rather obvious. It is called the *identity law* and states that a fold with the initial algebra as target is the identity.

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

A law that plays an important role in program calculation is the so-called *fusion law*. It states that the composition of a fold with an algebra homomorphism is again a fold.

$$f \circ h = g \circ Ff \;\Rightarrow\; f \circ (\!|h|\!)_F = (\!|g|\!)_F \tag{1}$$

The next law is known as *acid rain* or *fold-fold fusion*. The goal of acid rain is to combine functions that produce and consume elements of an intermediate data structure. In order to be removed by this law, the intermediate datatype is required to be produced by a fold whose target algebra is constructed in terms of a transformer. A *transformer* [11] is a function $\boldsymbol{T} : (FA \to A) \to (GA \to A)$ that converts $F$-algebras into $G$-algebras such that, if $f : A \to B$ is a homomorphism between $F$-algebras $h : FA \to A$ and $h' : FB \to B$, i.e. $f \circ h = h' \circ Ff$, then it is also a homomorphism between the corresponding $G$-algebras, i.e. $f \circ \boldsymbol{T}(h) = \boldsymbol{T}(h') \circ Gf$. Intuitively, a transformer $\boldsymbol{T}$ may be thought of as a polymorphic function that uses algebras of one class to construct algebras of another class.

Fold-fold fusion then states the following:

$$\boldsymbol{T} \text{ transformer} \quad \Rightarrow \quad (\!|h|\!)_F \circ (\!|\boldsymbol{T}(in_F)|\!)_G = (\!|\boldsymbol{T}(h)|\!)_G \tag{2}$$

This law can be proved as follows. Consider an algebra $h : FA \to A$. Since every fold is a homomorphism, by definition of transformer we have 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 $\boldsymbol{T}(h) : GA \to A$. Therefore, by fusion (1), it follows that $(\!|h|\!)_F \circ (\!|\boldsymbol{T}(in_F)|\!)_G = (\!|\boldsymbol{T}(h)|\!)_G$, which is the desired result.

Let $D_\mu A = \mu F_A$ be a parameterised inductive datatype induced by a bifunctor $F$. $D_\mu$ is thus a type constructor that can be made into a functor $D_\mu : \mathcal{C} \to \mathcal{C}$, called a *type functor* [5], by defining its action $D_\mu f : D_\mu A \to D_\mu B$ on each arrow $f : A \to B$:

$$D_\mu f = (\!|in_{F_B} \circ F(f, \mathsf{id}_{D_\mu B})|\!)_{F_A}$$

For instance, by expanding the definition of $\mathsf{list}(f) = (\!|[\mathsf{nil}, \mathsf{cons} \circ (f \times \mathsf{id})]|\!)_{L_A}$ we get the definition of the usual $\mathsf{map}$ function on lists [3].

A standard property of type functors is *map-fold fusion*. For $f : A \to B$ and $h : F_B\, C \to C$,

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

### 2.3 Coinductive Types

*Coinductive types* are greatest fixpoints of functors. They represent (potentially) infinite datatypes. Given a functor $F$, a *$F$-coalgebra* is an arrow $g : A \to FA$. The object $A$ is called the carrier of the coalgebra. A coalgebra map, or *$F$-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. The final object of this category, if it exists, gives the coinductive type with signature $F$. We denote the final coalgebra by $out_F : \nu F \to F\nu F$; it encodes the destructors of the coinductive type.

Finality means the existence of a unique homomorphism from any coalgebra $g : A \to FA$ to $out_F$, which gives rise to an operator, called *unfold* [19, 16] (or *anamorphism* [24]), that will be denoted by $[\!(g)\!]_F : A \to \mu F$. It satisfies the equation:

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

**Example 2.4** The functor $S_A = \underline{A} \times I$ captures the signature of the datatype of streams $A^\infty$, formed by infinite 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, $f = \mathsf{scons} \circ \langle h, f \circ t \rangle$. □

Calculational laws for unfold can be derived by finality (see e.g. [11]). Like for inductive types, each parameterised coinductive type $D_\nu A = \nu F_A$, with $F$ a bifunctor, can be made into a *type functor* $D_\nu : \mathcal{C} \to \mathcal{C}$ by defining its action on each arrow $f : A \to B$,

$$D_\nu f = [\!(F(f, \mathsf{id}_{D_\nu A}) \circ out_{F_A})\!]_{F_B}$$

For instance, $f^\infty = [\!(\langle f \circ \mathsf{head}, \mathsf{tail} \rangle)\!]_{S_A}$. A standard property of type functors is *unfold-map fusion*. For $f : A \to B$ and $g : C \to F_A\, C$,

$$D_\nu f \circ [\!(g)\!]_{F_A} = [\!(F(f, \mathsf{id}_C) \circ g)\!]_{F_B} \tag{3}$$

## 3 Comonads

In this section we define the notion of a comonad and related concepts. We also analyze those aspects of the interaction between functors and comonads that turn out to be essential for merging recursion and comonads.

Monads are a well-known mechanism to structure functional programs (or program segments) that produce side effects. A typical monadic program is of type $A \to MB$, where $M$ (the monad) is an abstract data type that encapsulates the action of the specific effect. Every monad provides, essentially, the means to compose effect-producing programs as well as to inject values into computations. In this section, we see that the dual structures to monads, called *comonads*, can also be used as an abstraction to model some kinds of computation. Comonads account naturally for effects that derive from the context in which a program is executed. A typical comonadic program is of type $NA \to B$, where $N$ is an abstract data type that encapsulates the effect modeled by the comonad. Every comonad provides, essentially, the means to compose comonadic functions as well as to project a value from any computation in the comonad.

The following is one of the formal definitions of a comonad.

**Definition 3.1** A *comonad* over a category $\mathcal{C}$ is a triple $(N, \epsilon, -^{\#})$, called a *Kleisli triple*, where $N$ is the action on objects of a functor $N : \mathcal{C} \to \mathcal{C}$, $\epsilon : N \Rightarrow I$ is a natural transformation, and $-^{\#}$ is an extension operator which for each arrow $f : NA \to B$ yields an arrow $f^{\#} : NA \to NB$, and such that the following equations hold: $\epsilon_A^{\#} = \mathsf{id}_{NA}$, for every $f : NA \to B$, $\epsilon_B \circ f^{\#} = f$, and for every $f : NA \to B$ and $g : NB \to C$, $g^{\#} \circ f^{\#} = (g \circ f^{\#})^{\#}$. □

If we understand $NA$ as a type of computations over $A$, then $\epsilon_A : NA \to A$ is the means to project a value from a computation. The extension operator $-^{\#}$ plays the same role as the extension operator for monads (the popular bind operator), in the sense that it provides a way of composing comonadic functions. Indeed, the Kleisli composition of $f : NA \to B$ and $g : NB \to C$ is defined by $g \bullet f = g \circ f^{\#}$. The comonad laws thus state that Kleisli composition is associative and that $\epsilon$ is a left and right identity with respect to it.

Like for monads, we can associate a Kleisli categoy to each comonad.

**Definition 3.2** For each Kleisli triple $(N, \epsilon, -^{\#})$ over $\mathcal{C}$, the *Kleisli category* $\mathcal{C}_N$ is defined as follows: the objects of $\mathcal{C}_N$ are those of $\mathcal{C}$; morphisms between objects $A$ and $B$ in $\mathcal{C}_N$ correspond to arrows $NA \to B$ in $\mathcal{C}$, i.e. $\mathcal{C}_N(A, B) \equiv \mathcal{C}(NA, B)$; identities are given by $\epsilon_A : NA \to A$; and composition is given by Kleisli composition. □

We can define a *lifting functor* $(\widetilde{-}) : \mathcal{C} \to \mathcal{C}_N$ as the identity on objects, and $\widetilde{f} = f \circ \epsilon_A : NA \to B$, for each $f : A \to B$.

**Example 3.3** The *product comonad* models the presence of contextual information that is passed around. Let $X$ be an object of $\mathcal{C}$ representing a type of contexts. Then,

$$NA = A \times X \qquad\qquad \epsilon_A = \pi_1 \qquad\qquad f^{\#} = \langle f, \pi_2 \rangle$$

for $f : NA \to B$. That is, $\epsilon$ projects the value contained in a computation discarding the context. The extension operator, on the other hand, applies function $f$ to the input computation and copies the context to the output. □

**Example 3.4** Computations in the *state in context comonad* deal with a state originated in the context and a function to make observations on the state. Let $S$ stand for a state space. Then,

$$NA = [S \to A] \times S \qquad\qquad \epsilon_A = \lambda(f, s).\ f(s) \qquad\qquad f^{\#} = \mathsf{curry}(f) \times \mathsf{id}_S$$

where recall that, for $f : [S \to A] \times S \to B$, $\mathsf{curry}(f) : [S \to A] \to [S \to B]$. The $\epsilon$ operator permits to project a value from the state. For every $f : NA \to B$, the extension operator takes a computation in the comonad and returns another composed by a new function to project a value from the state and the same state as in the input computation. □

The following is an alternative definition of a comonad.

**Definition 3.5** A *comonad* over $\mathcal{C}$ is a triple $(N, \epsilon, \gamma)$ formed by an endofunctor $N : \mathcal{C} \to \mathcal{C}$ and two natural transformations $\epsilon : N \Rightarrow I$ and $\gamma : N \Rightarrow NN$ which obey the laws: $\epsilon_{NA} \circ \gamma_A = \mathsf{id}_{NA} = N\epsilon_A \circ \gamma_A$ and $\gamma_{NA} \circ \gamma_A = N\gamma_A \circ \gamma_A$. □

The following equations hold as part of the relationship between both definitions of a comonad: $Nf = (f \circ \epsilon_A)^{\#}$, for $f : A \to B$, $\gamma_A = \mathsf{id}^{\#}_{NA}$, and $f^{\#} = Nf \circ \gamma_A$, for $f : NA \to B$.

**Example 3.6** The *stream comonad* [15] describes computations that produce an infinite sequence of results.

$$NA = A^{\infty} \qquad\qquad \epsilon_A = \mathsf{head} \qquad\qquad \gamma_A = \mathsf{tails}$$

where $\mathsf{tails} : A^{\infty} \to (A^{\infty})^{\infty}$ is the function that generates the sequence with all tails of a given stream. It is given by an unfold $\mathsf{tails} = [\![\langle \mathsf{id}, \mathsf{tail} \rangle]\!]_{S_A}$. Since

$$f^{\#} = Nf \circ \gamma_A = f^{\infty} \circ \mathsf{tails}$$

for each $f : A^{\infty} \to B$, and $\mathsf{tails}$ is an unfold, by applying unfold-map fusion, law (3), we get $f^{\#} = [\![\langle f, \mathsf{tail} \rangle]\!]_{S_A} : A^{\infty} \to B^{\infty}$. That is, $f^{\#}(s) = \mathsf{scons}(f(s), f^{\#}(\mathsf{tail}(s)))$. □

Other examples of comonads can be found in [6, 21, 31].

In this paper we are interested in studying recursive operators that involve comonadic computations. Combining recursion and comonads requires an analysis of the interaction between comonads and functors representing datatype signatures. For this analysis we will follow the guidelines given in previous works on monads (see e.g. [12, 28]).

The fundamental structure that needs to be considered for the interaction is a *distributive law* of a comonad $N$ over a functor $F$, that is, a natural transformation

$$\delta^F : NF \Rightarrow FN$$

From it we can derive the *comonadic extension* of functor $F$ over the comonad $N$, $\widetilde{F} : \mathcal{C}_N \to \mathcal{C}_N$, which is a construction that acts on elements of the Kleisli category. Indeed, given a distributive law $\delta^F$, the action of the corresponding extension $\widetilde{F}$ on each arrow $f : NA \to B$ is given by

$$\widetilde{F}f \;=\; NFA \xrightarrow{\delta^F_A} FNA \xrightarrow{Ff} FB$$

The action on objects is given by $\widetilde{F}A = FA$ for every extension, because the objects of $\mathcal{C}_N$ and $\mathcal{C}$ coincide. Comonadic extensions and distributive laws are actually in one-to-one correspondence.

Under certain conditions, the comonadic extension may be a functor on the Kleisli category itself. In that case $\widetilde{F}$ is said to be a *lifting*. The conditions are given in the following theorem.

**Theorem 3.7 ([27])** *Given a comonad $(N, \epsilon, \gamma)$ and a functor $F$ on $\mathcal{C}$, $\widetilde{F} : \mathcal{C}_N \to \mathcal{C}_N$ is a lifting of $F$ iff the corresponding distributive law $\delta^F : NF \Rightarrow FN$ satisfies these equations:*

$$F\epsilon_A \circ \delta^F_A \;=\; \epsilon_{FA} \tag{4}$$

$$\delta^F_{NA} \circ N\delta^F_A \circ \gamma_{FA} \;=\; F\gamma_A \circ \delta^F_A \tag{5}$$

Recall that we are considering signatures that are given by polynomial functors. Given an arbitrary functor $F$ and a comonad $(N, \epsilon, -^{\#})$, a distributive law $\delta^F$ is then defined by induction on the structure of $F$. Some cases of that definition are unproblematic, since they directly follow by type considerations:

$$\begin{array}{lll}
\delta^I_A & = \mathsf{id}_{NA} & : NA \to NA \\
\delta^{\underline{C}^n}_{(A_1,\ldots,A_n)} & = \epsilon_C & : NC \to C \\
\delta^{\Pi^n_i}_{(A_1,\ldots,A_n)} & = \mathsf{id}_{NA_i} & : NA_i \to NA_i \\
\delta^{F\langle G_i \rangle}_A & = F(\delta^{G_1}_A, \ldots, \delta^{G_n}_A) \circ \delta^F_{(G_i A)} & : NF(G_i A) \to F(G_i NA)
\end{array}$$

In the last line we used $(G_i Y)$, for some $Y$, as an abbreviation for $(G_1 Y, \ldots, G_n Y)$.

The following are typical cases:

$$\delta_A^{F\times G} = (\delta_A^F \times \delta_A^G) \circ \delta^{\times}_{(FA,GA)} \qquad\qquad \delta_A^{F+G} = (\delta_A^F + \delta_A^G) \circ \delta^{+}_{(FA,GA)}$$

The distributive laws for the product and the coproduct require some additional considerations. In the case of the product,

$$\delta^{\times}_{(A,B)} : N(A \times B) \to NA \times NB$$

is an arrow, in general not uniquely determined, that splits a computation. The following is a standard choice:

$$\delta^{\times}_{(A,B)} = \langle N\pi_1, N\pi_2 \rangle$$

which satisfies the equations of Theorem 3.7:

$$\begin{aligned} (\epsilon_A \times \epsilon_B) \circ \delta^{\times}_{(A,B)} &= \epsilon_{A\times B} \\ \delta^{\times}_{NA,NB} \circ N\delta^{\times}_{(A,B)} \circ \gamma_{A\times B} &= (\gamma_A \times \gamma_B) \circ \delta^{\times}_{(A,B)} \end{aligned}$$

Therefore, with this choice of $\delta^{\times}$, $\widetilde{\times} : \mathcal{C}_N \times \mathcal{C}_N \to \mathcal{C}_N$ results to be a lifting of $\times$.

A distributive law for the coproduct is an arrow

$$\delta^{+}_{(A,B)} : N(A + B) \to NA + NB$$

that does not always exist. One alternative would be to proceed by analogy with monads, and require, for the existence of the coproduct distribution, that the comonad is costrong. A comonad is said to be *costrong* when it comes equipped with a natural transformation $\rho_{A,B} = N(A+B) \to NA + B$, called a *costrength*, subject to four coherence conditions obtained from those for strong monads [26] by reversing the direction of all arrows and replacing all products by coproducts. We will not enter into the details here. For the product comonad, the comonad we will deal with in the next two sections, a distributive law for the coproduct exists under certain conditions, and, as we will see later, that distributive law turns out to be the natural choice.

From the distributive laws defined above we can derive an expression of the monadic extension for each polynomial functor:

$$\begin{array}{lcl@{\qquad\qquad}lcl} \widetilde{I} f & = & f & \widetilde{F\langle G_i\rangle} f & = & \widetilde{F}(\widetilde{G_1} f, \ldots, \widetilde{G_n} f) \\ \widetilde{\underline{C}^n} f & = & \epsilon_C & f \mathbin{\widetilde{\times}} g & = & \langle f \circ N\pi_1, g \circ N\pi_2 \rangle \\ \widetilde{\Pi_i^n}(f_1, \ldots, f_n) & = & f_i & f \mathbin{\widetilde{+}} g & = & (f + g) \circ \delta^+ \end{array}$$

Thus, in particular,

$$(\widetilde{F \times G}) f = \langle \widetilde{F} f \circ N\pi_1, \widetilde{G} f \circ N\pi_2 \rangle \qquad\qquad (\widetilde{F + G}) f = (\widetilde{F} f + \widetilde{G} f) \circ \delta^+$$

We conclude this section presenting a property specific to comonadic extensions of composite functors. In the next section, this property will help us to derive expansions of a comonadic recursive operator on specific datatypes.

**Proposition 3.8** Let $(N, \epsilon, -^{\#})$ be a comonad. Let $H = F\langle G_1, \ldots, G_n \rangle$ be a composite functor on $\mathcal{C}$ such that $F : \mathcal{C}^n \to \mathcal{C}$ has a lifting $\widetilde{F}$ over $N$. Then, for every $f : NA \to B$, the following diagram commutes:

$$\begin{tikzcd} NF(G_i A) \arrow[r, "(\widetilde{H} f)^{\#}"] \arrow[d, "\delta^F_{(G_i A)}"'] & NF(G_i B) \arrow[d, "\delta^F_{(G_i B)}"] \\ F(NG_i A) \arrow[r, "F((\widetilde{G_i} f)^{\#})"'] & F(NG_i B) \end{tikzcd}$$

**Proof** The commutativity of the desired diagram follows from the commutativity of the following composite diagram.

$$\begin{tikzcd}
NF(G_iA) \arrow[r, "(\delta^F)^\#"] \arrow[d, "\delta^F"'] \arrow[dr, phantom, "(I)"] & NF(NG_iA) \arrow[r, "NF(\delta^{G_i})"] \arrow[d, "\delta^F"'] \arrow[dr, phantom, "(II)"] & NF(G_iNA) \arrow[r, "NF(G_if)"] \arrow[d, "\delta^F"] \arrow[dr, phantom, "(III)"] & NF(G_iB) \arrow[d, "\delta^F"] \\
F(NG_iA) \arrow[r, "F(\mathsf{id}^\#)"'] & F(NNG_iA) \arrow[r, "F(N\delta^{G_i})"'] & F(NG_iNA) \arrow[r, "F(NG_if)"'] & F(NG_iB)
\end{tikzcd}$$

In fact, observe that

$$(\widetilde{H}f)^\# = NF(G_if) \circ NF(\delta^{G_i}) \circ (\delta^F)^\# \qquad F((\widetilde{G_i}f)^\#) = F(NG_if) \circ F(N\delta^{G_i}) \circ F(\mathsf{id}^\#)$$

Recall that, by hypothesis, $\widetilde{F}$ was assumed to be a lifting. So, in particular, $\delta^F$ satisfies the equations of Theorem 3.7. Therefore, $(I)$ commutes as it coincides with equation (5); note that $\mathsf{id}^\# = \gamma$ and $(\delta^F)^\# = N\delta^F \circ \gamma$. $(II)$ and $(III)$ commute by naturality of $\delta^F$. □

To see an instance of this property, consider the case of $H = F + G$. Then, the following equation holds:

$$\delta^+ \circ (\widetilde{H}f)^\# = ((\widetilde{F}f)^\# + (\widetilde{G}f)^\#) \circ \delta^+ \tag{6}$$

# 4 Functions with Parameters

Some recursive functions require extra (fixed) *parameters*, usually representing some context information, for their computation. A function of this kind can be defined in essentially two ways. One is to give it by a higher-order definition, i.e. as a curried function that yields a function on the parameters as result, something that is common practice in a higher-order functional language. From a categorical point of view, a definition of this kind can be given if the corresponding underlying category is *cartesian closed*, that is, a category such that for every pair of objects $A$ and $B$ there is an exponential object $[A \to B]$ satisfying an universal property (see e.g. [2]). To see an example, consider the function that adds two natural numbers. Its curried definition $\mathsf{add} : \mathbb{N} \to [\mathbb{N} \to \mathbb{N}]$ is given by

$$\mathsf{add}\ \mathsf{zero}\ n = n \qquad \mathsf{add}\ (\mathsf{succ}\ m)\ n = \mathsf{succ}\ (\mathsf{add}\ m\ n)$$

This definition corresponds to a higher-order fold, $\mathsf{add} = (\!| h_1, h_2 |\!)$, with $h_1 = \lambda n.n : \mathbb{N} \to \mathbb{N}$, and $h_2 = \lambda g.\lambda n.\, \mathsf{succ}\,(g\ n) : [\mathbb{N} \to \mathbb{N}] \to [\mathbb{N} \to \mathbb{N}]$.

The other possibility is to introduce the function with parameters as a first-order definition, i.e. as a function from the product between the recursive argument and the parameters to the result. The corresponding definition of $\mathsf{add}$, of type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$, is given by

$$\mathsf{add}\ (\mathsf{zero}, n) = n \qquad \mathsf{add}\ (\mathsf{succ}\ m, n) = \mathsf{succ}\ (\mathsf{add}\ (m, n))$$

Definitions of this kind can be written both in a first-order and in a higher-order language, but they cannot be represented as a fold. The problem is that fold does not possess the ability of explicitly managing parameters by itself. One way to overcome this problem is to introduce a new operator, called *pfold*, which is a sort of *fold with parameters*. Our motivation to study such an operator is by no means because we want to avoid the use of higher-order. Higher-order is without doubts one of the most important and useful features of modern functional languages. Our interest in pfold is, however, based on the fact that it represents an alternative way of defining a specific class of structural recursive functions, and, perhaps the most important reason, it constitutes a simple example of the combination between comonadic effects and recursion. On the other hand, pfold might be the only alternative available to define the functions in question in a traditional language without higher-order features.

As Cockett and Spencer [10] observed, to achieve a definition of pfold it is not necessary to assume that the underlying category $\mathcal{C}$ is cartesian closed. Instead, it is sufficient to assume that the initial algebra is *strongly initial* [10] (or *initial with parameters*). The theory of strong datatypes has been used as the basis for the design of the programming language CHARITY [9].

The concept of strong initiality is based on that of strong functor. A functor $F : \mathcal{C} \rightarrow \mathcal{C}$ is said to be *strong* if it is equipped with a natural transformation $\tau^F_{A,X} : FA \times X \rightarrow F(A \times X)$, called a *strength*, such that the following equations hold:

$$F\pi_1 \circ \tau^F_{A,X} = \pi_1 \tag{7}$$

$$F\alpha_{A,X,Y} \circ \tau^F_{A,X\times Y} = \tau^F_{A\times X,X} \circ (\tau^F_{A,X} \times \mathsf{id}_Y) \circ \alpha_{FA,X,Y} \tag{8}$$

Polynomial functors turn out to be strong under the additional assumption that category $\mathcal{C}$ is distributive. A category $\mathcal{C}$ is said to be *distributive* [34, 8] if it possesses both finite products and coproducts and binary products distribute over coproducts. This means that, for any objects $A$, $B$ and $C$, the canonical map

$$[\mathsf{inl} \times \mathsf{id}_C, \mathsf{inr} \times \mathsf{id}_C] : A \times C + B \times C \rightarrow (A + B) \times C$$

is an isomorphism whose inverse is the natural transformation denoted by

$$d_{A,B,C} : (A + B) \times C \rightarrow A \times C + B \times C$$

There is a plenty of examples of such categories, since every cartesian-closed category with coproducts is a distributive category. **Set** and **Cpo**[1] are typical cases.

A definition of strength for each polynomial functor $F$ can be given by induction on the structure of $F$.

$$\begin{array}{lll}
\tau^I_{A,X} & = \mathsf{id}_{A\times X} & : A \times X \rightarrow A \times X \\
\tau^{\underline{C}^n}_{(A_1,\ldots,A_n),X} & = \pi_1 & : C \times X \rightarrow C \\
\tau^{\Pi^n_i}_{(A_1,\ldots,A_n),X} & = \mathsf{id}_{A_i\times X} & : A_i \times X \rightarrow A_i \times X \\
\tau^{\times}_{(A,B),X} & = \langle \pi_1 \times \mathsf{id}_X, \pi_2 \times \mathsf{id}_X \rangle & : (A \times B) \times X \rightarrow (A \times X) \times (B \times X) \\
\tau^{+}_{(A,B),X} & = d_{A,B,X} & : (A + B) \times X \rightarrow A \times X + B \times X \\
\tau^{F\langle G_i\rangle}_{A,X} & = F(\tau^{G_1}_{A,X}, \ldots, \tau^{G_n}_{A,X}) \circ \tau^F_{(G_i A),X} & : F(G_i A) \times X \rightarrow F(G_i(A \times X))
\end{array}$$

It is easy to check that each $\tau$ above defined indeed satisfies the equations (7) and (8).

Strong functors can be lifted to work on $X$-actions, which are arrows of type $A \times X \rightarrow B$ for each $A$ and $B$. Given a strong functor $F$, for each $f : A \times X \rightarrow B$, we define $F^X f : FA \times X \rightarrow FB$ to be $F^X f = Ff \circ \tau^F_{A,X}$.

Given a strong functor $F$, an initial $F$-algebra $in_F$ is said to be *strongly initial* [10] if, for each object $X$ and $X$-action $h : FA \times X \rightarrow A$, there exists a unique $X$-action $f : \mu F \times X \rightarrow A$ that makes the following diagram in $\mathcal{C}$ commute

$$\begin{tikzcd}
F\mu F \times X \arrow[r, "in_F \times \mathsf{id}_X"] \arrow[d, "\langle F^X f, \pi_2\rangle"'] & \mu F \times X \arrow[d, "f"] \\
FA \times X \arrow[r, "h"'] & A
\end{tikzcd}$$

The unique arrow $f$ that results from strong initiality is precisely the definition of the pfold operator we are looking for. We denote it by $\mathsf{pfold}_F(h) : \mu F \times X \rightarrow A$.

The following proposition guarantees the existence of categories where strong initiality holds.

**Proposition 4.1** (**[10]**) If $\mathcal{C}$ is a cartesian closed category, then every initial algebra is strongly initial.

[1]By **Cpo** we mean the category of cpos (not necessarily having a bottom element) and continuous functions.

**Proof** Let $in_F$ be initial. Consider an $X$-action $h : FA \times X \to A$. With it, construct the $F$-algebra $k = \mathsf{curry}(j) : F[X \to A] \to [X \to A]$, where $j = h \circ \langle F^X \mathsf{apply}, \pi_2 \rangle : F[X \to A] \times X \to A$. Now, consider the following composite diagram:

$$\begin{tikzcd}
F\mu F \times \mathsf{id}_X \arrow[r, "Ff \times \mathsf{id}_X"] \arrow[d, "in_F \times \mathsf{id}_X"'] \arrow[dr, phantom, "(I)"] & F[X \to A] \times X \arrow[r, "\langle F^X \mathsf{apply}, \pi_2 \rangle"] \arrow[d, "k \times \mathsf{id}_X"] \arrow[dr, phantom, "(II)"] & FA \times X \arrow[d, "h"] \\
\mu F \times \mathsf{id}_X \arrow[r, "f \times \mathsf{id}_X"'] & {[X \to A] \times X} \arrow[r, "\mathsf{apply}"'] & A
\end{tikzcd}$$

where $f = (\!|k|\!)_F$. (I) commutes by initiality of $in_F$, whereas (II) commutes by the universal property of the exponential, i.e. $\mathsf{apply} \circ (\mathsf{curry}(j) \times \mathsf{id}_X) = j$. So, as a consequence, the outer rectangle commutes. By the bijection between the curried and uncurried version of any arrow, we have that, given $f : \mu F \to [X \to A]$, there is a unique $f' : \mu F \times X \to A$ such that $\mathsf{apply} \circ (f \times \mathsf{id}_X) = f'$. Therefore, since $\langle F^X \mathsf{apply}, \pi_2 \rangle \circ (Ff \times \mathsf{id}_X) = \langle F^X(\mathsf{apply} \circ (f \times \mathsf{id}_X)), \pi_2 \rangle$, it follows that $f'$ is the unique arrow such that

$$f' \circ (in_F \times \mathsf{id}_X) = h \circ \langle F^X f', \pi_2 \rangle$$

In other words, $in_F$ is strongly initial. Since $f'$ corresponds to $\mathsf{pfold}_F(h)$, as an aside we obtain this equation:

$$\mathsf{pfold}_F(h) = \mathsf{apply} \circ ((\!|\mathsf{curry}(h \circ \langle F^X \mathsf{apply}, \pi_2 \rangle)|\!)_F \times \mathsf{id}_X)$$

$\square$

Consider the product comonad $(N, \epsilon, -^{\#})$ described in Example 3.3. Observe that each $X$-action corresponds to an arrow $NA \to B$ in the Kleisli category $\mathcal{C}_N$. In particular, every pfold. Motivated by this fact we will restate the definition of pfold, now in terms of comonadic notions. As a result we will obtain a definition that makes explicit the fact that pfold is a special case of a *comonadic fold.*

First of all, observe that every strength $\tau^F_{A,X} : FA \times X \to F(A \times X)$ is a distributive law $\delta^F_A : NFA \to FNA$ of the product comonad over $F$. Therefore, $F^X$ corresponds to a comonadic extension $\widetilde{F}$, which can be shown to be indeed a lifting of $F$ when $F$ is polynomial. In addition, it holds that $h \circ \langle \widetilde{F} f, \pi_2 \rangle = h \circ (\widetilde{F} f)^{\#}$ and $f \circ (g \times \mathsf{id}_X) = f \circ \langle g \circ \pi_1, \pi_2 \rangle = f \circ (g \circ \pi_1)^{\#} = f \circ \widetilde{g}^{\#}$.

In summary, the universal property of pfold states that, for any $h : NFA \to A$, pfold is the unique arrow $f = \mathsf{pfold}_F(h) : N\mu F \to A$ that makes the following diagram commute:

$$\begin{tikzcd}
NF\mu F \arrow[r, "\widetilde{in_F}^{\#}"] \arrow[d, "(\widetilde{F} f)^{\#}"'] & N\mu F \arrow[d, "f"] \\
NFA \arrow[r, "h"'] & A
\end{tikzcd}$$

or equivalently,

$$\mathsf{pfold}_F(h) \bullet \widetilde{in_F} = h \bullet \widetilde{F}\, \mathsf{pfold}_F(h)$$

This means that pfold can be regarded as being the definition of a *comonadic fold* for the special case of the product comonad.

The following notions hold for every comonad. A *comonadic $F$-algebra* is an arrow $h : NFA \to A$. Viewed as an arrow in $\mathcal{C}_N$, a comonadic algebra corresponds to a $\widetilde{F}$-algebra in that category. Consider two $\widetilde{F}$-algebras $h : NFA \to A$ and $h' : NFB \to B$. A *homomorphism* between $h$ and $h'$ is an arrow $f : NA \to B$ such that $f \bullet h = h' \bullet \widetilde{F} f$, whereas a *pure homomorphism* between them is an arrow $f : A \to B$ such that $f \circ h = h' \circ NFf$.

Therefore, in the context of the product comonad, an initial algebra $in_F$ is said to be strongly initial when its lifting, $\widetilde{in_F} : NF\mu F \to \mu F$, happens to be the initial object in the category of $\widetilde{F}$-algebras. This means that, for each $\widetilde{F}$-algebra $h : NFA \to A$, the *comonadic fold* is defined as the unique homomorphism between $\widetilde{in_F}$ and $h$ that exists by initiality. Observe that, so defined, the comonadic fold (or equivalently, pfold) coincides with the standard fold of the category $\mathcal{C}_N$.

**Example 4.2** Like we saw in Example 2.1, the signature of natural numbers is given by the functor $K = \underline{1}+I$. Recall that $NA = A\times X$. Thus, $\delta_A^K = (\pi_1+\mathsf{id})\circ d : N(1+A) \to 1+NA$ and $\widetilde{K}f = (\pi_1+f)\circ d$, for $f : NA \to B$. For any algebra $h = [h_1, h_2]\circ d : N(1+A) \to A$, with $h_1 : 1\times X \to A$ and $h_2 : A\times X \to A$, pfold is the unique arrow $f = \mathsf{pfold}_K(h) : \mathbb{N} \times X \to A$ such that

$$f \circ ([\mathsf{zero}, \mathsf{succ}] \circ \mathsf{id}_X) = h \circ (\widetilde{K}f)^{\#}$$

Let us now derive an expansion of this definition. By equation (6), we have that $d\circ(\widetilde{K}f)^{\#} = (\pi_1^{\#}+f^{\#})\circ d$. So, $h \circ (\widetilde{K}f)^{\#} = [h_1 \circ \pi_1^{\#}, h_2 \circ f^{\#}] \circ d = [h_1, h_2 \circ \langle f, \pi_2\rangle] \circ d$. Pre-composing both sides of the equation with $d^{-1} = [\mathsf{inl} \times \mathsf{id}_X, \mathsf{inr} \times \mathsf{id}_X]$, we then obtain

$$f \circ [\mathsf{zero} \times \mathsf{id}_X, \mathsf{succ} \times \mathsf{id}_X] = [h_1, h_2 \circ \langle f, \pi_2\rangle]$$

Finally, by case analysis we get the desired equations:

$$f \circ (\mathsf{zero} \times \mathsf{id}_X) = h_1 \qquad\qquad f \circ (\mathsf{succ} \times \mathsf{id}_X) = h_2 \circ \langle f, \pi_2\rangle$$

□

**Example 4.3** Consider the list datatype. Its signature is given by the functor $L_A = \underline{1}+\underline{A}\times I$. For each $h = [h_1, h_2] \circ d : (1 + A \times B) \times X \to B$, pfold is the unique arrow $f = \mathsf{pfold}_{L_A}(h) : A^* \times X \to B$ that makes the following equation hold,

$$f \circ ([\mathsf{nil}, \mathsf{cons}] \times \mathsf{id}_X) = h \circ ((\widetilde{L_A}f)^{\#}$$

Like in the previous example, applying (6) and pre-composing both sides with $d^{-1}$, we obtain

$$f \circ [\mathsf{nil} \times \mathsf{id}_X, \mathsf{cons} \times \mathsf{id}_X] = [h_1, h_2 \circ \langle\langle \pi_1 \circ \pi_1, f \circ (\pi_2 \times \mathsf{id}_X)\rangle, \pi_2\rangle]$$

which is amenable to case analysis:

$$f \circ (\mathsf{nil} \times \mathsf{id}_X) = h_1 \qquad\qquad f \circ (\mathsf{cons} \times \mathsf{id}_X) = h_2 \circ \langle\langle \pi_1 \circ \pi_1, f \circ (\pi_2 \times \mathsf{id}_X)\rangle, \pi_2\rangle$$

In functional notation, $f\ (\mathsf{nil}, x) = h_1(x)$ and $f\ (\mathsf{cons}(a, \ell), x) = h_2(a, f(\ell, x), x)$. □

**Example 4.4** Consider the binary tree datatype. Its signature is given by the functor $B_A = \underline{1}+I\times\underline{A}\times I$. For each $h = [h_1, h_2] \circ d : (1 + B \times A \times B) \times X \to B$, pfold is the unique arrow $f = \mathsf{pfold}_{B_A}(h) : \mathsf{btree}(A) \times X \to B$ that makes the following equation hold,

$$f \circ ([\mathsf{empty}, \mathsf{node}] \times \mathsf{id}_X) = h \circ ((\widetilde{B_A}f)^{\#}$$

By similar arguments as in the preceding examples we obtain the following equations in functional notation:

$$f\ (\mathsf{empty}, x) = h_1(x) \qquad\qquad f\ (\mathsf{node}(t, a, u), x) = h_2(f(t, x), a, f(u, x), x)$$

□

The pfold operator can also be used to give a definition of a strength for any type functor $D_\mu$ corresponding to a strongly initial parameterised datatype induced by a bifunctor $F$ that satisfies to be bistrong. A bifunctor $F : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is called *bistrong* [9] if the functors $F(A, -)$ and $F(-, B)$ are strong. Then, $\tau^{D_\mu}_{A,X} : D_\mu A \times X \to D_\mu(A \times X)$ is given by

$$\tau^{D_\mu}_{A,X} = \mathsf{pfold}_{F_A}(\phi_{A,X}) \qquad \text{where} \qquad \phi_{A,X} = in_{F_{A\times X}} \circ \tau^{F(-,D_\mu(A\times X))}_{A,X}$$

A proof that $\tau^{D}_{A,X}$ is indeed a strength can be found in [29].

Now, let us see some laws for pfold. Assume that $F$ is polynomial; that way $\widetilde{F}$ is a lifting. We begin with an *identity law*, which states that a pfold with the lifting of the initial algebra as target is the identity in $\mathcal{C}_N$.

$$\mathsf{pfold}_F(\widetilde{in_F}) = \epsilon_{\mu F}$$

The *fusion law* states that the Kleisli composition of a pfold with a homomorphism is again a pfold.

$$f \bullet h = k \bullet \widetilde{F} f \;\Rightarrow\; f \bullet \mathsf{pfold}_F(h) = \mathsf{pfold}_F(k)$$

The following law states an obvious result: (the lifting of) every fold can be seen as a pfold that does not make use of the parameters.

$$\widetilde{(\!|h|\!)_F} = \mathsf{pfold}_F(\widetilde{h})$$

It is possible to state two *acid rain* laws for pfold. Each of them deal with a particular notion of transformer, different from the one introduced for fold in Section 2. The first law, called *pfold-pfold fusion*, permits to fuse two pfolds. The notion of transformer employed by pfold-pfold fusion is the following: A *transformer* is a function $\boldsymbol{T} : (NFA \to A) \to (NGA \to A)$ that converts $\widetilde{F}$-algebras into $\widetilde{G}$-algebras such that it preserves homomorphisms. That is, given two $\widetilde{F}$-algebras $h : NFA \to A$ and $h' : NFB \to B$, if $f : NA \to B$ is a homomorphism between them, i.e. $f \bullet h = h' \bullet \widetilde{F} f$, then it is also a homomorphism between the corresponding $\widetilde{G}$-algebras, i.e. $f \bullet \boldsymbol{T}(h) = \boldsymbol{T}(h') \bullet \widetilde{G} f$.

The definition of pfold-pfold fusion is the following:

$$\boldsymbol{T} \text{ transformer} \;\Rightarrow\; \mathsf{pfold}_F(h) \bullet \mathsf{pfold}_G(\boldsymbol{T}(\widetilde{in_F})) = \mathsf{pfold}_G(\boldsymbol{T}(h))$$

Note that, since pfold coincides with the standard fold when viewed as an arrow in $\mathcal{C}_N$, pfold-pfold fusion is nothing but fold-fold fusion (2) in $\mathcal{C}_N$.

The second acid rain law, called *pfold-fold fusion*, permits the fusion of compositions between pfolds and folds. It works with the following notion of transformer. A *transformer* is a function $\boldsymbol{T} : (FA \to A) \to (NGA \to A)$ that converts $F$-algebras into $\widetilde{G}$-algebras such that, if $f : A \to B$ is a homomorphism between $h : FA \to A$ and $h' : FB \to B$, then it is a pure homomorphism between the corresponding $\widetilde{G}$-algebras, that is, if $f \circ h = h' \circ Ff$ then $f \circ \boldsymbol{T}(h) = \boldsymbol{T}(h') \circ NGf$.

$$\boldsymbol{T} \text{ transformer} \;\Rightarrow\; (\!|h|\!)_F \circ \mathsf{pfold}_G(\boldsymbol{T}(in_F)) = \mathsf{pfold}_G(\boldsymbol{T}(h)) \tag{9}$$

Further laws for pfold can be found in [29].

**Example 4.5** The function $\mathsf{prune} : \mathsf{btree}(A) \times A^* \to \mathsf{btree}(A)$ takes a binary tree $t$ and a list $\ell$ and discards all subtrees whose roots occur in $\ell$.

$$\begin{aligned}
\mathsf{prune}(\mathsf{empty}, \ell) &= \mathsf{empty} \\
\mathsf{prune}(\mathsf{node}(t, a, t'), \ell) &= \mathbf{if}\ a \in \ell\ \mathbf{then}\ \mathsf{empty} \\
&\quad\ \mathbf{else}\ \mathsf{node}(\mathsf{prune}(t, \ell), a, \mathsf{prune}(t', \ell))
\end{aligned}$$

We can define it as pfold, $\mathsf{prune} = \mathsf{pfold}_{B_A}(\mathsf{alg\text{-}pr}) : N\mathsf{btree}(A) \to \mathsf{btree}(A)$, where $NA = A \times A^*$. The comonadic algebra $\mathsf{alg\text{-}pr} : NB_A(\mathsf{btree}(A)) \to \mathsf{btree}(A)$ is given by,

$$\begin{aligned}
\mathsf{alg\text{-}pr}(x, \ell) = \ & \mathbf{case}\ x\ \mathbf{of} \\
& \quad \mathsf{inl}(u) \to \mathsf{empty} \\
& \quad \mathsf{inr}(t, a, t') \to \mathbf{if}\ a \in \ell\ \mathbf{then}\ \mathsf{empty}\ \mathbf{else}\ \mathsf{node}(t, a, t')
\end{aligned}$$

It can be written as $\mathsf{alg\text{-}pr} = \boldsymbol{T}([\mathsf{empty}, \mathsf{node}])$, where $\boldsymbol{T} : (B_A C \to C) \to (NB_A C \to C)$ is the transformer given by,

$$\begin{aligned}
\boldsymbol{T}(h) = \lambda(x, \ell).\ & \mathbf{case}\ x\ \mathbf{of} \\
& \quad \mathsf{inl}(u) \to h_1 \\
& \quad \mathsf{inr}(c_1, a, c_2) \to \mathbf{if}\ a \in \ell\ \mathbf{then}\ h_1\ \mathbf{else}\ h_2(c_1, a, c_2)
\end{aligned}$$

with $h = [h_1, h_2]$.

Consider the function $\mathsf{size} : \mathsf{btree}(A) \to \mathbb{N}$ that counts the number of nodes of a tree.

$$\mathsf{size}(\mathsf{empty}) = \mathsf{zero} \qquad\qquad \mathsf{size}(\mathsf{node}(t, a, u)) = 1 + \mathsf{size}(t) + \mathsf{size}(t')$$

This function is a fold, $\mathsf{size} = (\!|\mathsf{alg\text{-}s}|\!)_{B_A}$; the algebra $\mathsf{alg\text{-}s} : 1 + \mathbb{N} \times A \times \mathbb{N} \to \mathbb{N}$ has the obvious definition. Suppose that now we want to count the number of nodes that remain in a tree after pruning.

$$\mathsf{count} \; = \; \mathsf{btree}(A) \times A^* \xrightarrow{\mathsf{prune}} \mathsf{btree}(A) \xrightarrow{\mathsf{size}} \mathbb{N}$$

By using pfold-fold fusion we can transform this composition into a single pfold, avoiding in that way the generation of the intermediate tree.

$$\mathsf{count} = \mathsf{size} \circ \mathsf{prune} = \mathsf{pfold}_{B_A}(\,\boldsymbol{T}(\mathsf{alg\text{-}s}))$$

That is,

$$\begin{aligned} \mathsf{count}(\mathsf{empty}, \ell) \; &= \; 0 \\ \mathsf{count}(\mathsf{node}(t, a, t'), \ell) \; &= \; \textbf{if } a \in \ell \textbf{ then } 0 \textbf{ else } 1 + \mathsf{count}(t, \ell) + \mathsf{count}(t', \ell) \end{aligned}$$

□

# 5 Accumulations

Accumulations are functions that use an extra parameter to keep intermediate results to be used during the computation (see e.g. [4, 13, 14, 18]). In this section we build up a comonadic operator for a kind of downwards accumulations by adding some ingredients to the definition of pfold.

For defining accumulations we can follow, essentially, the same two alternatives discussed before for functions with parameters. One is to define accumulations by higher-order folds. This is the approach adopted in [18] and [5]. As before, this alternative requires to work in a cartesian closed category. As an example, consider the function $\mathsf{asums}$ that computes the list of accumulated sums of a list of natural numbers. The curried version is of type $\mathsf{list}(\mathbb{N}) \to [\mathbb{N} \to \mathsf{list}(\mathbb{N})]$:

$$\mathsf{asums}\; [\,]\; e = [e] \qquad\qquad \mathsf{asums}\; (n : \ell)\; e = e : \mathsf{asums}\; \ell\; (e + n)$$

The other alternative is to give an uncurried definition of the accumulation. In the case of $\mathsf{asums}$,

$$\mathsf{asums}([\,], e) = [e] \qquad\qquad \mathsf{asums}(n : \ell, e) = e : \mathsf{asums}(\ell, e + n)$$

Definitions of this kind cannot be written as a fold nor as a pfold. Like for functions with parameters, a simple fold cannot be used because it lacks the possibility of managing extra arguments. The problem with pfold, on the other hand, is that it can deal with extra arguments, but they cannot be altered along the computation. Like in the previous section, the solution we will adopt consists of the introduction of a new operator, called *afold*, that corresponds to a *fold with accumulating parameters*. The motivations for defining such an operator are the same as the discussed for pfold. To achieve a definition of afold we will need to work with a modified version of strong initiality that reflects the presence of accumulations. We will also show that, like pfold, afold is a form of *comonadic fold*.

Let us fix an object $X$ that in this case will be regarded as an object of accumulators. Consider again the product comonad $(N, \epsilon, -^{\#})$, with $NA = A \times X$. Recall the diagram that defines pfold.

$$\begin{tikzcd} NF\mu F \arrow[r, "\widetilde{in_F}^{\#}"] \arrow[d, "(\widetilde{F}f)^{\#}"'] & N\mu F \arrow[d, "f"] \\ NFA \arrow[r, "h"'] & A \end{tikzcd}$$

As we saw in the previous section, the existence and uniqueness of a $f$ fulfilling this diagram is what characterizes the notion of strong initiality. Functions with parameters and accumulations are very similar structurally. The only difference between them is that accumulations modify the extra parameters during computation. Therefore, to achieve a definition of a fold with accumulating parameters starting off from that of pfold, we only need to alter the part of pfold that in the new operator represents the process of accumulation. That part is within $\widetilde{F}f$; more precisely, within the distributive law $\delta^F$. In pfold, the distributive law simply makes available the value of the parameters to the recursive calls. In an accumulation, however, we do not distribute the value of the parameters to the recursive calls, but an *accumulated value*, which is calculated from the current value of the parameters and the information contained in the node of the data structure that is being visited. In this respect, for a datatype with signature $F$, we will assume that the accumulation is performed by an *accumulating function* $g : F1 \times X \to X$, where $F1$ represents the information that is left in a node after discarding the sub-structures. We then introduce a modified version of the distributive law that reflects the process of accumulation:

$$\overline{\delta}_A^{F,g} \;=\; NFA \xrightarrow{\theta_A^{F,g}} NFA \xrightarrow{\delta_A^F} FNA$$

where

$$\theta_A^{F,g} \;=\; FA \times X \xrightarrow{\langle \pi_1, F(!_A) \times \mathsf{id}_X \rangle} FA \times (F1 \times X) \xrightarrow{\mathsf{id}_{FA} \times g} FA \times X$$

Observe that the new distributive law is indexed by the functor and the function that performs the accumulation. Indeed, the whole construction corresponding to the new operator will be parametric on $g$. Note also that $\overline{\delta}_A^{F,g}$ is natural on $A$, as required (in order to be a distributive law). This is because $g$ as well as the rest of the components of $\theta^{F,g}$ (which are in turn natural transformations) do not make any assumption about $A$. Moreover, $g$ acts on $X$, which is an internal datum of the comonad.

From $\overline{\delta}_A^{F,g}$ we can construct a new comonadic extension of $F$, let us call it $\overline{F}$. Then, $\overline{F}f = Ff \circ \overline{\delta}_A^{F,g}$, or which is the same, $\overline{F}f = \widetilde{F}f \circ \theta_A^{F,g}$. In general, we do not expect that the new extension is a lifting. However, equation (4) holds independently from the choice of $g$. This means that $\overline{F}$ preserves identities but lacks in general the preservation of Kleisli composition.

With the new constructions we can now state a modified version of strong initiality. Given a functor $F$ and an arrow $g : F1 \times X \to X$, we say that an initial algebra is *initial with accumulating parameters* if, for each $\overline{F}$-algebra $h : NFA \to A$, there exists a unique arrow $f = \mathsf{afold}_F(g, h) : N\mu F \to A$, called *fold with accumulating parameters*, that makes the following diagram in $\mathcal{C}$ commute

$$\begin{tikzcd}
NF\mu F \arrow[r, "\widetilde{in_F}^{\#}"] \arrow[d, "(\overline{F}f)^{\#}"'] & N\mu F \arrow[d, "f"] \\
NFA \arrow[r, "h"'] & A
\end{tikzcd}$$

or equivalently,

$$\mathsf{afold}_F(g, h) \bullet \widetilde{in_F} = h \bullet \overline{F}\, \mathsf{afold}_F(g, h)$$

**Proposition 5.1** If $\mathcal{C}$ is a cartesian closed category, then every initial algebra is initial with accumulating parameters.

**Proof** The proof of this proposition is exactly the same as the one showed for Proposition 4.1 except for the fact that we have to replace $F^X$ by $\overline{F}$. $\square$

In Section 4, we used Proposition 3.8, actually an instance of it given by equation (6), to derive expansions of pfold for particular datatypes. In the case of afold, however, it is necessary to state a slightly different property that takes account of the presence of $\theta^{F,g}$ as part of the distributive law. The new property assumes composite functors of the form $F = G_1 + \cdots + G_n$. We present here the simple case of $F = G_1 + G_2$, since it suffices for the examples we show below; the generalization to $n$ summands is immediate.

**Proposition 5.2** Let $(N, \epsilon, -^{\#})$ be the product comonad. Let $F = G_1 + G_2$ be a composite functor. Let the accumulating function $g : F1 \times X \to X$ be given by $g = [g_1, g_2] \circ d$, with $g_i : G_i1 \times X \to X$,

$i = 1, 2$. Then, for every $f : NA \to B$, the following diagram commutes:

$$\begin{tikzcd} N(G_1A + G_2A) \arrow[r, "(\overline{F}f)^{\#}"] \arrow[d, "\delta^+"'] & N(G_1B + G_2B) \arrow[d, "\delta^+"] \\ NG_1A + NG_2A \arrow[r, "(\overline{G_1}f)^{\#} + (\overline{G_2}f)^{\#}"'] & NG_1B + NG_2B \end{tikzcd}$$

**Proof** Recall that, for the product comonad, $\delta^+ = d$. The commutativity of the desired diagram follows from the commutativity of the following one.

$$\begin{tikzcd} N(G_1A + G_2A) \arrow[r, "(\delta^+ \circ \theta^{F,g})^{\#}"] \arrow[d, "\delta^+"'] \arrow[dr, phantom, "(I)"] & N(NG_1A + NG_2A) \arrow[r, "N(\widetilde{G_1}f + \widetilde{G_2}f)"] \arrow[d, "\delta^+"'] \arrow[dr, phantom, "(II)"] & N(G_1B + G_2B) \arrow[d, "\delta^+"] \\ NG_1A + NG_2A \arrow[r, "(\theta^{G_1,g_1})^{\#} + (\theta^{G_2,g_2})^{\#}"'] & N^2G_1A + N^2G_2A \arrow[r, "N\widetilde{G_1}f + N\widetilde{G_2}f"'] & NG_1B + NG_2B \end{tikzcd}$$

In fact, observe that

$$\begin{aligned} (\overline{F}f)^{\#} &= N(\widetilde{G_1}f + \widetilde{G_2}f) \circ (\delta^+ \circ \theta^{F,g})^{\#} \\ (\overline{G_1}f)^{\#} + (\overline{G_2}f)^{\#} &= (N\widetilde{G_1}f + N\widetilde{G_2}f) \circ ((\theta^{G_1,g_1})^{\#} + (\theta^{G_2,g_2})^{\#}) \end{aligned}$$

$(II)$ commutes by naturality of $\delta^+$. The proof of (I) is given by the following calculation. Some of its steps are, in turn, the result of straightforward but tedious calculations. Let us define $f_i = g_i \circ G_i(!_A)$, with $i = 1, 2$. Hence, $\theta^{G_i,g_i} = \langle \pi_1, f_i \rangle$. Moreover, it is easy to see that $\theta^{F,g} = \langle \pi_1, [g_1, g_2] \circ d \circ NF(!_A) \rangle = \langle \pi_1, [f_1, f_2] \circ d \rangle$.

$$\begin{aligned} \delta^+ \circ (\delta^+ \circ \theta^{F,g})^{\#} &= d \circ (d \circ \langle \pi_1, [f_1, f_2] \circ d \rangle)^{\#} \\ &= d \circ N(d \circ \langle \pi_1, [f_1, f_2] \circ d \rangle) \circ \gamma \\ &= d \circ N((\langle \pi_1, f_1 \rangle + \langle \pi_1, f_2 \rangle) \circ d) \circ \gamma \\ &= d \circ N(\theta^{G_1,g_1} + \theta^{G_2,g_2}) \circ Nd \circ \gamma \\ &= (N\theta^{G_1,g_1} + N\theta^{G_2,g_2}) \circ d \circ Nd \circ \gamma \\ &= (N\theta^{G_1,g_1} + N\theta^{G_2,g_2}) \circ (\gamma + \gamma) \circ d \\ &= ((\theta^{G_1,g_1})^{\#} + (\theta^{G_2,g_2})^{\#}) \circ \delta^+ \end{aligned}$$

□

Before showing examples of afold on specific datatypes, we derive a generic expansion of this operator for the case that the datatype signature is of the form $F = G_1 + G_2$. In fact, that is the form the base functors of the examples have. Let $h = [h_1, h_2] \circ d : NFA \to A$ and let $g = [g_1, g_2] \circ d : F1 \times X \to X$. Thus, afold is the unique arrow $f = \mathsf{afold}_F(g, h)$ that makes this diagram commute:

$$\begin{tikzcd} N(G_1\,\mu F + G_2\,\mu F) \arrow[rr, "Nin_F"] \arrow[d, "(\overline{F}f)^{\#}"'] & & N\mu F \arrow[d, "f"] \\ N(G_1A + G_2A) \arrow[r, "d"'] & NG_1A + NG_2A \arrow[r, "{[h_1, h_2]}"'] & A \end{tikzcd}$$

Applying Proposition 5.2 we obtain this equation

$$f \circ Nin_F = [h_1, h_2] \circ ((\overline{G_1}f)^{\#} + (\overline{G_2}f)^{\#}) \circ d$$

Let $in_F = [c_1, c_2]$. If we pre-compose both sides of this equation with $d^{-1}$ and apply product and coproduct laws we get

$$f \circ [c_1 \times \mathsf{id}_X, c_2 \times \mathsf{id}_X] = [h_1 \circ (\overline{G_1} f)^{\#}, h_2 \circ (\overline{G_2} f)^{\#}]$$

Hence, by case analysis we have

$$f \circ (c_1 \times \mathsf{id}_X) = h_1 \circ (\overline{G_1} f)^{\#} \qquad f \circ (c_2 \times \mathsf{id}_X) = h_2 \circ (\overline{G_2} f)^{\#}$$

where $\overline{G_i} f = \widetilde{G_i} f \circ \theta^{G_i, g_i} = \widetilde{G_i} f \circ \langle \pi_1, g_i \circ (G_i! \times \mathsf{id}_X)\rangle$.

**Example 5.3**

1. For the natural numbers, $f = \mathsf{afold}_K(g, h)$ is such that

   $$f \circ (\mathsf{zero} \times \mathsf{id}_X) = h_1 \qquad f \circ (\mathsf{succ} \times \mathsf{id}_X) = h_2 \circ \langle f \circ \langle \pi_1, g_2 \circ (! \times \mathsf{id}_X)\rangle, \pi_2\rangle$$

   where $h = [h_1, h_2] \circ d : (1 + A) \times X \to A$ and $g = [g_1, g_2] \circ d : (1 + 1) \times X \to X$. In functional notation, $f(\mathsf{zero}, x) = h_1(x)$ and $f(\mathsf{succ}(n), x) = h_2(f(n, g_2(x)), x)$.

2. For lists, $f = \mathsf{afold}_{L_A}(g, h)$ is such that

   $$f(\mathsf{nil}, x) = h_1(x) \qquad f(\mathsf{cons}(a, \ell), x) = h_2(a, f(\ell, g_2(a, x)), x)$$

   where $h = [h_1, h_2] \circ d : (1 + A \times B) \times X \to B$ and $g = [g_1, g_2] \circ d : (1 + A \times 1) \times X \to X$.
   For example, the $\mathsf{asum}$ function is given by $\mathsf{asum} = \mathsf{afold}(g, h) : \mathbb{N}^* \times \mathbb{N} \to \mathbb{N}^*$, with $h_1(x) = [x]$, $h_2(a, l, x) = \mathsf{cons}(x, l)$, $g_1(x) = x$, $g_2(a, x) = a + x$.

3. For binary trees, $f = \mathsf{afold}_{B_A}(g, h)$ is such that

   $$f(\mathsf{empty}, x) = h_1(x) \qquad f(\mathsf{node}(t, a, u), x) = h_2(f(t, g_2(a, x)), a, f(u, g_2(a, x)), x)$$

   where $h = [h_1, h_2] \circ d : (1 + C \times A \times C) \times X \to C$ and $g = [g_1, g_2] \circ d : (1 + 1 \times A \times 1) \times X \to X$.

$\square$

Now, we present some laws for afold. The *identity law* states that, whatever the action of the accumulating function is, an afold on the lifting of the initial algebra simply returns the element of the datatype that it takes as input. For every $g$,

$$\mathsf{afold}_F(g, \widetilde{in_F}) = \epsilon_{\mu F}$$

The identity law is a consequence of the fact that $\overline{F}$ preserves identities. The next law is a *fusion law*. It states that the composition of a afold with a pure homomorphism is again a afold.

$$f \circ h = k \circ NFf \;\Rightarrow\; f \circ \mathsf{afold}_F(g, h) = \mathsf{afold}_F(g, k) \tag{10}$$

The following law states that, fixed an accumulating function $g$, (the lifting of) every fold can be seen as an afold that does not make use of the accumulators.

$$\widetilde{(\!|h|\!)_F} = \mathsf{afold}_F(g, \widetilde{h})$$

An *acid rain law* can also be established for afold. We call it *afold-fold fusion* since it permits to fuse certain kinds of afolds with folds. The shape of the definition and the notion of transformer employed by this law coincide with those for pfold-fold fusion (9). Let us recall the notion of transformer. We say that a function $\boldsymbol{T} : (FA \to A) \to (NGA \to A)$ is a *transformer* if it converts $F$-algebras into $\overline{G}$-algebras in such a way that, if $f : A \to B$ is a homomorphism between $h : FA \to A$ and $h' : FB \to B$, then it is a pure homomorphism between the corresponding $\overline{G}$-algebras. That is, if $f \circ h = h' \circ Ff$ then $f \circ \boldsymbol{T}(h) = \boldsymbol{T}(h') \circ NGf$.

$$\boldsymbol{T} \text{ transformer} \;\Rightarrow\; (\!|h|\!)_F \circ \mathsf{afold}_G(g, \boldsymbol{T}(in_F)) = \mathsf{afold}_G(g, \boldsymbol{T}(h))$$

The proof of this law is as follows. Consider an algebra $h : FA \to A$. Since every fold is a homomorphism, by definition of transformer we have that $(\!|h|\!)_F : \mu F \to A$ is also a pure homomorphism between the algebras $\boldsymbol{T}(in_F) : NG\mu F \to \mu F$ and $\boldsymbol{T}(h) : NGA \to A$. Therefore, by afold fusion (10), it follows that $(\!|h|\!)_F \circ \mathsf{afold}_G(\boldsymbol{T}(in_F)) = \mathsf{afold}_G(\boldsymbol{T}(h))$, which is the desired result.

**Example 5.4** The function $\mathsf{cut} : \mathsf{btree}(A) \to \mathsf{btree}(A)$ takes a tree and removes all subtrees whose root has already occurred in any of its ancestors.

$$\mathsf{cut}(t) = \mathsf{prune}(t, \mathsf{nil})$$

where $\mathsf{prune} : \mathsf{btree}(A) \times A^* \to \mathsf{btree}(A)$ is such that

$$\begin{aligned}
\mathsf{prune}(\mathsf{empty}, \ell) &= \mathsf{empty} \\
\mathsf{prune}(\mathsf{node}(t, a, t'), \ell) &= \textbf{if } a \in \ell \textbf{ then } \mathsf{empty} \\
&\quad\ \textbf{else } \mathsf{node}(\mathsf{prune}(t, \mathsf{cons}(a, \ell)), a, \mathsf{prune}(t', \mathsf{cons}(a, \ell)))
\end{aligned}$$

This function can be written as an afold, $\mathsf{prune} = \mathsf{afold}_{B_A}(\mathsf{acc\text{-}pr}, \mathsf{alg\text{-}pr}) : N\mathsf{btree}(A) \to \mathsf{btree}(A)$, where $NA = A \times A^*$. The accumulating function $\mathsf{acc\text{-}pr} : B_A 1 \times A^* \to A^*$ is defined by

$$\begin{aligned}
\mathsf{acc\text{-}pr}(x, \ell) \;=\; & \textbf{case } x \textbf{ of} \\
& \quad \mathsf{inl}(u) \quad\quad\;\; \to \ell \\
& \quad \mathsf{inr}(u, a, u') \to \mathsf{cons}(a, \ell)
\end{aligned}$$

whereas the comonadic algebra $\mathsf{alg\text{-}pr} : NB_A(\mathsf{btree}(A)) \to \mathsf{btree}(A)$ is given by,

$$\begin{aligned}
\mathsf{alg\text{-}pr}(x, \ell) \;=\; & \textbf{case } x \textbf{ of} \\
& \quad \mathsf{inl}(u) \quad\quad\; \to \mathsf{empty} \\
& \quad \mathsf{inr}(t, a, t') \to \textbf{if } a \in \ell \textbf{ then } \mathsf{empty} \textbf{ else } \mathsf{node}(t, a, t')
\end{aligned}$$

This algebra can be written as $\mathsf{alg\text{-}pr} = \boldsymbol{T}([\mathsf{empty}, \mathsf{node}])$, where $\boldsymbol{T} : (B_A C \to C) \to (NB_A C \to C)$ is the transformer showed in Example 4.5.

Like in Example 4.5, we want to count the number of nodes that remain in a tree after pruning.

$$\mathsf{count} \;=\; \mathsf{btree}(A) \times A^* \xrightarrow{\;\mathsf{prune}\;} \mathsf{btree}(A) \xrightarrow{\;\mathsf{size}\;} \mathbb{N}$$

where recall from Example 4.5 that $\mathsf{size} = (\!|\mathsf{alg\text{-}s}|\!)_{B_A}$. Using afold-fold fusion we can transform this composition into a afold, eliminating as a result the generation of the intermediate tree.

$$\mathsf{count} = \mathsf{size} \circ \mathsf{prune} = \mathsf{afold}_{B_A}(\mathsf{acc\text{-}pr}, \boldsymbol{T}(\mathsf{alg\text{-}s}))$$

Expanding the afold we obtain

$$\begin{aligned}
\mathsf{count}(\mathsf{empty}, \ell) &= \mathsf{zero} \\
\mathsf{count}(\mathsf{node}(t, a, t'), \ell) &= \textbf{if } a \in \ell \textbf{ then } \mathsf{zero} \\
&\quad\ \textbf{else } 1 + \mathsf{count}(t, \mathsf{cons}(a, \ell)) + \mathsf{count}(t', \mathsf{cons}(a, \ell))
\end{aligned}$$

□

# 6 Conclusions and Future Work

This paper presented some results in an attempt to study recursive operators with effects modeled by comonads. We showed two common applications that neatly combine the product comonad with structural recursion, obtaining as a result two instances of comonadic fold.

Similar results can be achieved for other operators, like unfold or primitive recursion (paramorphisms [23]). It is to notice that the analysis of the interaction between comonads and corecursion is, in principle, not so interesting as it is in the case of (structural) recursion. The reason is that, in a similar manner as monadic folds reduce to folds (see e.g. [12, 25]), comonadic unfolds can esaily be reduced to unfolds. In fact, for $h : A \to NFA$, *comonadic unfold* is defined by the unique arrow $f : NA \to \nu F$ that makes this diagram commute:

$$\begin{tikzcd}
NFA \arrow[d, "\widetilde{F}f"'] & NA \arrow[l, "h^{\#}"'] \arrow[d, "f^{\#}"] \\
F\nu F & N\nu F \arrow[l, "\widetilde{out}_F"]
\end{tikzcd}$$

But, since $\widetilde{F} f = F f \circ \delta^F$ and $\widetilde{out_F} \circ f^{\#} = out_F \circ f$, we have that $out_F \circ f = F f \circ (\delta^F_A \circ h)$. Therefore, by finality $f = [\![(\delta^F_A \circ h)]\!]_F$. In other words, independently of the particular comonad, a comonadic unfold exists for every coinductive type.

Other directions for future study are::

- The search of more interesting cases of comonads that neatly interact with recursion.
- The derivation of further laws for accumulations, for example, laws involving type functors.
- The application of the accumulation strategy [4, 3] in connection with our notion of accumulation, and its comparasion with other approaches (like e.g. [18]).
- The combination of recursion with both monads and comonads.

**Acknowledgements** I would like to thank the anonymous referees for helpful suggestions and comments. Diagrams were drawn using Paul Taylor's macros.

## References

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

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

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

[4] R.S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4), October 1984.

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

[6] S. Brookes and S. Geva. Computational Comonads and Intensional Semantics. Technical Report CMU-CS-91-190, School of Computer Science, Carnegie Mellon University, 1991.

[7] S. Brookes and K. Van Stone. Monads and Comonads in Intensional Semantics. Technical Report CMU-CS-93-140, School of Computer Science, Carnegie Mellon University, 1993.

[8] R. Cockett. Introduction to Distributive Categories. *Mathematical Structures in Computer Science*, 3:277–307, 1993.

[9] R. Cockett and T. Fukushima. About Charity. Technical Report 92/480/18, University of Calgary, June 1992.

[10] R. Cockett and D. Spencer. Strong Categorical Datatypes I. In R.A.C. Seely, editor, *International Meeting on Category Theory 1991,* volume 13 of *Canadian Mathematical Society Conference Proceedings*, pages 141–169, 1991.

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

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

[13] J. Gibbons. Upwards and Downwards Accumulations on Trees. In R.S. Bird, C.C. Morgan, and J.C P. Woodcock, editors, *Mathematics of Program Construction,* LNCS 669. Springer-Verlag, 1993.

[14] J. Gibbons. Generic Downwards Accumulations. *Science of Computer Programming*, 37(1–3):37–65, 2000.

[15] J. Gibbons. Lecture Notes on Algebraic and Coalgebraic Methods for Calculating Functional Programs. In *Summer School on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Oxford, UK, April 2000.

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

[17] 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.

[18] Z. Hu, H. Iwasaki, and M. Takeichi. Calculating Accumulations. Technical Report METR 96-03, Faculty of Engineering, University of Tokyo, March 1996.

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

[20] 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.

[21] R. Kieburtz. Codata and Comonads in Haskell (Unpublished manuscript). Available from `http://www.cse.ogi.edu/~dick/dick.html`.

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

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

[24] 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.

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

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

[27] 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.

[28] A. Pardo. Fusion of Recursive Programs with Computational Effects. *Theoretical Computer Science* (to appear), 2000. Available from `http://www.fing.edu.uy/~pardo`.

[29] A. Pardo. A Calculational Approach to Strong Datatypes. In *Selected Papers from the 8th Nordic Workshop on Programming Theory.* Research Report 240, Department of Informatics, University of Oslo, 1997.

[30] 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.

[31] D. Turi. *Functorial Operational Semantics and its Denotational Dual.* PhD thesis, Free University, Amsterdam, June 1996.

[32] D. Turi and G. Plotkin. Towards a Mathematical Operational Semantics. In *LICS'97*, pages 280–291, 1997.

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

[34] R.F.C. Walters. Data Types in Distributive Categories. *Bull. Austral. Math. Soc.*, 40:79–82, 1989.