

# The Recursion Scheme from the Cofree Recursive Comonad

Tarmo Uustalu<sup>1</sup>

*Institute of Cybernetics at Tallinn University of Technology,  
Akadeemia tee 21, EE-12618 Tallinn, Estonia*

Varmo Vene<sup>2</sup>

*Dept. of Computer Science, University of Tartu,  
J. Livi 2, EE-50409 Tartu, Estonia*

---

## Abstract

We instantiate the general comonad-based construction of recursion schemes for the initial algebra of a functor  $F$  to the cofree recursive comonad on  $F$ . Differently from the scheme based on the cofree comonad on  $F$  in a similar fashion, this scheme allows not only recursive calls on elements structurally smaller than the given argument, but also subsidiary recursions. We develop a Mendler formulation of the scheme via a generalized Yoneda lemma for initial algebras involving strong dinaturality and hint a relation to circular proofs à la Cockett, Santocanale.

*Keywords:* structured recursion, comonad-based recursion, cofree comonads, cofree recursive comonads, Mendler recursion, circular proofs

---

## 1 Introduction

In this paper, we are interested in structured recursion schemes for initial algebras. These are a central tool for programming with inductive types in total functional programming languages like Charity of Cockett [8] or type-theoretically inspired dependently typed languages.

We have previously [36] developed a general structured recursion scheme that, for the initial algebra of a functor  $F$ , is parameterized by a comonad and a distributive law of the functor  $F$  over the comonad. We have also [34] demonstrated a technique for casting conventional-style structured recursion schemes into a format with similarities to general recursion that makes them convenient for programming while maintaining the beneficial totality guarantee. Originating from Mendler [19], this format is known as Mendler-style recursion, but has also been promoted under

---

<sup>1</sup> [tarmo@cs.ioc.ee](mailto:tarmo@cs.ioc.ee)

<sup>2</sup> [varmo@cs.ut.ee](mailto:varmo@cs.ut.ee)

the name of ‘type-based termination’. The idea is to control the manipulation of data within a recursion by polymorphic typing. Our approach to Mendler recursion is a cute simple application of the Yoneda lemma and its variants.

One instance of the comonad-based scheme arises from the cofree comonad on  $F$ , sending an object  $A$  to the cofree  $F$ -algebra on  $A$ , i.e., the final  $A \times F$ -coalgebra where  $(A \times F)X =_{\text{df}} A \times FX$ . This gives course-of-value iteration, the recursion scheme where recursive calls can be made not only on the predecessor of the given argument but on all structurally smaller values. A related comonad is the cofree recursive comonad on  $F$ , delivering initial  $A \times F$ -algebras. One could ask: Does this comonad also yield a recursion scheme?

In this paper, we answer exactly this question in the affirmative. There is indeed a canonical distributive law of  $F$  over the cofree recursive comonad on  $F$  and hence a recursion scheme is obtained. What is more, the cofree recursive comonad forms, despite being a little more technical, in a certain sense a more natural basis for a recursion scheme than the cofree comonad. The recursion scheme from the cofree recursive comonad supports subsidiary recursions on the predecessors of the given argument. It also admits a tenable Mendler-style formulation and becomes in that format a useful tool for devising categorical semantic descriptions for circular sequent versions of typed lambda-calculi that Cockett [7] and Santocanale [24] have studied as possible cores for total functional programming languages.

In this paper we study the new recursion scheme foremostly as an instance of comonad-based recursion and of the technique of deriving Mendler-style recursion schemes from conventional-style schemes. discussing the rather technical application to circular proofs only tangentially. This will be the subject of a separate paper with Cockett.

The paper is organized as follows. In Section 2, we first review the comonad-based recursion scheme [36] and especially its instantiation to cofree comonads, including course-of-value iteration. Then we continue with the Yoneda lemma, together with a generalized version for final coalgebras, and Mendler-style recursion. In Section 3, we proceed to the instance arising when the cofree comonad is replaced with the cofree recursive comonad. In Section 4, we comment on a Haskell implementation of the combinators. The connection of the Mendler-style scheme to circular proofs is explained in Section 5. Following an orientation about related work in Section 6 we conclude with final remarks in Section 7.

## 2 Structured recursion schemes for initial algebras

### 2.1 Recursion schemes from comonads

We start by reviewing structured recursion for initial algebras of functors (inductive types).

Let  $F$  be an endofunctor (typically, a polynomial functor representing a signature) on a category  $\mathcal{C}$  (typically a category with finite products, finite coproducts and possibly exponents, as well as with initial algebras and final coalgebras of functors of interest to us; we think of **Set**). We are interested in the initial  $F$ -algebra, which we denote  $(\mu F, \text{in}_F)$ . Lambek’s lemma states that the algebra

structure  $\text{in}_F : F(\mu F) \rightarrow \mu F$  (which we think of as the constructor of the inductive type) is an isomorphism.

The primary function definition principle associated to the initial  $F$ -algebra is the recursion scheme of *iteration* (a.k.a. *fold*), which is exactly its initiality: for any  $F$ -algebra  $(C, \phi)$ , there exists a unique  $F$ -algebra map  $f : (\mu F, \text{in}_F) \rightarrow (C, \phi)$ , i.e., a unique map  $\mu F \rightarrow C$ , satisfying

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\text{in}_F} & \mu F \\ Ff \downarrow & & \downarrow f \\ FC & \xrightarrow{\phi} & C \end{array}$$

We denote this map  $f$  by  $\text{iter}_F(\phi)$ . It is useful to think that  $f$  is defined here as the composition of an analysis of the given argument of interest into its predecessors<sup>3</sup> ( $\text{in}_F^{-1}$ ), recursive calls of  $f$  on these predecessors ( $Ff$ ) and assembling the result ( $\phi$ ).

A number of further structured recursion principles are consequences of iteration.

We [36] have shown (and Bartels [3] did the same independently for the dual situation of final coalgebras) that a wide variety of them are instances of general scheme parameterized by a comonad  $D = (D, \varepsilon, \delta)$  and a distributive law  $\kappa$  of the functor  $F$  over it. The idea is that call trees would have  $FD$  rather than  $F$  as the branching factor.

Specifically, *comonad-based recursion* says this: Given a comonad  $D$  and a distributive law  $\kappa$  of  $F$  over  $D$ , for any  $FD$ -algebra  $(C, \phi)$ , there exists a unique map  $f : \mu F \rightarrow C$  (denoted  $\text{comrec}_F(\kappa, \phi)$ ), satisfying

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\text{in}_F} & \mu F \\ F\iota \downarrow & & \downarrow f \\ FD(\mu F) & & \\ FDf \downarrow & & \\ FDC & \xrightarrow{\phi} & C \end{array}$$

where  $\iota =_{\text{df}} \text{iter}_F(D\text{in}_F \circ \kappa_{\mu F}) : \mu F \rightarrow D(\mu F)$  turns out to be a coalgebra structure of not only the functor  $D$  but also the comonad  $D$ . The map defined by the scheme is constructible as the postcomposition of an ordinary iteration with the counit. We have

$$\text{comrec}_F(\kappa, \phi) = \varepsilon_C \circ \text{iter}_F(D\phi \circ \kappa_{DC} \circ F\delta_C)$$

or, diagrammatically,

$$\begin{array}{ccccccc} F(\mu F) & \xrightarrow{\text{in}_F} & & & & & \mu F \\ Fg \downarrow & & & & & & \downarrow g \\ FDC & \xrightarrow{F\delta_C} & FDDC & \xrightarrow{\kappa_{DC}} & DFDC & \xrightarrow{D\phi} & DC \\ & & & & & & \downarrow \varepsilon_C \\ & & & & & & C \end{array} \quad \left. \vphantom{\begin{array}{ccccccc} F(\mu F) & \xrightarrow{\text{in}_F} & & & & & \mu F \end{array}} \right\} f$$

<sup>3</sup> Viewing elements of  $\mu F$  as wellfounded  $F$ -branching trees, these would be the immediate subtrees of the given tree.

Two important special cases covered by comonad-based recursion are primitive recursion and course-of-value iteration.

*Primitive recursion* (allowing direct use of the predecessors of the given argument) states that, for any object  $C$  and map  $\phi : F(C \times \mu F) \rightarrow C$ , there is a unique map  $f$  (denoted  $\text{rec}_F(\phi)$ ) satisfying

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\text{in}_F} & \mu F \\ F\langle f, \text{id}_{\mu F} \rangle \downarrow & & \downarrow f \\ F(C \times \mu F) & \xrightarrow{\phi} & C \end{array}$$

Primitive recursion is recovered from comonad-based recursion by taking  $DA =_{\text{df}} A \times \mu F$ ,  $\kappa_A =_{\text{df}} \langle F\text{fst}, \text{in}_F \circ F\text{snd} \rangle : F(A \times \mu F) \rightarrow FA \times \mu F$ . As a result,  $\iota = \langle \text{id}_{\mu F}, \text{id}_{\mu F} \rangle : \mu F \rightarrow \mu F \times \mu F$  and  $Df \circ \iota = \langle f, \text{id}_{\mu F} \rangle : \mu F \rightarrow C \times \mu F$ .

A slightly more general scheme (supporting simultaneity with an independent iteration) is obtained by choosing  $DA =_{\text{df}} A \times E$  where  $(E, \chi)$  is any  $F$ -algebra and  $\kappa_A =_{\text{df}} \langle F\text{fst}, \chi \circ F\text{snd} \rangle : F(A \times E) \rightarrow FA \times E$ . In this case  $\iota = \langle \text{id}_{\mu F}, \text{iter}_F(\chi) \rangle : \mu F \rightarrow \mu F \times E$  and  $Df \circ \iota = \langle f, \text{iter}_F(\chi) \rangle : \mu F \rightarrow C \times E$ . Primitive recursion corresponds to the instance  $(E, \chi) =_{\text{df}} (\mu F, \text{in}_F)$ .

*Course-of-value iteration* is the scheme that allows recursive calls on not only the predecessors of the given argument, but on all structurally smaller elements. For an object  $A$  and functor  $H$ , let  $A \times H$  denote the functor defined by  $(A \times H)X =_{\text{df}} A \times HX$ . The scheme says this: Any object  $C$  and map  $\phi : F(\nu(C \times F)) \rightarrow C$  defines a unique map  $f : \mu F \rightarrow C$  (denoted  $\text{cviter}_F(\phi)$ ) satisfying

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{\text{in}_F} & \mu F \\ F(\text{coit}_{C \times F}(\langle f, \text{in}_F^{-1} \rangle)) \downarrow & & \downarrow f \\ F(\nu(C \times F)) & \xrightarrow{\phi} & C \end{array}$$

The map  $\phi$  has access to recursive call results for all elements structurally smaller than the given argument. These are stored in a tree-like datastructure of the same shape as the argument (essentially a labelled copy of the argument), except that, by its type, this tree is not guaranteed to be wellfounded.

This scheme is obtained from the cofree comonad on  $F$  and the canonical distributive law of  $F$  over it.

We look first at a more general version involving the cofree comonad on any functor.

The cofree comonad on a functor  $H$  is carried by the functor that sends an object  $A$  to the final  $A \times H$ -coalgebra. We define  $DA =_{\text{df}} \nu(A \times H)$ ,  $\varepsilon_A =_{\text{df}} \text{fst} \circ \text{out}_{A \times H} : DA \rightarrow A$ ,  $\theta_A =_{\text{df}} \text{snd} \circ \text{out}_{A \times H} : DA \rightarrow HDA$ .

Further, we use coiteration (finality) to define  $\delta_A =_{\text{df}} \text{coit}_{DA \times H}(\langle \text{id}_{DA}, \theta_A \rangle) :$

$DA \rightarrow DDA$ , so  $\delta_A$  is a unique map  $f$  satisfying

$$\begin{array}{ccc} DA & \xrightarrow{\theta_A} & HDA \\ \parallel & \downarrow f & \downarrow Hf \\ DA & \xleftarrow{\varepsilon_{DA}} DDA & \xrightarrow{\theta_{DA}} HDDA \end{array}$$

The cofree comonad is given by the data  $(D, \varepsilon, \delta)$ , constituting a comonad, and the natural transformation  $\sigma : D \rightarrow H$  defined by  $\sigma_A =_{\text{df}} H\varepsilon_A \circ \theta_A : DA \rightarrow HA$ .

Distributive laws of  $F$  over  $D$  are in a natural bijection with natural transformations  $FD \rightarrow HF$ . The distributive law  $\bar{\lambda}$  of  $F$  over  $D$  induced by a natural transformation  $\lambda : FD \rightarrow HF$  is defined by  $\bar{\lambda}_A =_{\text{df}} \text{coit}_{FA \times H}(\langle F\varepsilon_A, \lambda_{DA} \circ F\delta_A \rangle) : FDA \rightarrow DFA$ , so  $\bar{\lambda}_A$  is a unique map  $f$  satisfying

$$\begin{array}{ccc} FDA & \xrightarrow{F\delta_A} FDDA & \xrightarrow{\lambda_{DA}} HFDA \\ \swarrow F\varepsilon_A & \downarrow f & \downarrow Hf \\ FA & \xleftarrow{\varepsilon_{FA}} DFA & \xrightarrow{\theta_{FA}} HDFA \end{array}$$

Given a functor  $H$  together with a natural transformation  $\lambda : FD \rightarrow HF$ , we can instantiate the general scheme with  $\kappa =_{\text{df}} \bar{\lambda}$ .

Course-of-value iteration is given by the special case  $HA =_{\text{df}} FA$ ,  $\lambda_A =_{\text{df}} F\sigma_A : FDA \rightarrow FFA$ . In this situation  $\iota = \text{coit}_{\mu F \times F}(\langle \text{id}_{\mu F}, \text{in}_F^{-1} \rangle)$  where  $\text{in}_F^{-1} =_{\text{df}} \text{iter}_F(F\text{in}_F) : \mu F \rightarrow F(\mu F)$  is the inverse of  $\text{in}_F$ .

Slightly more generally one can use a general functor  $H$  equipped with a natural transformation  $\chi : FH \rightarrow HF$ , as this gives a natural transformation  $\lambda : FD \rightarrow HF$  via  $\lambda_X = \chi_X \circ F\sigma_X$ . In this case  $\iota = \text{coit}_{\mu F \times H}(\langle \text{id}_{\mu F}, \text{iter}_F(H\text{in}_F \circ \chi_{\mu F}) \rangle)$ .

Generalized primitive recursion is a degenerate case where  $HA =_{\text{df}} E$  in which case a natural transformation  $\chi : FH \rightarrow HF$  becomes an algebra structure  $FE \rightarrow E$ .

Another special case is course-of-value primitive recursion, arising from  $HA =_{\text{df}} \mu F \times FA$ ,  $\chi_A =_{\text{df}} \langle \text{in}_F \circ F\text{fst}, F\text{snd} \rangle : F(\mu F \times FA) \rightarrow \mu F \times FFA$ .

## 2.2 A Yoneda lemma for final coalgebras

Next we proceed to the contravariant Yoneda lemma. Below we will reformulate our recursion schemes relying on this important fact, but we also need a generalization.

The *Yoneda lemma* states this: For any functor  $K : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$  and object  $C \in |\mathcal{C}|$ , there is an isomorphism

$$i_{K,C} : [\mathcal{C}^{\text{op}}, \mathbf{Set}](\mathcal{C}(-, C), K-) \rightarrow KC$$

natural in  $K$  and  $C$ <sup>4</sup>. The isomorphism is defined by  $i_{K,C}(\Theta) =_{\text{df}} \Theta_C \text{id}_C$  and its inverse by  $i_{K,C}^{-1}(x)(f) =_{\text{df}} Kf(x)$ . An important special case (of the *Yoneda embedding*) is  $K =_{\text{df}} \mathcal{C}(-, D)$  establishing an isomorphism  $[\mathcal{C}^{\text{op}}, \mathbf{Set}](\mathcal{C}(-, C), \mathcal{C}(-, D)) \rightarrow \mathcal{C}(C, D)$ .

<sup>4</sup> This is ignoring the issue that we cannot know the domain to actually be a set before having established that it is isomorphic to the codomain, about which we assume this.

It is not unnatural to ask whether the Yoneda lemma generalizes in any interesting way to mixed-variant hom-functors. The answer turns out to be positive: there is a generalization and it will be very useful for us. Curiously, we have not come across it in the literature in this explicit form<sup>5</sup>, but it is intimately related to the fact that the final  $F$ -coalgebra is the (big!) colimit of the  $F$ -coalgebra structure forgetting functor.

To present the generalized Yoneda lemma we must first digress to introduce strong (a.k.a. Barr) dinatural transformations [21].

Let  $H, K : \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathcal{D}$  be functors. A *dinatural transformation*  $H \rightrightarrows K$  is a family of maps  $\Theta_X : H(X, X) \rightarrow K(X, X)$  in  $\mathcal{D}$  for all objects  $X$  in  $\mathcal{C}$  such that, for any map  $f : X \rightarrow Y$  in  $\mathcal{C}$ , the following hexagon commutes:

$$\begin{array}{ccccc}
 & & H(X, X) & \xrightarrow{\Theta_X} & K(X, X) & & \\
 & & \nearrow^{H(f, X)} & & \searrow^{K(X, f)} & & \\
 H(Y, X) & & & & & & K(X, Y) \\
 & & \searrow^{H(Y, f)} & & \nearrow^{K(f, Y)} & & \\
 & & H(Y, Y) & \xrightarrow{\Theta_Y} & K(Y, Y) & & 
 \end{array}$$

A *strongly dinatural transformation* is also a family of maps  $\Theta_X : H(X, X) \rightarrow K(X, X)$  in  $\mathcal{D}$  for all objects  $X$  in  $\mathcal{C}$ , but the coherence condition is that, for any map  $f : X \rightarrow Y$  in  $\mathcal{C}$ , object  $W$  and maps  $p_0 : W \rightarrow H(X, X)$ ,  $p_1 : W \rightarrow H(Y, Y)$  in  $\mathcal{D}$ , if the square in the following diagram commutes, then so does the hexagon:

$$\begin{array}{ccccc}
 & & H(X, X) & \xrightarrow{\Theta_X} & K(X, X) & & \\
 & & \nearrow^{p_0} & \searrow^{H(X, f)} & & & \\
 W & & & & H(X, Y) & \Rightarrow & K(X, Y) \\
 & & \searrow^{p_1} & \nearrow^{H(f, Y)} & & & \\
 & & H(Y, Y) & \xrightarrow{\Theta_Y} & K(Y, Y) & & 
 \end{array}$$

Differently from the case of ordinary (Dubuc-Street) dinaturals [11], strong dinaturals compose unproblematically. We denote by  $\text{SDinat}(\mathcal{C}, \mathcal{D})$  the category of mixed-variant functors  $\mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathcal{D}$  and strong dinatural transformations between them. (As a downside, however,  $\mathcal{D}$  being Cartesian closed does not imply that so is  $\text{SDinat}(\mathcal{C}, \mathcal{D})$ .)

We are now ready to introduce the *generalized contravariant Yoneda lemma*: For any functors  $K : \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$  and  $H : \mathcal{C} \rightarrow \mathcal{C}$ , there is an isomorphism

$$i_{K, H} : \text{SDinat}(\mathcal{C}, \mathbf{Set})(\mathcal{C}(-, H+), K-) \rightarrow K(\nu H)$$

natural in  $K$  and  $H$ . This isomorphism is defined by  $i_{K, H}(\Theta) =_{\text{df}} \Theta_{\nu H} \text{out}_H$  and its inverse by  $i_{K, H}^{-1}(x)_C(\phi) =_{\text{df}} K(\text{coit}_H(\phi))(x)$ . Unwinding the definition, we learn that a family of maps  $\Theta_X : \mathcal{C}(X, HX) \rightarrow KX$  for any  $X$  is strongly dinatural if, for any  $f : X \rightarrow Y$ ,  $\phi : X \rightarrow HX$ ,  $\psi : Y \rightarrow HY$ , the condition  $Hf \circ \phi = \psi \circ f$  (of  $f$  being an  $H$ -coalgebra map) implies  $\Theta_X \phi = Kf(\Theta_Y \psi)$ . To verify that  $i_{K, H}$  is an isomorphism indeed, we convince ourselves that  $i_{K, H}^{-1}(i_{K, H}(\Theta))_X(\phi) = K(\text{coit}_H(\phi))(\Theta_{\nu H} \text{out}_H) =$

<sup>5</sup> (however see our old NWPT '00 slides [31] as well as the recent discussion in the Types mailinglist [17])

$\Theta_X \phi$  (using that  $\Theta$  is strongly dinatural and  $\text{out}_H \circ \text{coit}_H(\phi) = H\text{coit}_H(\phi) \circ \phi$ ) and  $i_{K,H}(i_{K,H}^{-1}(x)) = K(\text{coit}_H(\text{out}_H))(x) = K\text{id}_{\nu H}(x) = \text{id}_{K(\nu H)}(x) = x$ .

The specific choice  $K =_{\text{df}} \mathcal{C}(-, D)$  gives an isomorphism  $\text{SDinat}(\mathcal{C}, \mathbf{Set})(\mathcal{C}(-, H+), \mathcal{C}(-, D)) \rightarrow \mathcal{C}(\nu H, D)$ .

Note that the standard Yoneda lemma is recovered from the generalization by specializing for  $HX =_{\text{df}} C$  (with the effect that  $\nu H = C$ ,  $\text{out}_H = \text{id}_C$ ,  $\text{coit}_H(f) = f$ ).

### 2.3 Mendler-style recursion

The Yoneda lemma allows us to shape recursion schemes in the style of Mendler [19]. The idea is to go higher-order, so a recursion can be specified in terms of a transformation of functions to functions rather than (potential) function results to function results, and to use polymorphic typing to prohibit unwanted data manipulations that might otherwise become possible.

Recall that conventional-style iteration says that, for any  $F$ -algebra  $(C, \phi)$  (i.e., an object  $C$  and map  $\phi : FC \rightarrow C$ ), there is a unique map  $f$  satisfying  $f \circ \text{in}_F = \phi \circ Ff$ . Picking  $K =_{\text{df}} \mathcal{C}(F-, C)$ , we learn from the Yoneda lemma that maps  $FC \rightarrow C$  are in natural bijection with natural transformations  $\mathcal{C}(-, C) \rightarrow \mathcal{C}(F-, C)$ . The *Mendler-style* formulation of *iteration* is based on exactly this observation. It says: For any object  $C$  and natural transformation  $\Phi : \mathcal{C}(-, C) \rightarrow \mathcal{C}(F-, C)$ , there is a unique map  $f : \mu F \rightarrow C$  (denoted  $\text{miter}_F(\Phi)$ ) satisfying

$$f \circ \text{in}_F = \Phi_{\mu F} f$$

The two formats are interchangeable:  $\text{miter}_F(\Phi) = \text{iter}_F(\Phi \text{id}_C)$  and  $\text{iter}_F(\phi) = \text{miter}_F(\Phi)$  where  $\Phi_Y f =_{\text{df}} \phi \circ Ff$  for  $f : Y \rightarrow C$ . For intuition, it is useful to think of  $\Phi$  as an operation sending any approximation of  $f$  capable of handling the predecessors of the given argument of interest to an improved approximation of  $f$  that can also handle this argument (all this without actually looking at the predecessors).

For general *comonad-based recursion*, a similar reformulation is possible: Given a comonad  $D$  and a distributive law  $\kappa$  of  $F$  over  $D$ , for any object  $C$  and natural transformation  $\Phi : \mathcal{C}(-, DC) \rightarrow \mathcal{C}(F-, C)$ , we have a unique map  $f : \mu F \rightarrow C$  (denoted  $\text{mcomiter}_F(\kappa, \Phi)$ ) satisfying

$$f \circ \text{in}_F = \Phi_{\mu F}(Df \circ \iota)$$

where  $\iota =_{\text{df}} \text{iter}_F(D\text{in}_F \circ \kappa_{\mu F}) : \mu F \rightarrow D(\mu F)$ .

For a scheme of this generality a Mendler-style formulation is not particularly advantageous over the conventional formulation. In specific cases, however, considerably more specialized Mendler-style formulations are possible.

In the case of *primitive recursion*, we can take advantage of the natural isomorphism  $\mathcal{C}(C, A_0) \times \mathcal{C}(C, A_1) \cong \mathcal{C}(C, A_0 \times A_1)$  to arrive at the following formulation: For any object  $C$  and natural transformation  $\Phi : \mathcal{C}(-, C) \times \mathcal{C}(-, \mu F) \rightarrow \mathcal{C}(F-, C)$ , we have a unique map  $f : \mu F \rightarrow C$  (denoted  $\text{mrec}_F(\Phi)$ ) satisfying

$$f \circ \text{in}_F = \Phi_{\mu F}(f, \text{id}_{\mu F})$$

In the case of *course-of-value iteration* we can additionally apply the generalized Yoneda lemma of the previous subsection for final coalgebras. It tells us that the maps  $F(\nu(C \times F)) \rightarrow C$  are in a natural bijection with the strong dinatural transformations  $\mathcal{C}(-, C \times F+) \rightarrow \mathcal{C}(F-, C)$ . We get: Any object  $C$  and strongly dinatural transformation  $\Phi : \mathcal{C}(-, C) \times \mathcal{C}(-, F+) \rightarrow \mathcal{C}(F-, C)$  define a unique solution  $f : \mu F \rightarrow C$  to the equation

$$f \circ \text{in}_F = \Phi_{\mu F}(f, \text{in}_F^{-1})$$

Notice that products, pairing, final coalgebras and coiteration of the category  $\mathcal{C}$  are absent in the reformulated scheme. They disappeared in the “refunctionalization” due to the Yoneda lemmas. Hence, especially with this scheme, the Mendler-style format is lighter than the conventional formulation in the sense that it does not work with a type of trees to record recursive call results.

### 3 The scheme from the cofree recursive comonad

#### 3.1 Cofree recursive comonads

Recursive comonads, dualizing the completely iterative monads of Aczel, Adámek et al. [2,20], are comonads  $D \cong \text{Id} \times D_0$  supporting unique solvability of guarded equations of a certain kind.

While the cofree comonad on  $H$  is given by the final coalgebras of the functors  $A \times H$  for all objects  $A$ , the *cofree recursive comonad* on  $H$  is defined by the initial algebras instead.

Let us look at the data of this comonad. For convenience, we rely on the isomorphism  $\mu(A \times H) \cong A \times \mu(H(A \times \text{Id}))$ . We define  $DA =_{\text{df}} A \times \mu(H(A \times \text{Id}))$ ,  $\varepsilon_A =_{\text{df}} \text{fst} : DA \rightarrow A$ ,  $\theta_A =_{\text{df}} \text{in}_{H(A \times \text{Id})}^{-1} \circ \text{snd} : DA \rightarrow HDA$ . Further, we let  $\delta_A =_{\text{df}} \langle \text{id}_{DA}, \text{rec}_{H(A \times \text{Id})}(\text{in}_{H(DA \times \text{Id})} \circ H \langle \text{fst}, \text{snd} \circ \text{snd} \rangle, \text{fst} \circ \text{snd}) \rangle \circ \text{snd} : DA \rightarrow DDA$ . It is not hard to verify that  $\delta_A$  is a unique map  $f$  satisfying

$$\begin{array}{ccc} DA & \xrightarrow{\theta_A} & HDA \\ \parallel & \downarrow f & \downarrow Hf \\ DA & \xleftarrow{\varepsilon_{DA}} DDA & \xrightarrow{\theta_{DA}} HDDA \end{array}$$

This condition was the defining (coiteration) description for  $\delta_A$  in the case of the cofree comonad. In our new situation it is a derived characterization.

The comonad is given by the data  $(D, \varepsilon, \delta)$ . The cofreeness as a recursive comonad also involves  $\sigma : D \rightarrow H$  defined by  $\sigma_A =_{\text{df}} H\varepsilon_A \circ \theta_A$ .

Differently from the cofree comonad case, we can no more get a distributive law of  $F$  over  $D$  from any natural transformation  $FD \rightarrow HF$ . But a canonical distributive law  $\kappa$  of  $F$  over  $D$  still exists, if  $D$  is the cofree recursive comonad on  $H =_{\text{df}} F$ . It is defined by  $\kappa_A =_{\text{df}} \langle F\text{fst}, \text{in}_{F(A \times \text{Id})} \circ F(\text{iter}_{F(A \times \text{Id})}(\langle F\text{fst}, \text{in}_{F(A \times \text{Id})} \circ F\text{snd} \rangle) \circ \text{snd}) \rangle : FDA \rightarrow DFA$ . With some effort one can check that  $\kappa_A$  is a unique solution in  $f$

of the equation system

$$\begin{array}{ccccc}
 & & FDA & \xrightarrow{F\theta_A} & FFDA \\
 & F\varepsilon_A \swarrow & \downarrow f & & \downarrow Ff \\
 FA & \xleftarrow{\varepsilon_{FA}} & DFA & \xrightarrow{\theta_{FA}} & FDFA
 \end{array}$$

which is the same condition that defines  $\kappa_A$  by coiteration, if  $D$  is the cofree comonad on  $F$ .

### 3.2 The recursion scheme, basic version

Given the existence of a distributive law  $\kappa$  of  $F$  over the cofree recursive comonad  $D$  on  $F$ , we can instantiate the comonad-based recursion scheme with these  $D$  and  $\kappa$ .

To do so, we have to work out the coalgebra structure  $\iota : \mu F \rightarrow D(\mu F)$ . Generally,  $\iota =_{\text{df}} \text{iter}_F(D\text{in}_F \circ \kappa_{\mu F}) : \mu F \rightarrow D(\mu F)$ . A calculation demonstrates that a simpler expression for our particular choice of  $D$  and  $\kappa$  is  $\iota = \langle \text{id}_{\mu F}, \text{rec}_F(\text{in}_{F(C \times \text{Id})} \circ F\langle \text{snd}, \text{fst} \rangle) \rangle$ . As a result, for  $f : \mu F \rightarrow C$ ,  $Df \circ \iota = \langle f, \text{rec}_F(\text{in}_{F(C \times \text{Id})} \circ F\langle f \circ \text{snd}, \text{fst} \rangle) \rangle$

Summing up, we have established this scheme: For any object  $C$  and map  $\phi : F(C \times \mu(F(C \times \text{Id}))) \rightarrow C$ , there exists a unique map  $f : \mu F \rightarrow C$  (we denote it  $\text{iter}_{\times F}(\phi)$ ) satisfying

$$\begin{array}{ccc}
 F(\mu F) & \xrightarrow{\text{in}_F} & \mu F \\
 \downarrow F\langle f, \text{rec}_F(\text{in}_{F(C \times \text{Id})} \circ F\langle f \circ \text{snd}, \text{fst} \rangle) \rangle & & \downarrow f \\
 F(C \times \mu(F(C \times \text{Id}))) & \xrightarrow{\phi} & C
 \end{array}$$

From the general reduction of comonad-based recursion to iteration we can conclude, after simplifications, that

$$\text{iter}_{\times F}(\phi) = \text{fst} \circ \text{iter}_F(\langle \phi, \text{in}_{F(C \times \text{Id})} \rangle)$$

What is the practical meaning of this scheme? On the surface it is very similar to course-of-value iteration: the map  $\phi$  assembling the result of  $f$  on the given argument operates on a tree-like datastructure with recursive call results for all structurally smaller elements. But now the datastructure comes with a wellfoundedness guarantee, in fact it is a labelled copy of the parsed version of the given argument. How can this be used? Is there any difference from course-of-value iteration? Intuitively, if the datastructure is wellfounded, one can induct on it. Exactly this is also the potential of the intermediate datastructure here. The scheme shown allows recursions on the intermediate datastructure, which really mean subsidiary (inner) recursions on the predecessors of the given argument, with access to the results of the main (outer) recursion. The Mendler-style formulation to follow in the next subsection will make this very evident.

In the light of the above reflection, it is not surprising that the new scheme subsumes both primitive recursion (from labelled copies of the predecessors of the given

argument one should well be able to extract these predecessors) and course-of-value iteration. Using the natural transformations  $r_A =_{\text{df}} \text{id}_A \times \text{iter}_{F(A \times \text{Id})}(\text{in}_F \circ F \text{snd}) : A \times \mu(F(A \times \text{Id})) \rightarrow A \times \mu F$  and  $s_A =_{\text{df}} \text{out}_{A \times F}^{-1} \circ (\text{id}_A \times \text{iter}_{F(A \times \text{Id})}(F \text{out}_{A \times F}^{-1})) : A \times \mu(F(A \times \text{Id})) \rightarrow \nu(A \times F)$  we recover:  $\text{rec}_F(\phi) = \text{iter}_{\mathbf{x}_F}(\phi \circ F r_C)$  (for  $\phi : F(C \times \mu F) \rightarrow C$ ) and  $\text{cviter}_F(\phi) = \text{iter}_{\mathbf{x}_F}(\phi \circ F s_C)$  (for  $\phi : F(\nu(C \times F)) \rightarrow C$ ).

### 3.3 Mendler version of the scheme

The generalized contravariant Yoneda lemma for final coalgebras dualizes into a generalized covariant Yoneda lemma for initial algebras: For any functors  $K : \mathcal{C} \rightarrow \mathbf{Set}$  and  $H : \mathcal{C} \rightarrow \mathcal{C}$ , there is an isomorphism

$$i_{K,H} : \text{SDinat}(\mathcal{C}, \mathbf{Set})(\mathcal{C}(H-, +), K+) \rightarrow K(\mu H)$$

natural in  $K$  and  $H$ .

But a Mendler version of the recursion scheme from the cofree recursive comonad is not obtained as simply as in the case of the cofree comonad. Simply applying the generalized covariant Yoneda lemma for initial algebras is not possible, as the occurrence of  $\mu(F(C \times \text{Id}))$  in the expression  $\mathcal{C}(F(C \times \mu(F(C \times \text{Id}))), C)$  is contravariant.

Instead, we must first use the contravariant Yoneda lemma and then proceed with the generalized covariant Yoneda lemma for initial algebras, and finally apply the contravariant Yoneda lemma once again. We get the following chain of isomorphisms:

$$\begin{aligned} & \mathcal{C}(F(C \times \mu(F(C \times \text{Id}))), C) \\ & \cong [\mathcal{C}^{\text{op}}, \mathbf{Set}]_Y(\mathcal{C}(Y, C \times \mu(F(C \times \text{Id}))), \mathcal{C}(FY, C)) \\ & \cong [\mathcal{C}^{\text{op}}, \mathbf{Set}]_Y(\mathcal{C}(Y, C) \times \mathcal{C}(Y, \mu(F(C \times \text{Id}))), \mathcal{C}(FY, C)) \\ & \cong [\mathcal{C}^{\text{op}}, \mathbf{Set}]_Y(\mathcal{C}(Y, C) \times \text{SDinat}(\mathcal{C}, \mathbf{Set})_{C'_-, C'_+}(\mathcal{C}(F(C \times C'_-), C'_+), \mathcal{C}(Y, C'_+)), \mathcal{C}(FY, C)) \\ & \cong [\mathcal{C}^{\text{op}}, \mathbf{Set}]_Y(\mathcal{C}(Y, C) \\ & \quad \times \text{SDinat}(\mathcal{C}, \mathbf{Set})_{C'_-, C'_+}([\mathcal{C}^{\text{op}}, \mathbf{Set}]_{Y'}(\mathcal{C}(Y', C \times C'_-), \mathcal{C}(FY', C'_+)), \mathcal{C}(Y, C'_+)), \\ & \quad \mathcal{C}(FY, C)) \\ & \cong [\mathcal{C}^{\text{op}}, \mathbf{Set}]_Y(\mathcal{C}(Y, C) \\ & \quad \times \text{SDinat}(\mathcal{C}, \mathbf{Set})_{C'_-, C'_+}([\mathcal{C}^{\text{op}}, \mathbf{Set}]_{Y'}(\mathcal{C}(Y', C) \times \mathcal{C}(Y', C'_-), \mathcal{C}(FY', C'_+)), \mathcal{C}(Y, C'_+)), \\ & \quad \mathcal{C}(FY, C)) \end{aligned}$$

One can now check that the recursion scheme of the previous subsection implies the following Mendler-style scheme: Any object  $\mathcal{C}$  and natural transformation

$$\begin{aligned} \Phi : & [\mathcal{C}^{\text{op}}, \mathbf{Set}]_Y(\mathcal{C}(Y, C) \\ & \times \text{SDinat}(\mathcal{C}, \mathbf{Set})_{C'_-, C'_+}([\mathcal{C}^{\text{op}}, \mathbf{Set}]_{Y'}(\mathcal{C}(Y', C) \times \mathcal{C}(Y', C'_-), \mathcal{C}(FY', C'_+)), \mathcal{C}(Y, C'_+)), \\ & \mathcal{C}(FY, C)) \end{aligned}$$

define a unique map  $f : \mu F \rightarrow C$ , which we denote  $\text{miter2}_F(\Phi)$ , satisfying

$$f \circ \text{in}_F = \Phi_{\mu F}(f, \text{update}_{\mu F}(\text{mrec}_F, f))$$

where  $\text{update}_Y$  takes a natural transformation

$$\Theta : [\mathcal{C}^{\text{op}}, \mathbf{Set}]_{Y'}(\mathcal{C}(Y', C'_-) \times \mathcal{C}(Y', \mu F), \mathcal{C}(FY', C'_+)) \xrightarrow{\text{C}'_-, \text{C}'_+} \mathcal{C}(Y, C'_+)$$

and a map  $g : \mu F \rightarrow C$  and sends them to a natural transformation

$$\text{update}_Y(\Theta, g) : [\mathcal{C}^{\text{op}}, \mathbf{Set}]_{Y'}(\mathcal{C}(Y', C) \times \mathcal{C}(Y', C'_-), \mathcal{C}(FY', C'_+)) \xrightarrow{\text{C}'_-, \text{C}'_+} \mathcal{C}(Y, C'_+)$$

in the expectable way.

We see that the result of  $f$  can depend on the results of recursive calls of  $f$  on the predecessors of the given argument of interest plus a “local, enhanced iterator” on these predecessors that, in addition to what the normal iterator can do, gains access to  $f$  (in the course of iteration) on all structurally smaller elements.

In principle, it is possible to modify the scheme so that subsidiary recursions can be according to more sophisticated schemes than iteration (even according to the main scheme itself). We will not discuss this here, as the version we have just shown is higher-order enough to confuse. We advise the reader to check the Haskell implementation that we will comment next.

## 4 A Haskell implementation

To illustrate the mechanics of the schemes, it is useful to show some programs. We accompany this paper by a Haskell implementation, with some examples of uses, of all combinators discussed in the previous sections. For most of them, we show several alternative definitions, corresponding, besides the basic computation rule (the recursion equation), also to encodings in terms of other combinators etc. The implementation is available at <http://cs.ioc.ee/~tarmo/haskell/msfp08/>.

Some remarks are in order. First, we use Haskell only as a demonstration (for the lack of a better alternative for our purposes). Haskell is not a total functional language. We work in a safe corner where we have set up some basic infrastructure for initial algebras and final coalgebras and we program with them confining ourselves to just this infrastructure (avoiding confusing initial algebras and final coalgebras, using general recursion etc.). Second, Haskell has no mechanisms for enforcing functor and comonad laws etc. Likewise we ignore the question of when strong dinaturality is free.

As the code uses rank- $n$  type signatures, it only runs with GHC, with the `-fglasgow-exts` option. Some ugliness (bureaucratic constructors/destructors) in the code is due to the fact that lambda-abstraction is not available for type constructors and a type constructor is not automatically a datatype constructor.

Here are some comments on the code. Inductive types and iteration in the conventional format are introduced via a recursive datatype and a general-recursive function definition describing how iteration computes.

```
newtype Mu f = In {ini :: f (Mu f)}
```

```
iter :: Functor f => (f c -> c) -> Mu f -> c
```

```
iter phi (In x) = phi (fmap (iter phi) x)
```

To define comonad-based recursion, we first introduce a type-constructor class of comonads with member functions for the counit and comultiplication, by subclassing from the Prelude-defined class of functors.

```
class Functor d => Comonad d where
  counit :: d a -> a
  comult :: d a -> d (d a)
```

Likewise, distributive laws of functors over comonads are defined as a multi-parameter class.

```
class (Functor f, Comonad d) => Dist d f where
  dist :: f (d a) -> d (f a)
```

We are ready to define the comonad-based recursor, parameterized by a comonad and a distributive law. We can either give a general-recursive definition stating the computation rule of the combinator

```
comrec :: Dist d f => (f (d c) -> c) -> Mu f -> c
comrec phi (In x) = phi (fmap (fmap (comrec phi) . iota) x)
  where iota = iter (fmap In . dist)
```

or choose to reduce it to iteration

```
comrec phi = counit . iter (fmap phi . dist . fmap comult)
```

Course-of-value iteration is an instance corresponding to the cofree comonad on  $F$  and the distributive law of  $F$  over it. As the cofree comonad is given by cofree algebras, we need to introduce coinductive types and coiteration.

```
newtype PairF f a x = PF { unPF :: (a, f x) }
```

```
instance Functor f => Functor (PairF f a) where ...
```

```
data Nu f = Outi { out :: f (Nu f) }
```

```
coit :: Functor f => (c -> f c) -> c -> Nu f
coit psi = Outi . fmap (coit psi) . psi
```

```
newtype Cofree f a = Cf { unCf :: Nu (PairF f a) }
```

```
instance Functor f => Functor (Cofree f) where ...
```

```
instance Functor f => Comonad (Cofree f) where
  counit = fst . unPF . out . unCf
  comult = Cf . coit (PF . pair Cf (snd . unPF . out)) . unCf
```

```
instance Functor f => Dist (Cofree f) f where
  dist = Cf . coit (PF
    . pair (fmap (fst . unPF . out . unCf))
      (fmap (fmap Cf . snd . unPF . out . unCf)))
```

Now course-of-value iteration can be defined “from scratch” by its computation rule

```
cviter :: Functor f => (f (Cofree f c) -> c) -> Mu f -> c
cviter phi (In x) = phi (fmap (Cf . coit (PF . pair (cviter phi) ini)) x)
```

but it also arises as a special case of comonad-based recursion

```
cviter = comrec
```

The new scheme relies on the cofree recursive comonad instead of the cofree comonad.

```
newtype FPair f a x = FP { unFP :: f (a, x) }
```

```
instance Functor f => Functor (FPair f a) where ...
```

```
newtype CofreeRec f a = CfR { unCfR :: (a, Mu (FPair f a)) }
```

```
instance Functor f => Functor (CofreeRec f) where ...
```

```
instance Functor f => Comonad (CofreeRec f) where
```

```
  counit = fst . unCfR
```

```
  comult = CfR .
```

```
    pair id
```

```
      (rec (In . FP . fmap (pair (CfR . pair fst (snd . unP . snd))
```

```
          (fst . unP . snd))
```

```
          . unFP) . snd . unCfR)
```

```
instance Functor f => Dist (CofreeRec f) f where
```

```
  dist = CfR . pair (fmap (fst . unCfR))
```

```
    (In . FP . fmap (iter (pair (fmap fst)
```

```
                        (In . FP . fmap snd)
```

```
                        . unFP) . snd . unCfR))
```

The recursor can be defined by its computation rule

```
iterx :: Functor f => (f (CofreeRec f c) -> c) -> Mu f -> c
```

```
iterx phi (In x) = phi (fmap
```

```
  (CfR . pair (iterx phi)
```

```
    (rec (In . FP . fmap (pair (iterx phi . snd) fst . unP))))
```

```
  x)
```

or as a specialization of the comonad-based recursor

```
iterx = comrec
```

Mendler-style recursion schemes do not need intermediate datastructures, instead they rely on higher-order functions and polymorphism. For Mendler-style iterator, the definition via the computation rule is

```
miter :: Functor f => (forall y. (y -> c) -> f y -> c) -> Mu f -> c
```

```
miter psi (In x) = psi (miter psi) x
```

while one can also reduce it to conventional-style iteration

```
miter psi = iter (psi id)
```

(the functoriality of  $F$  is really only made use of in the latter case).

Mendler-style course-of-value iteration and Mendler-style iteration with subsidiary iterations are defined by their computation rules by

```
mcviter :: Functor f => (forall y. (y -> c) -> (y -> f y) -> f y -> c)
                                     -> Mu f -> c
mcviter psi (In x) = psi (mcviter psi) ini x
```

```
miter2 :: Functor f => (forall y. (y -> c)
-> (forall c'. (forall y'. (y' -> c) -> (y' -> c')) -> f y' -> c')
                                     -> y -> c')
                                     -> f y -> c)
                                     -> Mu f -> c
miter2 psi (In x) = psi (miter2 psi) (update mrec (miter2 psi)) x
```

```
update ::
  (forall c'. (forall y'. (y' -> c') -> (y' -> Mu f) -> f y' -> c')
                                     -> y -> c')
  -> (Mu f -> c)
  -> (forall c'. (forall y'. (y' -> c) -> (y' -> c')) -> f y' -> c')
                                     -> y -> c')
```

```
update theta g = \ psi -> theta (\ de io -> psi (g . io) de)
```

(For `update`, it is necessary to explicitly give the type, otherwise Haskell's type inferencer will not generalize enough.) The reductions to the conventional-style schemes are

```
mcviter psi = cviter (psi (fst . unPF . out . unCf)
                        (fmap Cf . snd . unPF . out . unCf))
```

```
miter2 psi = iterx (psi (fst . unCfR)
                      (\ psi' x -> iter (psi' fst snd . unFP)
                                       (snd (unCfR x))))
```

To show some example uses of the combinators, we define the inductive type of natural numbers.

```
data N x = Z | S x
```

```
instance Functor N where ...
```

```
type Nat = Mu N
```

```
zero :: Nat          suc :: Nat -> Nat
zero = In Z         suc n = In (S n)
```

Fibonacci numbers (specified informally by  $fib_0 = 0$ ,  $fib_1 = 1$ ,  $fib(n + 2) =$

$fib(n + 1) + fib\ n$ ) can now be defined with conventional-style course-of-value iteration (relying on a datastructure with results on elements structurally smaller than the given argument)

```
fib :: Nat -> Int
fib = cviter phi where
  phi Z = 0
  phi (S (Cf (Outi (PF (_, Z)))))) = 1
  phi (S (Cf (Outi (PF (f, S (Outi (PF (f', _)))))))) = f + f'
```

or with Mendler-style iteration, considerably closer to the style of general recursion (relying on an assumed function for making the recursive calls and an assumed predecessor function)

```
fib = mcviter psi where
  psi _ _ Z = 0
  psi fibo ini (S n) = case ini n of
    Z -> 1
    S n' -> fibo n + fibo n'
```

Notice that, in the definition of `psi`, the functions `fibo` and `ini` are formal parameters (so the definitions of these functions from outside are overridden) and that `fib :: y -> Int`, `ini :: y -> N y`, `n :: y` for a fresh type variable `y` which only gets instantiated to `Nat` when an application of `fib` is evaluated using the computation rule.

To demonstrate a use of the recursion scheme from the cofree recursive comonad, we look at a simple example of mutual iteration, specified informally by  $foo\ 0 = 1$ ,  $bar\ 0 = 0$ ,  $foo\ (n + 1) = 3 * foo\ n + bar\ n$ ,  $bar\ (n + 1) = foo\ n + 3 * bar\ n$ . We take `foo` to be our main function of interest and `bar` to be an auxiliary function.

The definition by conventional-style scheme defines the auxiliary function by iterating on a datastructure containing the results of the main function on elements structurally smaller than the given argument:

```
foo :: Nat -> Int
foo = iterx phi where
  phi Z = 1
  phi (S (CfR (f, fs))) =
    let bar = iter phi' where
        phi' (FP Z) = 0
        phi' (FP (S (f', g'))) = f' + 3 * g'
    in 3 * f + bar fs
```

(in this definition, `bar :: Mu (FPair N Int) -> Int` inputs not a natural number but a datastructure containing the result of `foo` on every smaller number).

In the case of the Mendler-style scheme we rely on assumed functions for doing the recursive calls:

```
foo = miter2 psi where
  psi :: (y -> Int)
  -> (forall c'. (forall y'. (y' -> Int) -> (y' -> c')) -> N y' -> c')
```

```

                                                    -> y -> c')
-> N y -> Int
psi _ _ Z = 1
psi foo miter (S n) = let bar = miter psi' where
                        -- psi' :: (y' -> Int) -> (y' -> Int)
                        -> N y' -> Int
                        psi' _ _ Z = 0
                        psi' foo bar (S m) = foo m + 3 * bar m
in 3 * foo n + bar n
    
```

## 5 Connection to circular proofs

Let us now turn to sequent-style versions of intuitionistic proof systems or typed lambda-calculi with least and greatest fixedpoint operators. We will be informal in the section, only attempting to convey some intuitions, as a fuller treatment would necessarily involve a significant amount of introduction of concepts and notation. In particular, we show no terms, only types.

The constructor `in` and the conventional iterator `iter` justify the following  $\mu$ -right and  $\mu$ -left inference rules in the spirit of Park and Kozen for positive (syntactically functorial)  $F$ :

$$\frac{\Gamma \longrightarrow F(\mu F)}{\Gamma \longrightarrow \mu F} \qquad \frac{\Gamma, F(\prod \Gamma \Rightarrow C) \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

(keep in mind that  $\Gamma, A \longrightarrow C$  should be equiderivable with  $A \longrightarrow \prod \Gamma \Rightarrow C$ ). It is also possible to consider alternative  $\mu$ -left inference rules for other conventional-style recursors, e.g., for `rec` in the form

$$\frac{\Gamma, F((\prod \Gamma \Rightarrow C) \times \mu F) \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

However the resulting systems are not particularly pleasant proof-theoretically. Some drawbacks are that the  $\mu$ -detour cut elimination rule introduces inferences by the derived inference rule of “functoriality” (whereby the derivation of the functoriality rule for  $F$  is by an induction “outside the system” on the type expression for  $F$ ) and that the inference rules are quite far from the design ideal of sequent calculi that the premises should analyze the main formula of the conclusion solely in terms of its subformulae.

Mendler-style recursion combinators open up a different design space, which leads to more pleasing systems, but at the cost of introducing higher-order inference rules à la Schroeder-Heister [25] (well within the coding power of logical frameworks such as LF). One can think of this as natural deduction on sequents.

The Mendler-style iterator `miter` justifies the following rule:

$$\frac{\Gamma, Y \longrightarrow C \quad \vdots \quad \Gamma, FY \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

saying that to prove  $\Gamma, \mu F \longrightarrow C$ , it suffices to prove  $\Gamma, FY \longrightarrow C$  where  $Y$  is a fresh type variable and in that proof it is legitimate to use a hypothesis (“local axiom”)  $\Gamma, Y \longrightarrow C$ .

Other Mendler-style recursors can be exploited similarly. The primitive recursor `mrec` justifies the  $\mu$ -left inference rule

$$\Gamma, Y \longrightarrow C \quad \frac{\Gamma', \mu F \longrightarrow C'}{\Gamma', Y \longrightarrow C'}$$

$$\vdots$$

$$\frac{\Gamma, FY \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

allowing the proof of  $\Gamma, FY \longrightarrow C$  to also depend on a “local rule” by which the task of proving any sequent  $\Gamma', Y \longrightarrow C'$  can be reduced to proving  $\Gamma', \mu F \longrightarrow C'$ . (We note that while the rule based on `miter` was 2nd-order, this rule is already 3rd-order.) Likewise the course-of-value iterator `mcviter` leads to the  $\mu$ -left inference rule

$$\Gamma, Y \longrightarrow C \quad \frac{\Gamma', FY \longrightarrow C'}{\Gamma', Y \longrightarrow C'}$$

$$\vdots$$

$$\frac{\Gamma, FY \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

allowing  $\Gamma', Y \longrightarrow C'$  to be concluded from  $\Gamma', FY \longrightarrow C'$  within a proof of  $\Gamma, FY \longrightarrow C$ .

What about the recursor `miter2` from the cofree recursive comonad? The corresponding (4th-order)  $\mu$ -left inference rule is:

$$\Gamma, Y \longrightarrow C \quad \left( \begin{array}{c} \Gamma, Y' \longrightarrow C \quad \Gamma', Y' \longrightarrow C' \\ \vdots \\ \Gamma', FY' \longrightarrow C' \\ \hline \Gamma', Y \longrightarrow C' \end{array} \right)$$

$$\vdots$$

$$\frac{\Gamma, FY \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

Now the proof of  $\Gamma, FY \longrightarrow C$  may depend on inferences by a local rule reminiscent of that for `miter` except that the role of  $\mu F$  is played by  $Y$  and there is an extra hypothesis. This corresponds to subsidiary iteration which has access to the results of the main recursion.

Circular proof systems have been offered as an alternative design to the Park and Kozen style. The  $\mu$ -right and left inference rules are

$$\frac{\Gamma \longrightarrow F(\mu F)}{\Gamma \longrightarrow \mu F} \qquad \frac{\Gamma, F(\mu F) \longrightarrow C}{\Gamma, \mu F \longrightarrow C} \mu L^*$$

corresponding essentially to `in` and `in-1` (and no recursor). But the concept of proofs is nonstandard. A proof is a nonwellfounded tree, satisfying two conditions:

(a) it is rational (i.e., has finitely many distinct subtrees), (b) every infinite path in the tree contains a trace of occurrences of a  $\mu$ -type with infinitely many  $\mu L^*$ -inferences whose main formula is the trace's designated occurrence of that  $\mu$ -type in the conclusion of the inference (the *progress condition*).

Our claim is that Mendler-style sequent calculi provide a way to recast rational circular proofs into wellfounded proofs. There are two ingredients.

First, a proof being a rational tree means that it has cycles. Such a tree is representable as a wellfounded tree with backpointers. Due to the progress condition, the cycles can be shifted so that they all begin with  $\mu L^*$  inferences, which means that we can deal with backpointers by making a  $\mu$ -left rule whose premise depends on a hypothesis:

$$\frac{\Gamma, \mu F \longrightarrow C \quad \vdots \quad \Gamma, F(\mu F) \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

Wellfounded proofs using this rule represent exactly those rational circular proofs where every infinite path contains an infinite number of  $\mu L^*$  inferences. However, there is a problem: We are not keeping track about staying on the right trace.

This brings us to the second trick, the use of a fresh type variable to ensure that at the end of any cycle we are on the same trace. We reformulate our rule as

$$\frac{\Gamma, Y \longrightarrow C \quad \vdots \quad \Gamma, FY \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

(notice that this is the rule we obtained from `miter` in our discussion above). The new attempt is sound, contrarily to the previous one, which naively used  $\mu F$  instead of  $Y$ , but now a cycle starting at a  $\mu L^*$  inference must finish with the next one on the same trace. This is remedied by the rule inspired by `mcviter`:

$$\frac{\Gamma, Y \longrightarrow C \quad \frac{\Gamma', FY \longrightarrow C'}{\Gamma', Y \longrightarrow C'}}{\Gamma, \mu F \longrightarrow C} \quad \vdots \quad \frac{\Gamma, FY \longrightarrow C}{\Gamma, \mu F \longrightarrow C}$$

With this rule a cycle starting with a  $\mu L^*$  inference can pass through several further  $\mu L^*$  inferences before reaching a backpointer. But still a problem remains: These further  $\mu L^*$  inferences cannot be used as starting points of new (entangled) cycles.

This is partially remedied by the rule dictated by the `miter2` recursor:

$$\Gamma, Y \longrightarrow C \left( \begin{array}{c} \Gamma, Y' \longrightarrow C \quad \Gamma', Y' \longrightarrow C' \\ \vdots \\ \frac{\Gamma', FY' \longrightarrow C'}{\Gamma', Y \longrightarrow C'} \\ \vdots \\ \frac{\Gamma, FY \longrightarrow C}{\Gamma, \mu F \longrightarrow C} \end{array} \right)$$

Here a cycle starting at a  $\mu L^*$  inference can pass through a subsequent  $\mu L^*$  inference which can start other cycles, but these subsidiary cycles must necessarily be of length 1 then—not passing through further  $\mu L^*$  inferences. This is a restriction, but the reason is that, with the way we chose to formulate our Mendler-style recursor from the cofree recursive comonad, the only flavour of subsidiary recursion we allow is iteration. It is possible (by an additional fixedpoint construction) to formulate a rule where the subsidiary recursions can be of the same kind as the main recursion.

## 6 Related work

Total functional programming has been experimented with in Cockett’s Charity language [8], based on categorical combinators; it has also been advocated for a number of years by D. A. Turner [29,30]. Totality is also the natural paradigm for type-theoretical proof assistants and dedicated languages for dependently typed programming where proofs are identified with functions.

Functional programming with inductive and coinductive types based on categorical combinators was pioneered by Hagino [15] and Cockett and Spencer [9]. The comonad-based recursion scheme was introduced by Uustalu, Vene, Pardo [36], in an attempt to extract the common pattern of primitive recursion and course-of-value iteration [33]. Independently, Bartels [3] described a dual corecursion scheme and looked also at the corresponding coinduction principle. Capretta, Uustalu, Vene [6] took the original work further, showing that the scheme extends from initial algebras to any coalgebras recursive in the sense of Osius.

Mendler-style (co)recursion originates from Mendler’s work [19] on a typed lambda-calculus with inductive and coinductive types. The original work was exploited and developed further (still in the typed lambda-calculi context) in papers by Leivant [16], Parigot [22], Geuvers [12], Splawski [26], de Bruin [10], Uustalu and Vene [32], Matthes [18]. More recently, Barthe et al. [4] and Abel [1] have popularized Mendler recursion under the name of “type-based termination”. The more subtle Mendler-style course-of-value iteration was first formulated by de Bruin [10]. The semantic connection to the Yoneda lemma was first pointed out by Uustalu, Vene [32]. An account of the Church (a.k.a. Böhm-Berarducci) encodings (fold/build syntax) of inductive types and the deforestation rule of fold/build fusion in terms of strongly dinatural transformations was given by Ghani, Uustalu and Vene [14].

Free completely iterative monads are the subject of a series of recent papers

by Aczel, Adámek, Milius and Velebil [2,20]. Cofree recursive comonads have been employed in a discussion of an application to tree transformations by Uustalu and Vene [35]; they were also touched upon in the work of Ghani et al. [13].

Circular sequent calculi (where proofs are rational or nonwellfounded trees) for logics with least and greatest fixedpoint operators have mainly been investigated in the context of classical predicate or modal logic. Such calculi were first considered by Pliuskevičius [23], Stirling and Walker [28] (in a model checking context) and Wałukiewicz [37] and have been studied further by, e.g., Sprenger and Dam [27], Brotherston and Simpson [5] (this list is far from complete). Into the context of sequent-style versions of intuitionistic proof systems and typed lambda-calculi, circular proof systems were introduced by Santocanale [24] and Cockett [7]. With syntactic progress conditions, they are related to typed lambda-calculi with syntactically guarded (co)recursion. With fresh type variables they are related to Mendler-style (co)recursion, as we have claimed here.

## 7 Conclusions

We have shown that the cofree recursive comonad yields a meaningful instance of the comonad-based recursion scheme. It gives a scheme that supports subsidiary recursions on the predecessors of the given argument. This happens by allowing recursion on a datastructure containing the results of the main recursion. While at first we had little intuition about the mechanics of this scheme, in retrospect it appears as a completely natural strengthening of course-of-value iteration, where essentially the same datastructure can only be finitely observed, as it comes without a wellfoundedness guarantee. It is also pleasant that there is a connection to the repeated regeneration phenomenon in circular proofs.

A number of issues require further work. First, in this paper we have depended on a rather naive approach to polymorphism based on strong dinaturality. We have done so deliberately, as this seems to be a good level of abstraction (due to the Yoneda-like natural isomorphisms that we can work with), allowing for a viable separation of concerns. But actual applications of our constructions must depend on meaningful sufficient conditions for a family of maps to be strongly dinatural, so we need to identify the most appropriate such conditions for our purposes. Second, the connection of Mendler recursion to circular proof systems should be worked out to the level of a full account of a categorical semantics for a circular sequent-style term calculus à la Cockett based on Mendler-style recursors. Third, the Yoneda “reductions” employed in this paper are bound to be related to deforestation and defunctionalization/refunctionalization in ways we have not yet explored. We would like to work out the details of these connections.

### Acknowledgements

Tarmo Uustalu is grateful to the PC of HOR 2007 for the invitation to give a talk and to Robin Cockett for many useful discussions on circular proofs. We also thank our referees for their helpful criticism.

This work was partially supported by the Estonian Science Foundation under grant no. 6940.

## References

- [1] Abel, A., *Termination checking with types*, Theor. Inform. and Appl. **38**(4) (2004), pp. 277–319.
- [2] Aczel, P., J. Adámek, S. Milius, and J. Velebil, *Infinite trees and completely iterative theories: a coalgebraic view*, Theor. Comput. Sci., **300**(1–3) (2003), pp. 1–45.
- [3] Bartels, F., *Generalised coinduction*, Math. Struct. in Comput. Sci. **13**(2) (2003), pp. 321–348.
- [4] Barthe, G., M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu, *Type-based termination of recursive definitions*, Math. Struct. in Comput. Sci. **14**(1) (2004), pp. 97–141.
- [5] Brotherston, J. and A. Simpson, *Complete sequent calculi for induction and infinite descent*, in: “Proc. of 22nd IEEE Symp. on Logic in Comput. Sci., LICS 2007,” IEEE CS Press, 2007, pp. 51–62.
- [6] Capretta, V., T. Uustalu, and V. Vene, *Recursive coalgebras from comonads*, Inform. and Comput., **204**(4) (2006), pp. 437–468.
- [7] Cockett, R., *Deforestation, program transformation, and cut-elimination*, in: A. Corradini, M. Lenisa, and U. Montanari, eds., “Proc. of 4th Wksh. on Coalgebraic Methods in Computer Science, CMCS ’01,” Electron. Notes in Theor. Comput. Sci. **44**(1), Elsevier, 2001, pp. 88–127.
- [8] Cockett, R. and T. Fukushima, “About Charity,” Yellow Series Report 92/480/18, Dept. of Comput. Sci., Univ. of Calgary, 1992.
- [9] Cockett, R. and D. Spencer, *Strong categorical datatypes I*, in: R. A. G. Seely, ed., “Proc. of Int. Summer Category Theory Meeting (Montréal, June 1991),” Canadian Mathematical Society Conf. Proc. **13**, AMS, 1992, pp. 141–169.
- [10] de Bruin, P. J., “Inductive Types in Constructive Languages,” PhD thesis, Dept. of Comput. Sci., Univ. of Groningen, 1995.
- [11] Dubuc, E. and R. Street, *Dinatural transformations*, in: S. Mac Lane, ed., “Reports of the Midwest Category Seminar IV,” Lecture Notes in Mathematics **137**, Springer, 1970, pp. 126–137.
- [12] Geuvers, H., *Inductive and coinductive types with iteration and recursion*, in: B. Nordström, K. Pettersson, and G. Plotkin, eds., “Informal Proc. of Wksh. on Types for Proofs and Programs, TYPES ’92,” Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., 1992, pp. 193–217.
- [13] Ghani, N., C. Lüth, F. De Marchi and J. Power, *Dualising initial algebras*, Math. Struct. in Comput. Sci. **13**(2) (2003), pp. 349–370.
- [14] Ghani, N., T. Uustalu, and V. Vene, *Build, augment and destroy, universally*, in: W.-N. Chin, ed., “Proc. of 2nd Asian Symp. on Programming Languages and Systems, APLAS 2004,” Lect. Notes in Comput. Sci. **3302**, Springer, 2004, pp. 327–347.
- [15] Hagino, T., *A typed lambda calculus with categorical type constructors*, in: D. H. Pitt, A. Poigné, and D. E. Rydeheard, eds., “Proc. of 2nd Int. Conf. on Category Theory and Computer Science, CTCS ’87,” Lect. Notes in Comput. Sci. **283**, Springer, 1987, pp. 140–157.
- [16] Leivant, D., *Contracting proofs to programs*, in: P. Odifreddi, ed., “Logic and Computer Science,” APIC Studies in Data Processing **31**, Academic Press, 1990, pp. 279–327.
- [17] Levy, P. B., T. Altenkirch et al., *Polymorphic isomorphisms*, Discussion in the Types mailinglist, 2008. <http://lists.seas.upenn.edu/pipermail/types-list/2008/001179.html>
- [18] Matthes, R., *Tarski’s fixed-point theorem and lambda calculi with monotone inductive types*, Synthese **133**(1) (2002), pp. 107–129.
- [19] Mendler, N. P., *Inductive types and type constraints in the second-order lambda-calculus*, Ann. of Pure and Appl. Logic **51**(1–2) (1991), pp. 159–172.
- [20] Milius, S., *Completely iterative algebras and completely iterative monads*, Inform. and Comput. **196**(1) (2005), pp. 1–41.
- [21] Mulry, P. S., *Strong monads, algebras and fixed points*, in: M. P. Fourman, P. T. Johnstone, and A. M. Pitts, eds., “Applications of Categories in Computer Science,” London Math. Society Lecture Note Series **177**, Cambridge Univ. Press, 1992, pp. 202–216.
- [22] Parigot, M., *Recursive programming with proofs*, Theor. Comput. Sci. **94**(2) (1992), pp. 335–356.
- [23] Pluskevičius, R., *Investigation of finitary calculi for the temporal logics by means of infinitary calculi*, in: B. Rován, ed., “Proc. of 15th Int. Conf. on Math. Foundations of Comput. Sci., MFCS ’90,” Lect. Notes in Comput. Sci. **452**, Springer, 1990, pp. 464–469.

- [24] Santocanale, L., *A calculus of circular proofs and its categorical semantics*, in: M. Nielsen and U. Engberg, eds., “Proc. of 5th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2002,” Lect. Notes in Comput. Sci. **2303**, Springer, 2002, pp. 357–371.
- [25] Schroeder-Heister, P., *A natural extension of natural deduction*, J. of Symb. Logic **49**(4) (1984), pp. 1284–1300.
- [26] Splawski, Z., “Proof-Theoretic Approach to Inductive Definitions in ML-Like Programming Languages versus Second-Order Lambda Calculus,” PhD thesis, Wrocław Univ., 1993.
- [27] Sprenger, C. and M. Dam, *On the structure of inductive reasoning: circular and tree-shaped proofs in the  $\mu$ -calculus*, in: A. D. Gordon, ed., “Proc. of 6th Int. Conf. on Foundations of Software Science and Computational Structures, FoSSaCS 2003,” Lect. Notes in Comput. Sci. **2620**, Springer, 2003, pp. 425–440.
- [28] Stirling, C. and D. Walker, *Local model checking in the modal  $\mu$ -calculus*, Theor. Comput. Sci. **89**(1) (1991), pp. 161–177.
- [29] Turner, D. A., *Elementary strong functional programming*, in: P. Hartel and R. Plasmeijer, eds., “Proc. of 1st Int. Symp. on Functional Programming Languages in Education, FPLE ’95,” Lect. Notes in Comput. Sci. **1022**, Springer, 1995, pp. 1–13.
- [30] Turner, D. A., *Total functional programming*, J. of Univ. Comput. Sci. **10**(7) (2004), pp. 751–768.
- [31] Uustalu, T., *Strong dinaturality and initial algebras*, Slides for NWPT ’00, 2000. <http://www.ii.uib.no/~nwpt00/Proceedings/Uustalu-slides.ps>
- [32] Uustalu, T. and V. Vene, *A cube of proof systems for the intuitionistic predicate  $\mu, \nu$ -logic*, in: M. Haveraaen and O. Owe, eds., “Selected Papers from 8th Nordic Wksh. on Programming Theory, NWPT ’96,” Res. Rep. 248, Dept. of Informatics, Univ. of Oslo, 1997, pp. 237–247.
- [33] Uustalu, T. and V. Vene, *Primitive (co)recursion and course-of-value (co)iteration, categorically*, Informatica **10**(1), pp. 5–26, 1999.
- [34] Uustalu, T. and V. Vene, *Coding recursion à la Mendler (extended abstract)*, in: J. Jeuring, ed., “Proc. of 2nd Wksh. on Generic Programming, WGP 2000,” Techn. Report UU-CS-2000-19, Dept. of Comput. Sci., Utrecht Univ., 2000, pp. 69–85.
- [35] Uustalu, T. and V. Vene, *The dual of substitution is redecoration*, in: K. Hammond and S. Curtis, eds., “Trends in Functional Programming 3,” Intellect, 2002, pp. 99–110.
- [36] Uustalu, T., V. Vene, and A. Pardo, *Recursion schemes from comonads*, Nordic J. of Comput. **8**(3) (2001), pp. 366–390.
- [37] Wałukiewicz, I., *Completeness of Kozen’s axiomatization of the propositional  $\mu$ -calculus*, Inform. and Comput. **157**(1–2) (2000), pp. 142–182.