Chapter 9

The Dual of Substitution is Redecoration

Tarmo Uustalu[1] and Varmo Vene[2]

Abstract: It is well known that type constructors of *incomplete trees* (trees with variables) carry the structure of a monad with *substitution* as the extension operation. Less known are the facts that the same is true of type constructors of *incomplete cotrees* (=non-wellfounded trees) and that the corresponding monads exhibit a special structure. We wish to draw attention to the dual facts which are as meaningful for functional programming: type constructors of *decorated cotrees* carry the structure of a comonad with *redecoration* as the coextension operation, and so do—even more interestingly—type constructors of *decorated trees*.

9.1 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. A good example is given in the categorical notions of monad and comonad: these are in several programming contexts useful as means for 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] also in programming as a means to set up an infrastructure for representing and manipulating computations with effects. 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

[1] Dep. de Informática, Univ. do Minho, Campus de Gualtar, P-4710-057 Braga, Portugal [on leave from Inst. of Cybernetics, Tallinn Techn. Univ., Akadeemia tee 21, EE-12618 Tallinn, Estonia]; Email: `tarmo@cs.ioc.ee`.

[2] Abt. Informatik, Univ. Trier, D-54286 Trier, Germany [on leave from Inst. of Comp. Sci., Univ. of Tartu, J. Liivi 2, EE-50409 Tartu, Estonia]; Email: `varmo@cs.ut.ee`.

framework for encapsulating effects of computations in context. Comonads and monads equipped with distributive laws can be used to specify complex recursion and corecursion schemes for inductive and coinductive types [UVP01, Bar01].

In this paper, we describe monad and comonad structures on certain type constructors of inductive and coinductive data. It is well known that type constructors of incomplete trees (trees with “variables”, or “mutable leaves”, for which trees can be substituted) carry the structure of a monad with substitution as the extension operator; this fact is the starting point of the categorical approach to universal algebra. The equally plausible, but not so apparent fact that the same holds of type constructors of incomplete cotrees (=non-wellfounded trees) has been pointed out only recently [AAV01, Mos01] (cf. also [G^+01]). We show that these two facts dualize for type constructors of what we call decorated trees and decorated cotrees: these determine comonads with redecoration as the coextension operator. The pragmatic implication is that a good deal of programming and reasoning about programs involving (co)trees, substitution, and redecoration can be done generically, once and for all. This is potentially very useful as these things are ubiquitous in programming: the simplest examples of substitution operations are addition for natural numbers, append for lists, substitution for variables in a term; functions marking the nodes of a tree with some information pertaining to the subtrees they root are redecoration functions. In a related paper, [Uus01], we generalize the monad and comonad constructions on (co)tree type constructors for a wider class of (co)inductive type constructors.

The organization of the paper is the following. In Sections 9.2, 9.3, we define the concepts of monad and comonad and list some simplest popular examples. In Section 9.4, we give the monad constructions on type constructors of incomplete trees and cotrees. These are dualized in Section 9.5 into comonad constructions on type constructors of decorated cotrees and trees. In Section 9.6, we conclude and mention some directions for future work.

Though the paper is motivated by issues in functional programming, the material is basically category-theoretic. We have striven for elementary exposition. The reader is assumed to know the basics of the categorical approach to functional programming (the types-as-objects, functions-as-morphisms identification), in particular, the central concepts of coproduct, product, and exponential objects (sum, product, and function-space types), initial algebras and final coalgebras (inductive and coinductive types). The more specific concepts of 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 to [Fok92, BdM97]. The recursion and corecursion schemes used in the paper are described in [UV99, UVP01]. The classic category-theory texts treating (co)monads are [Man76, BW84].

Throughout the paper, we work in one base category $\mathcal{C}$ about which we do not make any specific assumptions other than the existence of the particular coproducts, initial algebras etc. that we name. The category **Set** of sets and set-theoretic functions is always a possible choice for $\mathcal{C}$. The notation used is standard. [In particular, $|\mathcal{C}|$ is the collection of the objects of $\mathcal{C}$ and $\mathbf{End}(\mathcal{C})$ denotes the cate-

gory of endofunctors on $\mathcal{C}$; $\langle f, g \rangle$ is the notation for the pairing of f and g, $[f, g]$

is the case analysis of f and g; $\mathsf{ev}_{A,B}$ is application: $\mathsf{ev}_{A,B} : (A \Rightarrow B) \times A \to B$.]

9.2 MONADS

A monad is a structure on an object mapping. The structure of a monad is describable in several equivalent ways. For our purposes, it is most natural to work with the so-called extension form (Kleisli triples).

Definition 9.1. *A* monad (in extension form) *on $\mathcal{C}$ is a triple $(M, \eta, -^\star)$ consisting of an endomapping M on $|\mathcal{C}|$ (underlying object mapping), a $|\mathcal{C}|$-indexed family η of morphisms $\eta_A : A \to MA$* (unit)*, and an operation $-^\star$ taking every morphism $f : A \to MB$ in $\mathcal{C}$ to a morphism $f^\star : MA \to MB$* (extension operation) *such that*

$$f^\star \circ \eta_A = f \quad \textit{(for } f : A \to MB\textit{)} \tag{9.1}$$

$$\eta_A{}^\star = \mathsf{id}_{MA} \tag{9.2}$$

$$(g^\star \circ f)^\star = g^\star \circ f^\star \tag{9.3}$$

[The unit and the extension operation are called "return" and "bind" in Haskell.] Below are some simple examples of monad constructions popular in functional programming, where monads are best known as tools for encapsulating effects of computations: the exception, storage reading and state transformation monads.

Example 9.2. Given an object E in $\mathcal{C}$, the endomapping M on $|\mathcal{C}|$ defined by $MA = A + E$ carries a monad structure defined by $\eta_A = \mathsf{inl}_{A,E}$ and $f^\star = [\, f, \mathsf{inr}_{B,E} \,]$ for $f : A \to MB$.

Example 9.3. Given an object S in $\mathcal{C}$, the endomapping M on $|\mathcal{C}|$ defined by $MA = S \Rightarrow A$ carries a monad structure defined by $\eta_A = \mathsf{curry}(\mathsf{fst}_{A,S})$, $f^\star = \mathsf{curry}(\mathsf{ev}_{S,B} \circ \langle f \circ \mathsf{ev}_{S,A}, \mathsf{snd}_{MA,S} \rangle)$ for $f : A \to MB$.

Example 9.4. Given an object S in $\mathcal{C}$, the endomapping M on $|\mathcal{C}|$ defined by $MA = S \Rightarrow (A \times S)$ carries a monad structure defined by $\eta_A = \mathsf{curry}(\mathsf{id}_{A\times S})$ and $f^\star = \mathsf{curry}(\mathsf{ev}_{S,B\times S} \circ (f \times \mathsf{id}_S) \circ \mathsf{ev}_{S,A\times S})$ for $f : A \to MB$.

9.3 COMONADS

Comonads are the formal dual of monads. Below is a definition for the coextension form (coKleisli triples).

Definition 9.5. *A* comonad (in coextension form) *on $\mathcal{C}$ is a triple $(N, \varepsilon, -^\dagger)$ consisting of an endomapping N on $|\mathcal{C}|$, a $|\mathcal{C}|$-indexed family of morphisms $\varepsilon_A : NA \to A$* (counit)*, and an operation $-^\dagger$ taking every morphism $f : NA \to B$ to a morphism $f^\dagger : NA \to NB$* (coextension operation) *such that*

$$\varepsilon_B \circ f^\dagger = f \quad (\text{for } f : NA \to B) \tag{9.4}$$

$$\varepsilon_A{}^\dagger = \mathsf{id}_{NA} \tag{9.5}$$

$$(g \circ f^\dagger)^\dagger = g^\dagger \circ f^\dagger \tag{9.6}$$

Comonads, too, can be used for encapsulation of effects. Differently from monads, however, comonads handle incoming (externally produced) rather than outgoing (internally produced) effects. The simplest examples are the storage reading and "state in context" comonads. Storage reading comonads are the same as storage reading monads, modulo currying/uncurrying.

Example 9.6. Given an object S on $\mathcal{C}$, the endomapping N on $|\mathcal{C}|$ defined by $NA = A \times S$ carries a comonad structure defined by $\varepsilon_A = \mathsf{fst}_{A,S}$, $f^\dagger = \langle f, \mathsf{snd}_{A,S} \rangle$ for $f : NA \to B$.

Example 9.7. Given an object S on $\mathcal{C}$, the endomapping N on $|\mathcal{C}|$ defined by $NA = (S \Rightarrow A) \times S$ carries a comonad structure defined by $(N, \varepsilon, -^\dagger)$ by setting $\varepsilon_A = \mathsf{ev}_{S,A}$, $f^\dagger = \mathsf{curry}(f) \times \mathsf{id}_S$ for $f : NA \to B$.

9.4 MONADS OF TREES AND COTREES

9.4.1 Monads of trees

The starting point for the categorical approach to universal algebra is the monad construction on type constructors of what we call incomplete trees (trees with variables).

Given an endofunctor H on $\mathcal{C}$. Define a mapping $- + H$ from $|\mathcal{C}|$ to $|\mathbf{End}(\mathcal{C})|$ by $(A+H)X = A + HX$, $(A+H)\xi = \mathsf{id}_A + H\xi$. Define an endomapping M on $|\mathcal{C}|$ and two $|\mathcal{C}|$-indexed families η, τ of morphisms $\eta_A : A \to MA$, $\tau_A : HMA \to MA$ from the initial $(A+H)$-algebras $(\mu(A+H), \mathsf{in}_{A+H})$ by

$$MA = \mu(A+H) \quad \wedge \quad \eta_A = \mathsf{in}_{A+H} \circ \mathsf{inl}_{A,HMA} \quad \wedge \quad \tau_A = \mathsf{in}_{A+H} \circ \mathsf{inr}_{A,HMA} \tag{9.7}$$

If $\mathcal{C}$ is a category of types and functions, then the object MA given by an object A is the type collecting the *A-incomplete H-trees*, i.e., H-branching trees involving variables drawn from the type A. The morphisms η_A, τ_A are the two construction functions of this type: they produce an A-incomplete H-tree from either a variable from A or an H-structure of A-incomplete H-trees. To give an example, if $HX = 1 + X \times X$, then MA is the type of binary A-incomplete trees. Category theorists speak of (MA, η_A, τ_A) as the *free H-algebra over A* (notice that (MA, τ_A) is an H-algebra!), usefully generalizing the corresponding concept from universal algebra. Universal algebra treats situations where $\mathcal{C}$ is **Set** and the functor H is polynomial. Then H encodes a signature and the A-incomplete H-trees are nothing else than (the syntax-tree presentations of) the *terms over A* in this signature. In the case of $HX = 1 + X \times X$, we might say that MA is the set of terms over A in a signature with one nullary operator (constant) and one binary operator. To get, e.g., arithmetical expressions over A built of numerals and two operators addition and multiplication, we should use $HX = \mathsf{Nat} + 2 \times X \times X$.

Now, a significant fact is that we can also canonically produce a morphism $f^\star : MA \to MB$ from any given morphism $f : A \to MB$. This morphism is defined as an iteration (=catamorphism, =fold)

$$f^\star = (\!|\, [\, f, \tau_B \,] \,|\!)_{A+H} \tag{9.8}$$

so that $f^\star$ is the unique morphism $h : MA \to MB$ satisfying

$$h \circ [\eta_A, \tau_A] = [f, \tau_B] \circ (\mathsf{id}_A + Hh) \tag{9.9}$$

$$\begin{tikzcd}
A + HMA \arrow[r, "{[\eta_A,\tau_A]}"] \arrow[d, "\mathsf{id}_A+Hh"'] & MA \arrow[d, "h"] \\
A + HMB \arrow[r, "{[f,\tau_B]}"] & MB
\end{tikzcd}$$

or, equivalently,

$$h \circ \eta_A = f \quad \wedge \quad h \circ \tau_A = \tau_B \circ Hh$$

$$\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[dr, "f"'] & MA \arrow[d, "h"] & HMA \arrow[l, "\tau_A"'] \arrow[d, "Hh"] \\
& MB & HMB \arrow[l, "\tau_B"]
\end{tikzcd}$$

From the functional programming point-of-view, $f^\star$ is a function that takes an A-incomplete H-tree and replaces its variables (these are drawn from A) with B-incomplete H-trees, relying on f as a guideline, and leaves the rest the same (notice that $f^\star$ is a H-algebra homomorphism from (MA, τ_A) to (MB, τ_B)!). In short, $f^\star$ is the *substitution function* corresponding to f seen as a *substitution rule*. The triple $(M,\eta,-^\star)$ turns out to be a monad, with the monad laws following solely from the fact that $f^\star$ is a unique solution of Eq. 9.9 (the fact that $(MA,[\eta_A,\tau_A])$ is initial among the $(A+H)$-algebras is not needed!). Summing up, we have the following propositions.

Proposition 9.8. *Given an endofunctor H on $\mathcal{C}$. Given also an endomapping M on $|\mathcal{C}|$, two $|\mathcal{C}|$-indexed families η, τ of morphisms $\eta_A : A \to MA$, $\tau_A : HMA \to MA$, and an operation $-^\star$ taking every morphism $f : A \to MB$ into a morphism $f^\star : MA \to MB$ that uniquely solves 9.9 wrt. h. Then $(M,\eta,-^\star)$ is a monad.*

Proposition 9.9. *Given an endofunctor H on $\mathcal{C}$. Then M, η, τ, $-^\star$ defined by Eq.s 9.7, 9.8 satisfy the assumptions of Prop. 9.8 and hence $(M,\eta,-^\star)$ is a monad.*

9.4.2 Monads of cotrees

Knowing that type constructors of incomplete trees carry a monad structure with substitution as the extension operation, it is meaningful to ask whether the same holds of type constructors of incomplete cotrees (=non-wellfounded incomplete trees). Intuition suggests that such type constructors ought to support a substitution operation and then a monad ought to be at hand. This is indeed so.

Given an endofunctor H, use the final $(A+H)$-coalgebras $(\nu(A+H),\mathsf{out}_{A+H})$ to define an endomapping M on $|\mathcal{C}|$ and two $|\mathcal{C}|$-indexed families η, τ of morphisms $\eta_A : A \to MA$, $\tau_A : HMA \to MA$ by

$$MA = \nu(A+H) \quad\wedge\quad \eta_A = \mathsf{out}_{A+H}^{-1} \circ \mathsf{inl}_{A,HMA} \quad\wedge\quad \tau_A = \mathsf{out}_{A+H}^{-1} \circ \mathsf{inr}_{A,HMA}$$

(9.10)

employing the fact that out_{A+H} is iso (Lambek's lemma) and, hence, $\nu(A+H)$ carries a canonical $(A+H)$-algebra structure out^{-1}_{A+H}. In the programming interpretation, the object MA induced by an object A is the type of all H-branching A-incomplete cotrees (=non-wellfounded trees) and the morphisms η_A and τ_A are its two constructor functions, η_A producing an A-incomplete cotree from an inhabitant of A and τ_A making one from an H-structure of A-incomplete cotrees. In case $HX = 1 + X \times X$, MA collects the binary cotrees with variables from A.

It remains to define a morphism $f^\star : MA \to MB$ for any given morphism $f : A \to MB$. This can be done using primitive corecursion (= apomorphism):

$$f^\star = [\![(*) \circ (([\eta_B, \tau_B]^{-1} \circ f) + \mathsf{id}_{HMA}) \circ [\eta_A, \tau_A]^{-1}]\!]_{B+H} \qquad (9.11)$$

where $(*) = [\mathsf{id}_B + H\mathsf{inr}_{MA,MB}, \mathsf{inr}_{B,H(MA+MB)} \circ H\mathsf{inl}_{MA,MB}]$, so that $f^\star$ is the unique morphism $h : MA \to MB$ satisfying

$$[\eta_B, \tau_B]^{-1} \circ h = (\mathsf{id}_B + H[h, \mathsf{id}_{MB}]) \circ (*) \circ (([\eta_B, \tau_B]^{-1} \circ f) + \mathsf{id}_{HMA}) \circ [\eta_A, \tau_A]^{-1}$$

$$B + H(MA + MB) \xleftarrow{(*)} (B + HMB) + HMA \xleftarrow{([\eta_B,\tau_B]^{-1} \circ f) + \mathsf{id}_{HMA}} A + HMA \xleftarrow{[\eta_A,\tau_A]^{-1}} MA$$

$$\begin{tikzcd} {} \arrow[d, "\mathsf{id}_B + H[h, \mathsf{id}_{MB}]"] & & {} \arrow[d, "h"] \\ B + HMB & & MB \arrow[ll, "{[\eta_B, \tau_B]^{-1}}"'] \end{tikzcd}$$

A simplification of the last equation using properties of coproduct and the fact that $[\eta_A, \tau_A]$ is iso gives Eq. 9.9. Thus, $-^\star$ is a substitution operation again, but now for incomplete cotrees, not trees as previously. Prop. 9.8 applies and establishes that $(M, \eta, -^\star)$ is a monad.

Proposition 9.10. *Given an endofunctor H on $\mathcal{C}$. Then M, η, τ, $-^\star$ defined by Eq.s 9.10, 9.11 meet the assumptions of Prop. 9.8 and hence $(M, \eta, -^\star)$ is a monad.*

There is more to type constructors of incomplete cotrees: they are not just monads, but *"iterative"* monads, in the following sense. For any morphism $g : A \to M(A+B)$ such that

$$g = [\eta_{A+B} \circ \mathsf{inr}_{A,B}, \tau_{A+B}] \circ \varphi \tag{9.12}$$

for some morphism $\varphi : A \to B + HM(A+B)$, we can canonically point out a morphism $g^{\mathrm{it}} : A \to MB$. This morphism is smoothly definable as an application of an instance of the generic monad-controlled corecursion scheme of [UVP01] so that g^{it} is the unique morphism $f : A \to MB$ satisfying the equation

$$[\eta_B, \tau_B]^{-1} \circ f = (\mathsf{id}_B + H f^\sharp) \circ \varphi$$

$$\begin{tikzcd}
B+HM(A+B) \arrow[d, "\mathsf{id}_B+Hf^\sharp"'] & & A \arrow[ll, "\varphi"'] \arrow[d, "f"] \\
B+HMB & & MB \arrow[ll, "{[\eta_B,\tau_B]^{-1}}"']
\end{tikzcd}$$

where $f^\sharp = [\,f,\eta_B]^\star : M(A+B) \to MB$. This equation simplifies to the equation

$$f = f^\sharp \circ g \tag{9.13}$$

which is an “iteration” equation for substitutions rules (the word “iteration” refers here to tail-recursion, not to a scheme of structured recursion for inductive types). In programming terms, g^{it} is the substitution rule assigning B-incomplete H-cotrees to variables from A which is obtained by repeating g where g is seen as a substitution rule assigning $(A+B)$-incomplete H-cotrees to variables from A. Such repetition is well-defined provided that g is productive in the sense of not replacing a variable from A with another variable from A: it may only replace it with a variable from B or an H-structure of $(A+B)$-incomplete H-cotrees.

Proposition 9.11. *Given an endofunctor H on $\mathcal{C}$. Then the monad of Prop. 9.10 carries an “iteration” operation $-^{\mathrm{it}}$ taking every morphism $g : A \to M(A+B)$ satisfying Eq. 9.12 into a morphism $g^{\mathrm{it}} : A \to MB$ that uniquely solves Eq. 9.13 wrt. f.*

9.4.3 General monads and substitution

As matter of fact, every monad M on **Set** has something to do with substitution in the usual sense of term substitution: MA may be understood as the free (i.e., term-equivalence-class) algebra over A for some (possibly infinitary) signature and collection of equations [Man76, Sec. 1.5]. This and further related theory, however, remains outside of the scope of this paper. A practical conclusion is that a monad may always be thought of as a type constructor endowed with a substitution-like operation. The monad laws state basic properties of substitution. One might say they are abstractions of the facts $x[r/x] = r$, $t[x/x] = t$, and $(t[r/x])[s/y] = t[r[s/y]/x]$ provided $y \notin \mathrm{FV}(t)$. Many useful properties of substitution follow from these laws alone. It is easy to prove, for example, an abstract version of the well-known lemma stating that $(t[r/x])[s/y] = t[r[s/y]/x, s/y] = (t[s/y])[r[s/y]/x]$, provided $x \notin \mathrm{FV}(s)$. For $f : A \to M(B+C)$ and $g : B \to MC$,

$$\begin{tikzcd}
M(A+(B+C)) \arrow[r, "f^\sharp"] \arrow[d, shift right] & M(B+C) \arrow[r, "g^\sharp"] & MC \arrow[d, equal] \\
M((A+B)+C) \arrow[rr, "{[g^\sharp \circ f, g]^\sharp}"] \arrow[u, shift right] \arrow[d, shift right] & & MC \arrow[d, equal] \\
M(B+(A+C)) \arrow[r, "(M\mathsf{inr}_{A,C} \circ g)^\sharp"] \arrow[u, shift right] & M(A+C) \arrow[r, "(g^\sharp \circ f)^\sharp"] & MC
\end{tikzcd}$$

where $-^\sharp$ is defined as above. Such laws are potentially useful in program transformation.

9.5 COMONADS OF (CO)TREES

In the preceding section, we have seen two monad constructions with clear relevance for functional programming. Not surprisingly, they have duals which de-

liver comonads, but a noteworthy fact is that the dual constructions also exhibit a functional programming reading.

9.5.1 Comonads of cotrees

Given an endofunctor H on $\mathcal{C}$, we now begin by defining a mapping $- \times H$ from $|\mathcal{C}|$ to $|\mathbf{End}(\mathcal{C})|$ by $(A \times H)X = A \times HX$, $(A \times H)\xi = \mathsf{id}_A \times H\xi$. For the dual of the first monad construction, we define an endomapping N on $|\mathcal{C}|$ and two $|\mathcal{C}|$-indexed families ε, θ of morphisms $\varepsilon_A : NA \to A$, $\theta_A : NA \to HNA$ from the final $(A \times H)$-coalgebras $(\nu(A \times H), \mathsf{out}_{A \times H})$ by

$$NA = \nu(A \times H) \quad \wedge \quad \varepsilon_A = \mathsf{fst}_{A,HNA} \circ \mathsf{out}_{A \times H} \quad \wedge \quad \theta_A = \mathsf{snd}_{A,HNA} \circ \mathsf{out}_{A \times H} \tag{9.14}$$

In functional programming terms, the object NA induced by an object A is usefully thought of as the type of all *A-decorated H-cotrees*, i.e., H-cotrees with every node (the root of every subcotree) paired with a decoration from the type A. The morphisms ε_A, θ_A are the two associating destruction functions; they analyze an A-decorated H-cotree into a decoration from A and an H-structure of A-decorated H-cotrees. In the special case $HX = X$, NA is the type of streams with element type A and ε_A, θ_A are the head and tail functions. If $HX = 1 + X \times X$, NA is the type of A-decorated binary cotrees.

We also define, for any morphism $f : NA \to B$, a morphism $f^\dagger : NA \to NB$ as a coiteration (= anamorphism, =unfold) by

$$f^\dagger = [\![\langle f, \theta_A \rangle]\!]_{B \times H} \tag{9.15}$$

with the effect that $f^\dagger$ is the unique morphism $h : NA \to NB$ satisfying

$$\langle \varepsilon_B, \theta_B \rangle \circ h = (\mathsf{id}_B \times Hh) \circ \langle f, \theta_A \rangle \tag{9.16}$$

$$\begin{tikzcd}
NA \arrow[r, "\langle f,\theta_A \rangle"] \arrow[d, "h"'] & B \times HNA \arrow[d, "\mathsf{id}_B \times Hh"] \\
NB \arrow[r, "\langle \varepsilon_B,\theta_B \rangle"] & B \times HNB
\end{tikzcd}$$

or, equivalently,

$$\varepsilon_B \circ h = f \quad \wedge \quad \theta_B \circ h = Hh \circ \theta_A$$

$$NA \xrightarrow{\theta_A} HNA$$

$$\begin{tikzcd}
 & {} \arrow[dl, "f"'] \arrow[d, "h"] & {} \arrow[d, "Hh"] \\
B & NB \arrow[l, "\varepsilon_B"'] \arrow[r, "\theta_B"] & HNB
\end{tikzcd}$$

Then, in the functional programming interpretation, $f^\dagger$ is a function that takes an A-decorated H-cotree and replaces the decoration of the root of its every subcotree (these are all A-decorated H-cotrees) with a decoration from B using f as

a guideline; everything else remains untouched ($f^\dagger$ is an H-coalgebra homomorphism from (NA, θ_A) to (NB, θ_B)). In short, the morphism $f^\dagger$ is the *redecoration function* corresponding to f as a *redecoration rule*. (Gibbons [Gib93] calls $f^\dagger$ the *upwards pass* given by f.) In the case of streams, $f : NA \to A^2$ could be the function that extracts the 1st and 2nd elements of a stream, $f^\dagger : NA \to NA^2$ is then the function that takes a stream and returns the stream of its adjacent element pairs. The triple $(N, \varepsilon, -^\dagger)$ is a comonad; moreover, the comonad laws follow just from the fact that $f^\dagger$ is a unique solution of Eq. 9.16. Thus we have:

Proposition 9.12. *Given an endofunctor H on $\mathcal{C}$. Given also an endomapping N on $|\mathcal{C}|$, two $|\mathcal{C}|$-indexed families ε, θ of morphisms $\varepsilon_A : NA \to A$, $\theta_A : NA \to HNA$, and an operation $-^\dagger$ taking every morphism $f : NA \to B$ into a morphism $f^\dagger : NA \to NB$ that uniquely solves Eq. 9.16 wrt. h. Then $(N, \varepsilon, -^\dagger)$ is a comonad.*

Proposition 9.13. *Given an endofunctor H on $\mathcal{C}$. Then N, ε, θ, $-^{\dagger}$ defined by Eq.s 9.14, 9.15 satisfy the assumptions of Prop. 9.12 and hence $(N, \varepsilon, -^{\dagger})$ is a comonad.*

9.5.2 Comonads of trees

For the dual of the second monad construction, we use the initial $(A \times H)$-algebras $(\mu(A \times H), \mathsf{in}_{A \times H})$ and define an endomapping N on $|\mathcal{C}|$ and two $|\mathcal{C}|$-indexed families ε, θ of morphisms $\varepsilon_A : NA \to A$, $\theta_A : NA \to HNA$ by

$$NA = \mu(A \times H) \quad \wedge \quad \varepsilon_A = \mathsf{fst}_{A,HNA} \circ \mathsf{in}_{A \times H}^{-1} \quad \wedge \quad \theta_A = \mathsf{snd}_{A,HNA} \circ \mathsf{in}_{A \times H}^{-1} \qquad (9.17)$$

Then, the object NA corresponding to an object A models the type of A-*decorated* H-*trees* and ε_A, θ_A are the two accompanying functions for destructing an A-decorated H-tree into a decoration in A and an H-structure of A-decorated H-trees. If $HX = 1 + X \times X$, NA is the type of A-decorated binary trees.

For any given morphism $f : NA \to B$, define a morphism $f^{\dagger} : NA \to NB$ as a primitive recursion (=paramorphism) by

$$f^{\dagger} = \langle\!| \langle \varepsilon_B, \theta_B \rangle^{-1} \circ ((f \circ \langle \varepsilon_A, \theta_A \rangle^{-1}) \times \mathsf{id}_{HNB}) \circ (*) |\!\rangle_{A \times H} \qquad (9.18)$$

where $(*) = \langle \mathsf{id}_A \times H\mathsf{snd}_{NB,NA}, H\mathsf{fst}_{NB,NA} \circ \mathsf{snd}_{A,H(NB \times NA)} \rangle$, meaning that $f^{\dagger}$ is the unique morphism $h : NA \to NB$ such that

$$h \circ \langle \varepsilon_A, \theta_A \rangle^{-1} = \langle \varepsilon_B, \theta_B \rangle^{-1} \circ ((f \circ \langle \varepsilon_A, \theta_A \rangle^{-1}) \times \mathsf{id}_{HNB}) \circ (*) \circ (\mathsf{id}_A \times H\langle h, \mathsf{id}_{NA} \rangle)$$

$$\begin{tikzcd}
NA \arrow[d, "h"] & & & A \times HNA \arrow[lll, "\langle \varepsilon_A, \theta_A \rangle^{-1}"'] \arrow[d, "\mathsf{id}_A \times H\langle h, \mathsf{id}_{NA} \rangle"] \\
NB & B \times HNB \arrow[l, "\langle \varepsilon_B, \theta_B \rangle^{-1}"'] & (A \times HNA) \times HNB \arrow[l, "(f \circ \langle \varepsilon_A, \theta_A \rangle^{-1}) \times \mathsf{id}_{HNB}"'] & A \times H(NB \times NA) \arrow[l, "(*)"']
\end{tikzcd}$$

This equation simplifies into Eq. 9.16. Thus, $-^{\dagger}$ is a redecoration or upwards pass operation again, although this time for decorated trees, not cotrees. In the

case of binary trees, we could have, e.g., $f : NA \to \mathsf{Nat}$ be the function returning the height of a given A-decorated tree. Then $f^{\dagger}$ is the function that takes an A-decorated tree and returns a tree where the A-decoration of each node has been replaced by the height of the subtree rooted by this node. Applying Prop. 9.12, we have that $(N, \varepsilon, -^{\dagger})$ is a comonad.

Proposition 9.14. *Given an endofunctor H on $\mathcal{C}$. Then N, ε, θ, $-^{\dagger}$ defined by Eq.s 9.17, 9.18 meet the assumptions of Prop. 9.12 and hence $(N, \varepsilon, -^{\dagger})$ is a comonad.*

While monads of incomplete cotrees are "iterative", comonads of decorated trees are "*recursive*". For any morphism $g : N(B \times A) \to B$ satisfying

$$g = \varphi \circ \langle \mathsf{snd}_{B,A} \circ \varepsilon_{B\times A}, \theta_{B\times A} \rangle \tag{9.19}$$

for some morphism $\varphi : A \times HN(B\times A) \to B$, we can define a morphism $g^{\mathrm{rec}} : NA \to B$ as an application of an instance of the generic comonad-controlled recursion scheme of [UVP01] so that g^{rec} is the unique morphism $f : NA \to B$ satisfying the equation

$$f \circ \langle \varepsilon_A, \theta_A \rangle^{-1} = \varphi \circ (\mathsf{id}_A \times H f^{\flat})$$

$$\begin{tikzcd}
NA \arrow[d, "f"'] & A \times HNA \arrow[l, "\langle \varepsilon_A, \theta_A \rangle^{-1}"'] \arrow[d, "\mathsf{id}_A \times H f^{\flat}"] \\
B & A \times HN(B\times A) \arrow[l, "\varphi"']
\end{tikzcd}$$

where $f^{\flat} = \langle f, \varepsilon_A \rangle^{\dagger} : NA \to N(B\times A)$. This equation simplifies to the equation

$$f = g \circ f^{\flat} \tag{9.20}$$

a "recursion" equation for decoration rules (the word "recursion" referring to the dual of tail-recursion). In programming terms, g^{rec} is the redecoration rule assigning decorations from B to A-decorated H-trees that results from repeating g where g is seen as a redecoration rule assigning decorations from B to $(B\times A)$-decorated

H-trees. Such repetition is well-defined provided that f never uses the decoration from B of a given $(B \times A)$-decorated H-tree, only looking at the decoration from A and the H-structure of subordinate $(B \times A)$-decorated H-trees. In the case of binary trees, we could think, e.g., of a function $g : N(\mathsf{Nat} \times A) \to \mathsf{Nat}$ which takes a binary tree and returns 0, if the root is a leaf, and 1 plus the maximum of the Nat-decorations of its two constituent subtrees, if it is a branching node. $g^{\mathrm{rec}} : NA \to \mathsf{Nat}$ then calculates the height of a given binary tree. More generally, it can be remarked that redecoration functions corresponding to "recursively" defined redecoration rules are a generalization of Bird and Gibbons' *upwards accumulations* [Gib93, BdMH96], i.e., upwards passes induced by catamorphisms: if there is a morphism $\varphi' : A \times HB \to B$ such that $\varphi = \varphi' \circ (\mathsf{id}_A \times H(\mathsf{fst}_{B,A} \circ \varepsilon_{B \times A}))$,

then $g^{\mathrm{rec}} = (\!|\, \varphi' \,|\!)_{A \times H}$. The *upwards accumulation lemma* saying that $(\!|\, \varphi' \,|\!)_{A \times H}{}^{\dagger} = (\!|\, \langle \varepsilon_B, \theta_B \rangle^{-1} \circ \langle \varphi' \circ (\mathsf{id}_A \times H\varepsilon_B), \mathsf{snd}_{A,HNB} \rangle \,|\!)_{A \times H}$ follows from the more general truth that $(g^{\mathrm{rec}})^{\flat} = (\!|\, \langle \varepsilon_{B \times A}, \theta_{B \times A} \rangle^{-1} \circ \langle \langle \varphi, \mathsf{fst}_{A,HN(B \times A)} \rangle, \mathsf{snd}_{A,HN(B \times A)} \rangle \,|\!)_{A \times H}$.

Proposition 9.15. *Given an endofunctor H on $\mathcal{C}$. Then the comonad of Prop. 9.14 carries an "recursion" operation $-^{\mathrm{rec}}$ taking every morphism $g : N(B \times A) \to B$ satisfying Eq. 9.19 into a morphism $g^{\mathrm{rec}} : NA \to B$ that uniquely solves Eq. 9.20 wrt. f.*

9.5.3 General comonads and redecoration

Just as every monad at an abstract enough level is about incomplete trees and substitution, every comonad is about decorated trees and redecoration. The comonad laws state basic properties of redecoration-like operations and a number of further properties are simple corollaries. Among them is, e.g., the following lemma on composing redecoration functions. For $f : NA \to B$ and $g : N(B \times A) \to C$,

$$\begin{tikzcd}
NA \arrow[r, "f^\flat"] \arrow[d, equal] & N(B \times A) \arrow[r, "g^\flat"] & N(C \times (B \times A)) \arrow[d, shift right] \\
NA \arrow[rr, "\langle g \circ f^\flat, f \rangle^\flat"] \arrow[d, equal] & & N((C \times B) \times A) \arrow[u, shift right] \arrow[d, shift right] \\
NA \arrow[r, "(g \circ f^\flat)^\flat"] & N(C \times A) \arrow[r, "(f \circ N\mathsf{snd}_{C,A})^\flat"] & N(B \times (C \times A)) \arrow[u, shift right]
\end{tikzcd}$$

where $-^\flat$ is defined as above.

9.6 CONCLUSION AND FUTURE WORK

We have shown that type constructors of incomplete (co)trees carry the structure of a monad and type constructors of decorated (co)trees carry the structure of a comonad. The constructions presented make generic use of different generic recursion and corecursion schemes. We expect them to be useful as building blocks in applications such as representation and manipulation of syntax, processing of hierarchical data along the lines of “explosive” programming as studied in [Oli98] or computing with attribute grammars.

ACKNOWLEDGEMENTS

T. Uustalu is grateful to P. Hancock for first pointing it out to him that not only term algebras, but also coterm algebras yield monads. He is also indebted to Graduiertenkolleg “Logik in der Informatik” for an invitation to LMU Munich, where this work was first presented at the “Rekursion für Koalgebren” seminar in May 2001. Three referees helped us with constructive criticism.

This research was supported by the Portuguese Foundation for Science and Technology under grant No. PRAXIS XXI/C/EEI/14172/98 and by the Estonian Science Foundation under grant No. 4155.

REFERENCES

[AAV01] P. Aczel, J. Adámek, and J. Velebil. A Coalgebraic View of Infinite Trees and Iteration. In [CLM01].

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

[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.

[BdMH96] R. Bird, O. de Moor, and P. Hoogendijk. Generic Programming with Types and Relations. *J. of Functional Programming*, 6(1):1–21, 1996.

[BG92] S. Brookes and S. Geva. Computational Comonads and Intensional Semantics. In M. P. Fourman et al., eds., *Applications of Categories in Computer Science*, vol. 177 of *LMS Lecture Note Series*, 1–44. Cambridge Univ. Press, 1992.

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

[Gib93] J. Gibbons. Upwards and Downwards Accumulations on Trees. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, eds., *Proc. of 2nd Int. Conf. on Math. of Program Construction, MPC'92 (Oxford, June/July 1992)*, vol. 669 of *Lect. Notes in Comp. Sci.*, 122–138. Springer-Verlag, 1993.

[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. *Theor. Comp. Sci.*, 260(1–2):139–163, 2001.

[Oli98] J. N. Oliveira. “Explosive” Programming Controlled by Calculation. Techn. Report UM-DI TR 02/98, Dep. de Informática, Univ. do Minho, Braga, 1998.

[Uus01] T. Uustalu. (Co)monads from Inductive and Coinductive Types. In L. M. Pereira and P. Quaresma, eds., *Proc. of 2001 APPIA-GULP-PRODE Joint Conf. on Decl. Programming, AGP’01 (Évora, Sept. 2001)*, 47–61. Univ. do Évora, 2001.

[UV99] T. Uustalu and V. Vene. Primitive (Co)recursion and Course-of-value (Co)iteration, Categorically. *Informatica*, 10(1):5–26, 1999.

[UVP01] T. Uustalu, V. Vene, and A. Pardo. Recursion Schemes from Comonads. *Nordic J. of Computing*, 8(3):366–390, 2001.

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