(Co)Monads from Inductive and Coinductive Types (Extended Abstract)

Tarmo Uustalu
Departamento de Informática, Universidade do Minho
Campus de Gualtar, P-4710-057 Braga, Portugal
email `tarmo@di.uminho.pt`

Abstract

Monads and comonads are important in functional programming as a modularization tool in language semantics, but also as a program structuring device, especially in generic programming. In this paper, we study monads and comonads arising from inductive and coinductive types, aiming at generic programming applications. It is known that the types of trees with variables and also the types of cotrees with variables of a fixed branching factor give rise to a monad. We show that the same is more generally true of both the inductive and coinductive types given by the partial applications $F'(A, -)$ of any bifunctor F' whose partial applications $F'(-, X)$ uniformly admit a monad structure. Dual constructions deliver comonads.

Keywords: *monads, comonads, inductive and coinductive types, recursion and corecursion schemes, trees and cotrees, substitution, redecoration, functional programming, genericity*

1 Introduction

The developments in both language design and programming methodology for functional programming have repeatedly demonstrated the usefulness of category-theory insights in the construction and organization of programming idioms. Initial algebras and final coalgebras, to give an example, are a succinct and very general mathematization of the ideas of inductive and coinductive types. The legitimacy of various structured recursion and corecursion schemes (circular definition schemes for functions with inductive domains and coinductive codomains) is conveniently checked category-theoretically. The central categorical notions of monad and comonad, to give another example, are in several programming contexts useful as means for

On leave from Inst. of Cybernetics, Tallinn Techn. Univ., Akadeemia tee 21, EE-126 18 Tallinn, Estonia, email `tarmo@cs.ioc.ee`.

uncovering or imposing structure. Monads were originally introduced into programming by Moggi [Mog91] as a modularization tool in language semantics and then quickly popularized by Wadler [Wad92] as a means to set up an infrastructure for representing and manipulating computations with effects also in actual programming. Comonads, although not as popular as monads, have been employed, e.g., to describe intensional semantics [BG92]. Kieburtz [Kie99] argues that comonads are good as a framework for the representation and manipulation of computations in context. Monads and comonads matter for inductive and coinductive types, too. It is, for instance, possible to specify recursion and corecursion schemes by means of comonads and monads equipped with distributive laws [UVP01, Bar01].

In this paper, we study monads and comonads arising from inductive and coinductive types. It is folklore knowledge (which, as a matter of fact, forms the starting point of the categorical approaches to universal algebra and representation and manipulation of programming language syntax [Man76, Tur96, TP97]) that the types of trees with variables of a fixed branching factor give rise to a monad with substitution as the extension operation. The equally plausible, but not so apparent fact that the same holds of the types of cotrees (non-wellfounded trees) with variables has been pointed out recently [AAV01, Mos01, G^{+}01]. We show that these two facts dualize for the types of what we call decorated trees and decorated cotrees: these determine comonads with redecoration as the coextension operation. Moreover, both the constructions for trees and cotrees with variables and those for decorated trees and cotrees are instances of significantly more general constructions: both the inductive and coinductive types given by the partial applications $F'(A, -)$ of any bifunctor F' whose partial applications $F'(-, X)$ uniformly admit a monad structure give rise to a monad, and a dual statement holds about comonads. Written down as programs, the constructions present an elegant example of seriously generic programming involving, among other things, generic uses of different generic recursion and corecursion combinators.

The paper may be seen as a successor to [UV99, Ven00, UVP01], which are all concerned with recursion and corecursion schemes. A discussion of decorated trees and cotrees appears in a related paper [UV01].

The organization of the paper is the following. In Section 2, we introduce the concepts of monad and comonad. In Section 3, we outline the basics about the categorical approach to inductive and coinductive types as initial algebras and final coalgebras. The tree and cotree (co)monad constructions are presented in Section 4, to be followed by the generalized constructions of general inductive and coinductive (co)monads in Section 5. In Section 6, we conclude. Appendix A contains a demonstration of a Haskell implementation of the generalized constructions.

Though motivated by issues in language design and programming methodology, the material of the paper is basically category-theoretic. We have therefore striven for a self-contained and elementary exposition, suppressing some abstractions that would have added conciseness, but at the expense of indirection of the connection to the concrete concepts and facts of interest. The reader is assumed to know the concepts of functor and natural transformation and the basics of the categorical approach to functional programming (i.e., the types-as-objects, functions-as-morphisms paradigm), incl. the highly relevant notions of coproduct, product and exponential objects which constitute the structure of a bicartesian closed category. The more specific concepts of initial algebra and final coalgebra and monad and comonad are briefly introduced. For a good introduction to initial algebras and final coalgebras from the programming perspective and to the categorical approach to functional programming in general, we refer the reader to [Fok92, BdM97]. The classic category-theory texts treating (co)monads are [Man76, BW84].

The notation used is fairly standard. The coproduct of objects A and B is written $(A+B, \mathsf{inl}_{A,B}, \mathsf{inr}_{A,B})$, the case analysis of $f : A \to C$, $g : B \to C$ is denoted $[\,f, g\,]$. The product of objects A and B is denoted $(A \times B, \mathsf{fst}_{A,B}, \mathsf{snd}_{A,B})$, the pairing of $f : C \to A$ and $g : C \to B$ is $\langle\, f, g\,\rangle$. The final object is denoted 1 and the final morphism from C is $\diamondsuit_C$. The exponential object "A raised to the power of B" is written $(B \Rightarrow A, \mathsf{ev}_{B,A})$ and the currying of a morphism $f : C \times B \to A$ is written $\mathsf{curry}(f)$. unit is the natural transformation $- \dot{\to} - \times 1$, assoc is the natural transformation $(-_1 \times -_2) \times -_3 \dot{\to} -_1 \times (-_2 \times -_3)$.

2 (Co)monads

2.1 Monads

A *monad* on a category $\mathcal{C}$ is a triple (M, η, μ) consisting of an endofunctor M on $\mathcal{C}$ (*underlying functor*), and two natural transformations $\eta : \mathsf{Id} \dot{\to} M$, $\mu : M \cdot M \dot{\to} M$ (*unit* and *multiplication*) such that

$$\begin{tikzcd}
MA \arrow[r, "M\eta_A"] \arrow[d, "\eta_{MA}"'] \arrow[dr, equal] & M(MA) \arrow[d, "\mu_A"] \\
M(MA) \arrow[r, "\mu_A"] & MA
\end{tikzcd}
\qquad
\begin{tikzcd}
M(M(MA)) \arrow[r, "M\mu_A"] \arrow[d, "\mu_{MA}"'] & M(MA) \arrow[d, "\mu_A"] \\
M(MA) \arrow[r, "\mu_A"] & MA
\end{tikzcd}$$

Monads are equivalent to Kleisli triples. A *Kleisli triple* on $\mathcal{C}$ is a triple $(M, \eta, -^\star)$ consisting of an endofunction M on $|\mathcal{C}|$ (underlying object mapping), a $|\mathcal{C}|$-indexed family η of morphisms $\eta_A : A \to MA$, and an operation $-^\star$ taking every morphism $f : A \to MB$ to a morphism $f^\star : MA \to MB$ (*extension* of f) such that $f^\star \circ \eta_A = f$, if $f : A \to MB$, $\eta_A^\star = \mathsf{id}_{MA}$, and $(g^\star \circ f)^\star = g^\star \circ f^\star$. A monad is converted into a Kleisli triple by putting $f^\star = \mu_B \circ Mf$, if $f : A \to MB$. For conversion in the opposite direction, one sets $Mf = (\eta_B \circ f)^\star$, if $f : A \to B$, and $\mu_A = \mathsf{id}_{MA}^\star$.

Some simple, but popular and important examples of monad constructions are the following.

- Given an object E in $\mathcal{C}$, the endofunctor $M = - + E$ on $\mathcal{C}$ is extended into a monad (M, η, μ) by setting $\eta_A = \mathsf{inl}_{A,E}$, $\mu_A = [\,\mathsf{id}_{MA}, \mathsf{inr}_{A,E}\,]$.
- Given an object E in $\mathcal{C}$, the endofunctor $M = E \Rightarrow -$ on $\mathcal{C}$ is extended into a monad (M, η, μ) by setting $\eta_A = \mathsf{curry}(\mathsf{fst}_{A,E})$, $\mu_A = \mathsf{curry}(\mathsf{ev}_{E,A} \circ \langle\, \mathsf{ev}_{E,MA}, \mathsf{snd}_{M(MA),E}\,\rangle)$.
- Given a monoid (E, e, m) in $\mathcal{C}$, the endofunctor $M = - \times E$ on $\mathcal{C}$ is extended into a monad (M, η, μ) by setting $\eta_A = (\mathsf{id}_A \times e) \circ \mathsf{unit}_A$, $\mu_A = (\mathsf{id}_A \times m) \circ \mathsf{assoc}_{A,E,E}$.

2.2 Comonads

A *comonad* on a category $\mathcal{C}$ consists of an endofunctor N on $\mathcal{C}$, and two natural transformations $\varepsilon : N \dot{\to} \mathsf{Id}$, $\delta : N \dot{\to} N \cdot N$ (*counit* and *comultiplication*) such that, for any object A,

$$\begin{tikzcd} N(NA) \arrow[r, "N\varepsilon_A"] & NA \\ NA \arrow[u, "\delta_A"] \arrow[r, "\delta_A"] \arrow[ur, equal] & N(NA) \arrow[u, "\varepsilon_{NA}"'] \end{tikzcd} \qquad \begin{tikzcd} N(NA) \arrow[r, "N\delta_A"] & N(N(NA)) \\ NA \arrow[u, "\delta_A"] \arrow[r, "\delta_A"] & N(NA) \arrow[u, "\delta_{NA}"'] \end{tikzcd}$$

Comonads are essentially the same as coKleisli triples. A *coKleisli triple* on $\mathcal{C}$ is a triple $(N, \varepsilon, -^\dagger)$ consisting of an endofunction N on $|\mathcal{C}|$ (underlying object mapping), a $|\mathcal{C}|$-indexed family of morphisms $\varepsilon_A : NA \to A$, and an operation $-^\dagger$ taking every morphism $f : NA \to B$ to a morphism $f^\dagger : NB \to NA$ (*coextension* of f) such that $\varepsilon_B \circ f^\dagger = f$, if $f : NA \to B$, ${\varepsilon_A}^\dagger = \mathsf{id}_{NA}$, and $(g \circ f^\dagger)^\dagger = g^\dagger \circ f^\dagger$. A comonad is converted into a coKleisli triple by putting $f^\dagger = Nf \circ \delta_A$, if $f : NA \to B$. For conversion in the opposite direction, one sets $Nf = (f \circ \varepsilon_A)^\dagger$, if $f : A \to B$, and $\delta_A = {\mathsf{id}_{NA}}^\dagger$.

The the simplest examples of comonads with relevance to programming are the following:

- Given an object E in $\mathcal{C}$, the endofunctor $N = - \times E$ on $\mathcal{C}$ is extended into a comonad (N, ε, δ) by setting $\varepsilon_A = \mathsf{fst}_{A,E}$, $\delta_A = \langle\, \mathsf{id}_{NA}, \mathsf{snd}_{A,E}\,\rangle$.
- Given a monoid (E, e, m) in $\mathcal{C}$, the endofunctor $N = E \Rightarrow -$ on $\mathcal{C}$ is extended into a comonad (N, ε, δ) by setting $\varepsilon_A = \mathsf{ev}_{NA,E} \circ (\mathsf{id}_{NA} \times e) \circ \mathsf{unit}_{NA}$, $\delta_A = \mathsf{curry}(\mathsf{curry}(\mathsf{ev}_{NA,E} \circ (\mathsf{id}_{NA} \times m) \circ \mathsf{assoc}_{NA,E,E}))$.

3 Inductive and coinductive types and functors

Given an endofunctor F on a category $\mathcal{C}$, an F-*algebra* is a pair (A, φ) consisting of an object A (underlying object or *carrier*) and a morphism $\varphi : FA \to A$ (*algebra structure*). An F-*algebra map* from (A, φ) to (B, ψ) is a morphism $h : A \to B$ such that

$$\begin{tikzcd} FA \arrow[r, "Fh"] \arrow[d, "\varphi"'] & FB \arrow[d, "\psi"] \\ A \arrow[r, "h"] & B \end{tikzcd}$$

An F-*coalgebra*, dually, is a pair (A, φ) consisting of an object A (underlying object or *carrier*) and a morphism $\varphi : A \to FA$ (*coalgebra structure*). An F-*coalgebra map* from (A, φ) to (B, ψ) is a morphism $h : A \to B$ such that

$$\begin{tikzcd} FA \arrow[r, "Fh"] & FB \\ A \arrow[u, "\varphi"] \arrow[r, "h"] & B \arrow[u, "\psi"'] \end{tikzcd}$$

Both the F-algebras and the F-coalgebras form a category (with identity and composition inherited from $\mathcal{C}$).

The initial object of the category of F-algebras (if it exists) is called the *initial F-algebra*, written $(\mu F, \mathsf{in}_F)$. Under the view of $\mathcal{C}$ as a category of types, it is exactly what type theorists and functional programmers call the F-based *inductive type* (*data type*), together with the accompanying *data constructor*. By the initiality, for any morphism $\varphi : FA \to A$, there exists a unique morphism $h : \mu F \to A$ such that

$$\begin{tikzcd}
F(\mu F) \arrow[r, "Fh"] \arrow[d, "\mathsf{in}_F"'] & FA \arrow[d, "\varphi"] \\
\mu F \arrow[r, "h"] & A
\end{tikzcd}$$

This h is often called the *catamorphism* or *fold* of φ and denoted $(\!|\, \varphi \,|\!)_F$ or $\mathsf{fold}_F(\varphi)$. Conceptually, h is the function determined by φ as a step function by the most basic *recursion* scheme of *iteration*.

It is also true that, for any morphism $\varphi : F(A \times \mu F) \to A$, there exists a unique morphism $h : \mu F \to A$ such that

$$\begin{tikzcd}
F(\mu F) \arrow[r, "F\langle h, \mathsf{id}_{\mu F} \rangle"] \arrow[d, "\mathsf{in}_F"'] & F(A \times \mu F) \arrow[d, "\varphi"] \\
\mu F \arrow[r, "h"] & A
\end{tikzcd}$$

This h, sometimes called the *paramorphism* of φ and written $\langle\!|\, \varphi \,|\!\rangle_F$, is the function determined by φ by *primitive recursion*. By its "direct" expressive power, primitive recursion is stronger than iteration. With the help of "tupling", however, it is easily reduced to iteration: for any $\varphi : F(A \times \mu F) \to A$, it holds that $\langle\!|\, \varphi \,|\!\rangle_F = \mathsf{fst}_{A,\mu F} \circ (\!|\, \langle \varphi, \mathsf{in}_F \circ F\mathsf{snd}_{A,\mu F} \rangle \,|\!)_F$.

The initial algebra structure in_F is an isomorphism with inverse $\mathsf{in}_F^{-1} = (\!|\, F\mathsf{in}_F \,|\!)_F : \mu F \to F\mu F$, the *data destructor* for μF. Hence, μF carries not only an algebra structure, but a coalgebra structure, too.

The final object in the category of F-coalgebras (if it exists) is called the *final F-coalgebra* and denoted $(\nu F, \mathsf{out}_F)$. If $\mathcal{C}$ is a category of types, then the final F-coalgebra is the F-based *coinductive type* (*codata type*), together with its accompanying *codata destructor*.

For any morphism $\varphi : A \to FA$, there exists a unique morphism $h : A \to \nu F$ such that

$$\begin{tikzcd}
FA \arrow[r, "Fh"] & F(\nu F) \\
A \arrow[u, "\varphi"] \arrow[r, "h"] & \nu F \arrow[u, "\mathsf{out}_F"']
\end{tikzcd}$$

This h, commonly called the *anamorphism* or *unfold* of φ and written $[\!(\, \varphi \,)\!]_F$ or $\mathsf{unfold}_F(\varphi)$, represents the function produced from φ by *coiteration*.

For any morphism $\varphi : A \to F(A + \nu F)$, there is a unique morphism $h : A \to \nu F$ such that

$$\begin{tikzcd}
F(A + \nu F) \arrow[r, "F[h, \mathsf{id}_{\nu F}]"] & F(\nu F) \\
A \arrow[u, "\varphi"] \arrow[r, "h"] & \nu F \arrow[u, "\mathsf{out}_F"']
\end{tikzcd}$$

This morphism h, which stands for the function determined by φ by *primitive corecursion*, is referred to as the *apomorphism* of φ and written $[\!\langle \varphi \rangle\!]_F$. Primitive corecursion is reduced to coiteration with recourse to "varianting": for any $\varphi : A \to F(A + \nu F)$, it is the case that $[\!\langle \varphi \rangle\!]_F = [\!([\varphi, F\mathsf{inr}_{A,\nu F} \circ \mathsf{out}_F])\!]_F \circ \mathsf{inl}_{A,\nu F}$.

The final coalgebra structure out_F is an isomorphism with inverse $\mathsf{out}_F^{-1} = (\!| F\mathsf{out}_F |\!)_F : F(\nu F) \to \nu F$, the *codata constructor* for νF.

Some popular examples of inductive and coinductive types include the following:

- Let N be the endofunctor $1 + -$ on $\mathcal{C}$. Then the objects $\mu\mathsf{N}$ and $\nu\mathsf{N}$ of $\mathcal{C}$ are the types of naturals and conaturals (naturals plus "infinity").

- For an object E of $\mathcal{C}$, let L_E be the endofunctor $1 + E \times -$ on $\mathcal{C}$. Then $\mu\mathsf{L}_E$ and $\nu\mathsf{L}_E$ are the types of lists and colists (possibly infinite lists) over E.

Given an endobifunctor F' on $\mathcal{C}$, the initial algebras and final coalgebras of its partial applications $F'(A, -)$ (if they all exist) are most economically studied as families. The initial algebras for $F'(A, -)$ give rise to an endofunctor $\hat{\mu}F'$ on $\mathcal{C}$ defined by $(\hat{\mu}F')A = \mu(F'(A, -))$, $(\hat{\mu}F')f = (\!| \mathsf{in}_{F'(B,-)} \circ F'(f, \mathsf{id}_{(\hat{\mu}F')B}) |\!)_{F'(A,-)}$, if $f : A \to B$. They also determine a natural isomorphism $\hat{\mathsf{in}}_{F'} : F' \cdot \langle \mathsf{Id}, \hat{\mu}F' \rangle \dot{\to} \hat{\mu}F'$ given by $(\hat{\mathsf{in}}_{F'})_A = \mathsf{in}_{F'(A,-)}$. Dually, the final coalgebras for $F'(A, -)$ determine an endofunctor $\hat{\nu}F'$ on $\mathcal{C}$ defined by $(\hat{\nu}F')A = \nu(F'(A, -))$ and $(\hat{\nu}F')f = [\!(F'(f, \mathsf{id}_{(\hat{\nu}F')A}) \circ \mathsf{out}_{F'(A,-)})\!]_{F'(B,-)}$, if $f : A \to B$, and a natural isomorphism $\hat{\mathsf{out}}_{F'} : \hat{\nu}F' \dot{\to} F' \cdot \langle \mathsf{Id}, \hat{\nu}F' \rangle$ given by $(\hat{\mathsf{out}}_{F'})_A = \mathsf{out}_{F'(A,-)}$. The functors $\hat{\mu}F'$ and $\hat{\nu}F'$ are sometimes called the F'-given *inductive functor* (*data functor*) and *coinductive functor* (*codata functor*).

We shall have use for the following examples of inductive and coinductive functors.

- Given an endofunctor H on $\mathcal{C}$, let T^H be the endobifunctor $-_1 + H-_2$ on $\mathcal{C}$. Then the objects $(\hat{\mu}\mathsf{T}^H)A$ and and $(\hat{\nu}\mathsf{T}^H)A$ of $\mathcal{C}$ represent the types of H-branching trees and cotrees (non-wellfounded trees) with variables. [If H is polynomial, H-branching trees with variables are (the syntax trees of) H-terms in the sense of universal algebra.] In category theory, the initial algebras $((\hat{\mu}\mathsf{T}^H)A, (\hat{\mathsf{in}}_{\mathsf{T}^H})_A)$ are known as *free H-algebras*.

- Given an endofunctor H on $\mathcal{C}$, let D^H be the endobifunctor $-_1 \times H-_2$ on $\mathcal{C}$. Then the objects $(\hat{\mu}\mathsf{D}^H)A$ and $(\hat{\nu}\mathsf{D}^H)A$ capture the types of decorated H-branching trees and cotrees (non-wellfounded trees). The final coalgebras $((\hat{\nu}\mathsf{T}^H)A, (\hat{\mathsf{out}}_{\mathsf{T}^H})_A)$ bear the name of *cofree H-coalgebras*.

4 Tree and cotree (co)monads

4.1 Tree and cotree monads

It is well known that, given a functor H, the types of H-branching trees with variables give rise to a monad or, to be more precise, the functor $\hat{\mu}\mathsf{T}^H$ extends to a monad. To define this monad, write (M, α) for $(\hat{\mu}\mathsf{T}^H, \hat{\mathsf{in}}_{\mathsf{T}^H})$. Decompose the natural isomorphism $\alpha : \mathsf{T}^H \cdot \langle \mathsf{Id}, M\rangle \dot{\to} M$ into two natural transformations $\eta : \mathsf{Id} \dot{\to} M$ and $\tau : H \cdot M \dot{\to} M$ by letting $\eta_A = \alpha_A \circ \mathsf{inl}_{A,H(MA)}$, $\tau_A = \alpha_A \circ \mathsf{inr}_{A,H(MA)}$, so that $\alpha_A = [\,\eta_A, \tau_A\,]$. The unit of the monad on M is η. The multiplication $\mu : M{\cdot}M \dot{\to} M$ is defined by iteration: μ_A is the unique morphism h such that

$$\begin{tikzcd}
MA + H(M(MA)) \arrow[r, "\mathsf{id}_{MA}+Hh"] \arrow[d, "{\alpha_{MA}=[\,\eta_{MA},\tau_{MA}\,]}"'] & MA + H(MA) \arrow[d, "{[\,\mathsf{id}_{MA},\tau_A\,]}"] \\
M(MA) \arrow[r, "h"] & MA
\end{tikzcd}$$

Conceptually, η_A is the function that takes a variable from A and makes it into a tree with variables from A, while μ_A is a function that takes a tree with variables from MA and "flattens" it into a tree with variables from A by replacing each of its variables by the tree this variable constitutes. The extension $f^\star : MA \to MB$ of a morphism $A \to MB$ takes a tree over A and replaces all its variables with trees over B using f as a guideline, producing a tree over B. Hence, $f^\star$ is the substitution function given by f as a substitution rule. Remarkably, several important properties of substitution follow just from the monad laws for M.

It seems rather intuitive that, for cotrees with variables, it ought to be possible to define something analogous to the flattening and substitution that exist for trees. This is indeed the case: the functor $\hat{\nu}\mathsf{T}^H$ also admits a monad structure and in a good sense a very similar one to that on $\hat{\mu}\mathsf{T}^H$ [AAV01, Mos01, G$^+$01]. To define it, write (M, α) for $(\hat{\nu}\mathsf{T}^H, \hat{\mathsf{out}}^{-1}_{\mathsf{T}^H})$. Decompose again $\alpha : \mathsf{T}^H \cdot \langle \mathsf{Id}, M\rangle \dot{\to} M$ into $\eta : \mathsf{Id} \dot{\to} M$ and $\tau : H \cdot M \dot{\to} M$ as before; η is the monad unit on M. Defining the multiplication $\mu : M \cdot M \dot{\to} M$ needs primitive corecursion (coiteration suffices only in conjunction with "varianting" and yields then essentially the same definition in a clumsier formulation): μ_A is defined as the unique morphism h such that

$$\begin{tikzcd}
A + H(M(MA) + MA) \arrow[r, "{\mathsf{id}_A + H[\,h,\mathsf{id}_{MA}\,]}"] & A + H(MA) \\
(A + H(M(MA) + MA)) + H(M(MA) + MA) \arrow[u, "{[\,\mathsf{id}_{A+H(M(MA)+MA)},\mathsf{inl}_{A,H(M(MA)+MA)}\,]}"] & \\
(A + H(MA)) + H(M(MA)) \arrow[u, "(\mathsf{id}_A+H\mathsf{inr}_{M(MA),MA})+H\mathsf{inr}_{M(MA),MA}"] & \\
MA + H(M(MA)) \arrow[u, "\alpha_A^{-1}+\mathsf{id}_{H(M(MA))}"] & \\
M(MA) \arrow[u, "\alpha_{MA}^{-1}"] \arrow[r, "h"] & MA \arrow[uuuu, "\alpha_A^{-1}"']
\end{tikzcd}$$

Notably, this description is equivalent to the more appealing one that constituted the iterative definition of μ in the case of $M = \hat{\mu}\mathsf{T}^H$.

4.2 Tree and cotree comonads

The functors $\hat{\mu}\mathsf{D}^H$ and $\hat{\nu}\mathsf{D}^H$ are the formal duals of the functors $\hat{\nu}\mathsf{T}^H$ and $\hat{\mu}\mathsf{T}^H$. It is therefore a matter of straightforward dualization to establish that they both admit the structure of a comonad.

To define the comonad on $\hat{\mu}\mathsf{D}^H$, write (N, β) for $(\hat{\mu}\mathsf{D}^H, \hat{\mathsf{in}}^{-1}_{\mathsf{D}^H})$. Decompose the natural isomorphism $\beta : N \mathrel{\dot{\to}} \mathsf{D}^H \cdot \langle \mathsf{Id}, N \rangle$ into two natural transformations $\varepsilon : N \mathrel{\dot{\to}} \mathsf{Id}$ and $\theta : N \to H \cdot N$ by setting $\varepsilon_A = \mathsf{fst}_{A,H(NA)} \circ \beta_A$ and $\theta_A = \mathsf{snd}_{A,H(NA)} \circ \beta_A$; the counit of the comonad structure on N is ε. The comultiplication $\delta : N \to N \cdot N$ is defined by primitive recursion: δ_A is defined as the unique morphism h such that

$$\begin{tikzcd}
A \times H(NA) \arrow[r, "\mathsf{id}_A \times H\langle h, \mathsf{id}_{NA}\rangle"] \arrow[dddd, "\beta_A^{-1}"'] & A \times H(N(NA) \times NA) \arrow[d, "\langle \mathsf{id}_{A \times H(N(NA) \times NA)}, \mathsf{snd}_{A,H(N(NA) \times NA)} \rangle"] \\
 & (A \times H(N(NA) \times NA)) \times H(N(NA) \times NA) \arrow[d, "(\mathsf{id}_A \times H\mathsf{snd}_{N(NA),NA}) \times H\mathsf{fst}_{N(NA),NA}"] \\
 & (A \times H(NA)) \times H(N(NA)) \arrow[d, "\beta_A^{-1} \times \mathsf{id}_{H(N(NA))}"] \\
 & NA \times H(N(NA)) \arrow[d, "\beta_{NA}^{-1}"] \\
NA \arrow[r, "h"] & N(NA)
\end{tikzcd}$$

To define the comonad on $\hat{\nu}\mathsf{D}^H$, write (N, β) for $(\hat{\nu}\mathsf{D}^H, \hat{\mathsf{out}}_{\mathsf{D}^H})$. Decompose $\beta : N \mathrel{\dot{\to}} \mathsf{D}^H \cdot \langle \mathsf{Id}, N \rangle$ into $\varepsilon : N \mathrel{\dot{\to}} \mathsf{Id}$ and $\theta : N \to H \cdot N$ as above; the counit is ε. The comultiplication $\delta : N \mathrel{\dot{\to}} N \cdot N$ is defined by coiteration: δ_A is the unique morphism h such that

$$\begin{tikzcd}
NA \times H(NA) \arrow[r, "\mathsf{id}_{NA} \times Hh"] & NA \times H(N(NA)) \\
NA \arrow[u, "\langle \mathsf{id}_{NA}, \theta_A \rangle"] \arrow[r, "h"] & N(NA) \arrow[u, "\langle \varepsilon_{NA}, \theta_{NA} \rangle = \beta_{NA}"']
\end{tikzcd}$$

Importantly, the primitive corecursive definition of δ in the case $N = \hat{\mu}\mathsf{D}^H$ is also equivalent to this description.

The comonads arising from the types of decorated (co)trees make good programming sense [UV01], bearing relevance, e.g., for attribute grammars. The counit ε_A takes an A-decorated tree and returns its decoration. The comultiplication δ_A takes an A-decorated tree and "inflates" it into an NA-decorated tree by replacing the A-decoration of each of its subtrees with this subtree itself. The coextension $f^\dagger : NA \to NB$ of a morphism $f : NA \to B$ takes an A-decorated tree and replaces the A-decoration of each of its subtrees by the B-decoration assigned to this subtree by f; if f is viewed as a redecoration rule, $f^\dagger$ is the corresponding redecoration function. Certain properties of redecoration follow from the comonad laws.

5 General inductive and coinductive (co)monads

5.1 General inductive and coinductive monads

When defining monads on the functors $\hat{\mu}\mathsf{T}^H$ and $\hat{\nu}\mathsf{T}^H$ we overlooked the fact that their base bifunctor T^H had something to do with monads already: the partial applications $\mathsf{T}^H(-,X)$ carry a monad structure, in fact, a monad structure uniform in X. A fresh look at the definitions given above makes it clear that they work exactly because of this circumstance. This directs us to the formulation of a general procedure for producing monads from families of inductive and coinductive types.

Let us say that a *bimonad* on $\mathcal{C}$ is triple (M',η',μ') consisting of a endobifunctor M' on $\mathcal{C}$ and two natural transformations $\eta' : \mathsf{Fst} \dot{\to} M'$ and $\mu' : M' \cdot \langle M', \mathsf{Snd}\rangle \dot{\to} M'$ such that

$$\begin{tikzcd}
M'(A,X) \arrow[r, "M'(\eta'_{A,X}, \mathsf{id}_X)"] \arrow[d, "\eta'_{M'(A,X),X}"'] \arrow[dr, equal] & M'(M'(A,X),X)) \arrow[d, "\mu'_{A,X}"] \\
M'(M'(A,X),X) \arrow[r, "\mu'_{A,X}"] & M'(A,X)
\end{tikzcd}
\quad
\begin{tikzcd}
M'(M'(M'(A,X),X),X) \arrow[r, "M'(\mu'_{A,X}, \mathsf{id}_X)"] \arrow[d, "\mu'_{M'(A,X),X}"'] & M'(M'(A,X),X) \arrow[d, "\mu'_{A,X}"] \\
M'(M'(A,X),X) \arrow[r, "\mu'_{A,X}"] & M'(A,X)
\end{tikzcd}$$

To give an example, the bifunctor T^H corresponding to a functor H is equipped with a bimonad structure by setting $\eta'_{A,X} = \mathsf{inl}_{A,HX}$, $\mu'_{A,X} = [\,\mathsf{id}_{\mathsf{T}^H(A,X)}, \mathsf{inr}_{A,HX}\,]$.

Now, given a bimonad (M',η',μ'), both the M'-based inductive functor $\hat{\mu}M'$ and coinductive functor $\hat{\nu}M'$ extend into a monad. Pleasantly, the constructions are not more complicated than those for the monads from tree and cotree types; on the contrary, they rather explain these special cases.

To define the monad structure on $\hat{\mu}M'$, write M for $\hat{\mu}M'$ and α for $\hat{\mathsf{in}}_{M'} : M' \cdot \langle \mathsf{Id}, M\rangle \dot{\to} M$. The unit $\eta : \mathsf{Id} \dot{\to} M$ of the monad with M as the underlying functor is defined via η' explicitly by putting $\eta_A = \alpha_A \circ \eta'_{A,MA}$. The multiplication $\mu : M \cdot M \dot{\to} M$ is defined via μ' by iteration: μ_A is determined as the unique h such that

$$\begin{tikzcd}
M'(MA, M(MA)) \arrow[r, "M'(\mathsf{id}_{MA}, h)"] \arrow[dddd, "\alpha_{MA}"'] & M'(MA, MA) \arrow[d, "M'(\alpha_A^{-1}, \mathsf{id}_{MA})"] \\
& M'(M'(A, MA), MA) \arrow[d, "\mu'_{A,MA}"] \\
& M'(A, MA) \arrow[dd, "\alpha_A"] \\
& \\
M(MA) \arrow[r, "h"] & MA
\end{tikzcd}$$

To specify the monad structure on $\hat{\nu}M'$, write M for $\hat{\nu}M'$ and α for $\hat{\mathsf{out}}_{M'}^{-1}$: $M' \cdot \langle \mathsf{Id}, M\rangle \dot{\to} M$. The unit $\eta : \mathsf{Id} \dot{\to} M$ is, just as before, defined explicitly from η'

by setting $\eta_A = \alpha_A \circ \eta'_{A,MA}$. The multiplication $\mu : M \cdot M \dot{\to} M$ is definable from μ', but only by primitive corecursion: μ_A is the unique morphism h such that

$$\begin{tikzcd}
M'(A, M(MA) + MA) \arrow[r, "M'(\mathsf{id}_A, [h, \mathsf{id}_{MA}])"] & M'(A, MA) \\
M'(M'(A, M(MA) + MA), M(MA) + MA) \arrow[u, "\mu'_{A,M(MA)+MA}"] & \\
M'(M'(A, MA), M(MA)) \arrow[u, "M'(M'(\mathsf{id}_A, \mathsf{inr}_{M(MA),MA}), \mathsf{inl}_{M(MA),MA})"] & \\
M'(MA, M(MA)) \arrow[u, "M'(\alpha_A^{-1}, \mathsf{id}_{M(MA)})"] & \\
M(MA) \arrow[u, "\alpha_{MA}^{-1}"] \arrow[r, "h"] & MA \arrow[uuuu, "\alpha_A^{-1}"']
\end{tikzcd}$$

In both cases, the defining characterization of μ is equivalent to this more symmetric characterization: μ_A is the unique morphism $h : M(MA) \to MA$ such that

$$\begin{tikzcd}
M'(M'(A, MA), M(MA)) \arrow[r, "M'(\mathsf{id}_{M'(A,MA)}, h)"] \arrow[d, "M'(\alpha_A, \mathsf{id}_{M(MA)})"'] & M'(M'(A, MA), MA) \arrow[d, "\mu'_{A,MA}"] \\
M'(MA, M(MA)) \arrow[d, "\alpha_{MA}"'] & M'(A, MA) \arrow[d, "\alpha_A"] \\
M(MA) \arrow[r, "h"] & MA
\end{tikzcd}$$

Checking the monad laws for M can be done for both cases at one go: it suffices to rely on the defining explicit definition of ε and the derived description of μ in combination with the knowledge that M' is a bimonad, M is a functor and α a natural isomorphism.

5.2 General inductive and coinductive comonads

Not surprisingly, the above general construction of monads from inductive and coinductive types is readily dualizable into one delivering comonads. Say that a *bicomonad* on $\mathcal{C}$ is a triple $(N', \varepsilon', \delta')$ consisting of an endobifunctor N' on $\mathcal{C}$ together with two natural transformations $\varepsilon' : N' \dot{\to} \mathsf{Fst}$ and $\delta' : N' \dot{\to} N' \cdot \langle N', \mathsf{Snd} \rangle$ such that

$$\begin{tikzcd}
N'(N'(A,X),X) \arrow[r, "N'(\varepsilon'_{A,X}, \mathsf{id}_X)"] & N'(A,X) \\
N'(A,X) \arrow[u, "\delta'_{A,X}"] \arrow[r, "\delta'_{A,X}"] \arrow[ur, equal] & N'(N'(A,X),X) \arrow[u, "\varepsilon'_{N'(A,X),X}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
N'(N'(A,X),X) \arrow[r, "N'(\delta'_{A,X}, \mathsf{id}_X)"] & N'(N'(N'(A,X),X),X) \\
N'(A,X) \arrow[u, "\delta'_{A,X}"] \arrow[r, "\delta'_{A,X}"] & N'(N'(A,X),X) \arrow[u, "\delta'_{N'(A,X),X}"']
\end{tikzcd}$$

The bifunctor D^H, corresponding to a functor H, for example, is extended into a bicomonad $(\mathsf{D}^H, \varepsilon', \delta')$ by putting $\varepsilon'_{A,X} = \mathsf{fst}_{A,HX}$, $\delta'_{A,X} = \langle\, \mathsf{id}_{\mathsf{D}^H(A,X)}, \mathsf{snd}_{A,HX} \,\rangle$.

Given a bicomonad $(N', \varepsilon', \delta')$ on $\mathcal{C}$, the inductive functor $\hat{\mu}N'$ and the coinductive functor $\hat{\nu}N'$ based on its underlying bifunctor N' are both extendable into a comonad.

To define the comonad structure on $\hat{\mu}N'$, write N for $\hat{\mu}N'$ and β for $\mathsf{in}_{N'}^{-1} : N \dot{\to} N' \cdot \langle \mathsf{Id}, N \rangle$. The counit $\varepsilon : N \dot{\to} \mathsf{Id}$ of the comonad is manufactured from ε' explicitly by letting $\varepsilon_A = \varepsilon'_{A,NA} \circ \beta_A$. The comultiplication $\delta : N \dot{\to} N \cdot N$ is defined in terms of δ' using primitive recursion: δ_A is the unique morphism h such that

$$\begin{tikzcd}
N'(A, NA) \arrow[r, "N'(\mathsf{id}_A, \langle h, \mathsf{id}_{NA} \rangle)"] \arrow[dddd, "\beta_A^{-1}"'] & N'(A, N(NA) \times NA) \arrow[d, "\delta'_{A, N(NA) \times NA}"] \\
 & N'(N'(A, N(NA) \times NA), N(NA) \times NA) \arrow[d, "N'(N'(\mathsf{id}_A, \mathsf{snd}_{N(NA),NA}), \mathsf{fst}_{N(NA),NA})"] \\
 & N'(N'(A, NA), N(NA)) \arrow[d, "N'(\beta_A^{-1}, \mathsf{id}_{N(NA)})"] \\
 & N'(NA, N(NA)) \arrow[d, "\beta_{NA}^{-1}"] \\
NA \arrow[r, "h"] & N(NA)
\end{tikzcd}$$

To spell out the comonad structure on $\hat{\nu}N'$, write N for $\hat{\nu}N'$ and β for $\mathsf{out}_{N'}$: $N \dot{\to} N' \cdot \langle \mathsf{Id}, N \rangle$. The counit $\varepsilon : N \dot{\to} \mathsf{Id}$ for the comonad on N is defined via ε' explicitly as before: $\varepsilon_A = \varepsilon'_{A,NA} \circ \beta_A$. The comultiplication $\delta N \dot{\to} N \cdot N$ is produced from δ' by means of coiteration: δ_A is specified as the unique h such that

$$\begin{tikzcd}
N'(NA, NA) \arrow[r, "N'(\mathsf{id}_{NA}, h)"] & N'(NA, N(NA)) \\
N'(N'(A, NA), NA) \arrow[u, "N'(\beta_A^{-1}, \mathsf{id}_{NA})"] & \\
N'(A, NA) \arrow[u, "\delta'_{A,NA}"] & \\
NA \arrow[u, "\beta_A"] \arrow[r, "h"] & N(NA) \arrow[uuu, "\beta_{NA}"']
\end{tikzcd}$$

In both cases, the defining characterization of δ is equivalent to this more symmetric charaterization: δ_A is the unique morphism $h : NA \to N(NA)$ such that

$$\begin{tikzcd}
N'(N'(A, NA), NA) \arrow[r, "N'(\mathsf{id}_{N'(A,NA)}, h)"] & N'(N'(A, NA), N(NA)) \\
N'(A, NA) \arrow[u, "\delta'_{A,NA}"] & N'(NA, N(NA)) \arrow[u, "N'(\beta_A, \mathsf{id}_{N(NA)})"'] \\
NA \arrow[u, "\beta_A"] \arrow[r, "h"] & N(NA) \arrow[u, "\beta_{NA}"']
\end{tikzcd}$$

6 Conclusion and future work

We have shown that the fact that the types of trees and cotrees with variables give rise to monads is but a special instance of a more general phenomenon and the same holds about the types of decorated trees and cotrees yielding comonads. Written down as programs, the constructions present an example of seriously generic programming. Potentially, they ought to be useful as building blocks in applications, e.g., in the representation and manipulation of syntax or in the processing of hierarchical data along the lines of "explosive" programming as studied by Oliveira [Oli98]. This is envisaged as future work.

Acknowledgements

The simple but important fact that coterm algebras give rise to monads in the same way as term algebras was first brought to my attention by Peter Hancock. Thanks are also due to Varmo Vene, Ralph Matthes and Felix Joachimski for discussions leading to further insights and to the referees of this paper.

This work received support from the Portuguese Foundation for Science and Technology under grant no. PRAXIS XXI/C/EEI/14172/98 and from the Estonian Science Foundation under grant no. 4155.

The diagrams were typeset using the XYpic macro package by Kristoffer Rose.

References

[AAV01] P. Aczel, J. Adámek, and J. Velebil. A coalgebraic view of infinite trees and iteration. In [CLM01].

[Bar01] F. Bartels. Generalised coinduction. In [CLM01].

[BdM97] R. Bird and O. de Moor. *Algebra of Programming*, vol. 100 of *Prentice Hall Int. Series in Computer Science.* Prentice Hall, 1997.

[BG92] S. Brookes and S. Geva. Computational comonads and intensional semantics. In M. P. Fourman, P. T. Johnstone and A. M. Pitts, eds., *Applications of Categories in Computer Science (Proceedings LMS Symp., Durham, July 1991)*, vol. 177 of *LMS Lecture Note Series*, 1–44. Cambridge Univ. Press, 1992.

[BW84] M. Barr and C. Wells. *Toposes, Triples and Theories*, vol. 278 of *Grundlehren der mathematischen Wissenschaften*, Springer-Verlag, 1984.

[CLM01] A. Corradini, M. Lenisa, and U. Montanari, eds. *Proceedings 4th Workshop on Coalgebraic Methods in Computer Science, CMCS'01 (Genova, Apr. 2001)*, vol. 44(1) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2001.

[Fok92] M. M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Univ. of Twente, 1992.

[G$^+$01] N. Ghani, C. Lüth, F. de Marchi, and J. Power. Algebras, coalgebras, monads and comonads. In [CLM01].

[Kie99] R. Kieburtz. Codata and comonads in Haskell. Unpublished draft, 1999.

[Man76] E. G. Manes. *Algebraic Theories*, vol. 26 of *Graduate Texts in Mathematics*. Springer-Verlag, 1976.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[Mos01] L. S. Moss. Parametric corecursion. *Theoretical Computer Science*, 260(1–2):139–163, 2001.

[Oli98] J. N. Oliveira. "Explosive" programming controlled by calculation. Technical Report UM-DI TR 02/98, Dep. de Informática, Univ. do Minho, Braga, 1998.

[TP97] D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *Proceedings 12th Ann. IEEE Symp. on Logic in Computer Science, LICS'97 (Warsaw, June/July 1997)*, 280–291. IEEE CS Press, 1997.

[Tur96] D. Turi. *Functorial Operational Semantics and Its Denotational Dual.* PhD thesis, Free University, Amsterdam, June 1996.

[UV99] T. Uustalu and V. Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *INFORMATICA*, 10(1):5–26, 1999.

[UV01] T. Uustalu and V. Vene. The dual of substitution is redecoration. In *Draft Proceedings 3rd Scottish Functional Programming Workshop, SFP'01 (Stirling, Aug. 2001)*, 201–211. Dept. of Comp. Sci. and Math., Univ. of Stirling, 2001.

[UVP01] T. Uustalu, V. Vene, and A. Pardo. Recursion schemes from comonads. To appear in *Nordic J. of Computing* Special Issue 12th Nordic Workshop on Programming Theory, NWPT'00 (Bergen, Oct. 2000), 2001.

[Ven00] V. Vene. *Categorical Programming with Inductive and Coinductive Types.* PhD thesis, Univ. of Tartu, 2000.

[Wad92] P. Wadler. The essence of functional programming. In *Conf. Record 19th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'92 (Albuquerque, Jan. 1992)*, 1–12. ACM Press, 1992.

A Implementation in Haskell

The Haskell code below is an implementation the general construction of inductive and coinductive (co)monads from bi(co)monads. Note that extensive use is made of constructors of higher kinds, constructor classes and inheritance.

```
-- Functors

{-
class Functor f where
  fmap :: (a -> b) -> f a -> f b
-}

-- Monads, comonads

class (Functor m) => Mon m where
  unit :: a -> m a
  mult :: m (m a) -> m a

class (Functor n) => Comon n where
  counit :: n a -> a
  comult :: n a -> n (n a)


-- Bifunctors

class BiFunctor f where
  bifmap :: (a -> b) -> (a' -> b') -> f a a' -> f b b'

-- Bimonads, bicomonads

class (BiFunctor m) => BiMon m where
  biunit :: a -> m a a'
  bimult :: m (m a a') a' -> m a a'

class (BiFunctor n) => BiComon n where
  bicounit :: n a a' -> a
  bicomult :: n a a' -> n (n a a') a


-- Pointwise mu, nu type constructors

data Mu f a = In (f a (Mu f a))

unIn :: Mu f a  -> f a (Mu f a)
unIn (In x) = x

cata :: (BiFunctor f) => (f a c -> c) -> Mu f a -> c
cata phi = phi . bifmap id (cata phi) . unIn

para :: (BiFunctor f) => (f a (c, Mu f a) -> c) -> Mu f a -> c
para phi = phi . bifmap id (both (para phi) id) . unIn

both :: (c -> a) -> (c -> a') -> c -> (a, a')
both g g' x = (g x, g' x)
```

```
data Nu f a = UnOut (f a (Nu f a))

out :: Nu f a -> f a (Nu f a)
out (UnOut x) = x

ana :: (BiFunctor f) => (c -> f a c) -> c -> Nu f a
ana phi = UnOut . bifmap id (ana phi) . phi

apo :: (BiFunctor f) => (c -> f a (Either c (Nu f a))) -> c -> Nu f a
apo phi = UnOut . bifmap id (either (apo phi) id) . phi

{-
either :: (a -> c) -> (a' -> c) -> Either a a' -> c
either g g' (Left  x) = g  x
either g g' (Right x) = g' x
-}

-- Pointwise mu, nu functors

instance (BiFunctor f) => Functor (Mu f) where
  fmap g = cata (In . bifmap g id)

instance (BiFunctor f) => Functor (Nu f) where
  fmap g = ana (bifmap g id . out)


-- Pointwise mu, nu monads

instance (BiMon m) => Mon (Mu m) where
  unit = In . biunit
  mult = cata (In . bimult . bifmap unIn id)

instance (BiMon m) => Mon (Nu m) where
  unit = UnOut . biunit
  mult = apo (bimult . bifmap (bifmap id Right) Left . bifmap out id . out)

-- Pointwise mu, nu comonads

instance (BiComon n) => Comon (Mu n) where
  counit = bicounit . unIn
  comult = para (In . bifmap In id . bifmap (bifmap id snd) fst . bicomult)

instance (BiComon n) => Comon (Nu n) where
  counit = bicounit . out
  comult = ana (bifmap UnOut id . bicomult . out)
```