Coding Recursion a la Mendler

Extended Abstract

Tarmo Uustalu[1*] and Varmo Vene[2]

[1] Inst. of Cybernetics, Tallinn Technical University
Akadeemia tee 21, EE-126 18 Tallinn, Estonia; tarmo@cs.ioc.ee
[2] Inst. of Computer Science, University of Tartu
J. Liivi 2, EE-504 09 Tartu, Estonia; varmo@cs.ut.ee

Abstract We advocate the Mendler style of coding terminating recursion schemes as combinators by showing on the example of two simple and much used schemes (course-of-value iteration and simultaneous iteration) that choosing the Mendler style can sometimes lead to handier constructions than following the construction style of cata and para like combinators.

1 Introduction

This paper is intended as an advert for something we call the *Mendler style*. This is a not too widely known style of coding terminating recursion schemes by combinators that differs from the construction style of the famous cata and para combinators (for iteration and primitive-recursion, respectively) [Mal90,Mee92], here called the *conventional style*. The paper argues that, for certain recursion schemes, opting for the Mendler style gives simpler constructions than choosing the conventional style. This is demonstrated on the example of two simple and useful terminating recursion schemes, course-of-value iteration and simultaneous iteration. Course-of-value iteration is a scheme for the definition of functions of an inductive argument where the result value for any given argument value is determined by the "course" of the result value for its ancestral (not just the result values for its immediate predecessors, as is the case for simple iteration). Simultaneous iteration is a scheme for the definition of functions of two inductive arguments where the result value for any given pair of argument values is determined by the result values for the pairs of their immediate predecessors.

What is the Mendler style about and what is its difference from the conventional style? Let us explain this on the example of simple iteration. Suppose we want to define a function f from an inductive type μF to type C by iteration. The recursive defining equation would be of the form

$$f \circ \mathsf{in}_F = \ldots f \ldots$$

or, more concisely,

$$f \circ \mathsf{in}_F = \Phi\, f \tag{1}$$

where Φ is some definable function on morphisms from μF to C which returns morphisms from $F\,\mu F$ to C. Just in this general form, the equation (1) does not necessarily define f iteratively; in fact, it may not define f at all, as it need not determine the output of f on all possible inputs. The conventional method to ensure that (1) is a definition by iteration (the method used in the cata construction) consists in insisting that $\Phi\, f = \phi \circ F\, f$ (for any $f : \mu F \to C$) where ϕ is some morphism that can only be from $F\ C$ to C. This means imposing a relatively *syntactic* condition on the right-hand side of (1): no expression other than '$\phi \circ F\, f$' is acceptable unless we are eager and able to prove that it equals $\phi \circ F\, f$ (which may require quite a bit of equational reasoning).

* On leave to Dept. of Informatics, Univ. of Minho, Campus de Gualtar, P-4710-057 Braga, Portugal; tarmo@di.uminho.pt.

The Mendler-style method to ensure the iterative definition-ness of (1) (the method of the Mendler-style version of the cata construction) is to leave the form of its right-hand side as we ideally wanted it, i.e., '$\Phi\, f$', but to require Φ not to use any specifics about the type μF. This is achievable by insisting that Φ be an instance of a function parametric in A on morphisms from A to C for any type A that returns morphisms from $F\,A$ to C (which is verifiable by type-checking). This means adopting a considerably more *semantic* approach to controlling the right-hand side of (1).

In terms of what can be programmed, the two approaches to coding iteration are clearly equivalent. Any ϕ can be lifted to a Φ doing the same job by defining $\Phi\,\gamma = \phi \circ F\gamma$ and any Φ can be unlifted to an equipotent ϕ by defining $\phi = \Phi\,\mathsf{id}$ (the Yoneda lemma!). But being able to program more is not really the point with the Mendler style. The point is not having to struggle as much when programming the same. It is not possible to demonstrate this on the example of simple iteration (opting for the Mendler style for this scheme yields too little gain; course-of-value iteration and simultaneous iteration discussed below are good examples). But the reason why the Mendler style is easier to work with in some cases can be pointed out already on the example of simple iteration. The morphism $\phi : F\,C \to C$ relates (candidate) result values of f (for argument values that stand in a well-defined relation to each other, but cannot be named). The function Φ, sending morphisms $A \to C$ to morphisms $F\,A \to C$, at the same time, is best thought of as relating (candidate) approximations of f (restrictions of f to various subtypes A of μF). The conventional style gives the programmer access by reference only to (candidate) result values of the function. The Mendler style allows explicit reference to argument values and to (candidate) approximations of the function, and this can make writing programs a good deal easier.

What are the origins of the Mendler style and why the name? It seems that the technique was invented by N P Mendler, in type theory. In [Men87] (a conference paper), he studied a simply typed lambda calculus with an unusual primitive-recursion combinator; [Men91] (the journal version) treats a simpler calculus with a similar iteration combinator. Mendler had a type theory reason for dissatisfaction with the conventional style: the right-hand side of the defining equation for a conventional-style combinator always mentions F (more exactly, the morphism-mapping component of F, the map-function). This, although fine from the category theory point of view, is not too well with a type theorist. In category theory, functors are "born" bipartite, consisting of an object mapping and a morphism mapping. In type theory, the situation is different: at first, there is a (definable) object mapping (an iterative predicate on the structure of the expression for it tells whether it is certain that the object mapping is functorial, i.e., extendible into a functor), and only then a morphism mapping is associated to it (by explicitly defining it using another, similar iteration). Some important works commenting on [Men87]/[Men91] and, in particular, on the embeddings between simply typed lambda calculi with conventional- and Mendler-style iterators and primitive-recursors and system F are [Lei90,Geu92,Spł93].

In [UV97,UV00,Uus98,Mat98,Mat00], which are more recent, an observation is emphasized that type theory systems with Mendler-style recursion combinators do not loose any of their desirable meta-theoretic properties, if μ is allowed to apply also to non-covariant functors. It is also shown how to interpret such systems in lattice theory (μF is not necessarily a (pre-)fixed point of F, if F is non-monotonic). The same lattice theory explanations appear in [SU99], again a work on embeddings between type theory systems. [UV99b] gives a category-theoretic intepretation of a simply typed lambda calculus with Mendler-style iterator, based on suitably defined concepts of m-algebra and m-algebra homomophism.

In this paper, we build on [UV99b], but ignore the mixed-variant (difunctorial) generalization which leads away from conventional datatypes. Instead, we concentrate on recursion combinators definable via the cata combinator, i.e., the conventional-style iterator that the value constructor for an inductive type comes together with. The discussions of conventional-style combinators for course-of-value iteration and simultaneous iteration appeared in [UV99a,Uus99]; those of Mendler-style combinators for these recursion schemes are a novel contribution of this paper. A type-theoretic presentation of the Mendler-style course-of-value recursion combinator can be found in [UV97,UV00,Uus98].

The rest of the paper is organized as follows. In Section 2, we first recapitulate the customary categorical constructions for inductive types and iteration, namely, the initial algebra and the cata combinator, and then present the Mendler-style version of the latter. In Section 3, we describe the construction of both a conventional and a Mendler-style combinator for course-of-value iteration, trying to show that the Mendler-style construction is simpler and handier to work with. In Section 4, we do the same with simultaneous iteration, and in Section 5, we conclude by discussing both some issues for further work that are internal to the setting we have adopted in this paper and some possible connections to be checked out in the future with work by others. In Sections A, B, C of the Appendix, all recursion combinators and examples of the paper appear implemented in Haskell (on the Mendler-style side, this requires using "rank-2 type signatures"). This code may well be easier to read than the combinator expressions in the paper, as the these are "pointfree", but the code is "pointwise".

2 Initial algebras and catamorphisms

Any generic definition of the notion of inductive type states that an inductive type is a type on which some fixed recursion scheme coded in some fixed way properly defines functions. The most common practice is to use iteration in the conventional coding. This is also how inductive types are usually constructed in category theory: catamorphisms, unique homomorphisms from an initial algebra to other algebras, capture iteration. Let us recall this categorical-algebraic definition of inductive type and accept it as the basis for the subsequent exposition.

Let F be a (covariant) functor. An *F-algebra* with carrier C is a morphism $\phi : F\,C \to C$. A *homomorphism* between F-algebras ψ, ϕ with carriers D, C is a morphism $h : D \to C$ such that

$$h \circ \psi = \phi \circ F\,h$$

$$\begin{tikzcd} F\,D \arrow[r, "\psi"] \arrow[d, "F\,h"'] & D \arrow[d, "h"] \\ F\,C \arrow[r, "\phi"] & C \end{tikzcd}$$

Assume that there exists an initial F-algebra. Denote it by in_F and its carrier by μF. Then, by the definition of initiality, for any F-algebra ϕ with carrier C, a unique morphism $f : \mu F \to C$ exists such that

$$f \circ \mathsf{in}_F = \phi \circ F\,f$$

$$\begin{tikzcd} F\,\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "F\,f"'] & \mu F \arrow[d, "f"] \\ F\,C \arrow[r, "\phi"] & C \end{tikzcd}$$

This f is called the *F-catamorphism* of ϕ and denoted by $(\!|\,\phi\,|\!)_F$. The equation for catamorphisms says roughly that the result value of f for any given argument value is equal to ϕ applied to the result values of f for its predecessors; this is the conventional coding of iteration. The statement that the equation has a unique solution says that conventional-style iteration defines functions.

The definition says that the cata combinator is calculationally characterized by the following law. If ϕ is an F-algebra, then

$$f = (\!|\,\phi\,|\!)_F \;\Leftrightarrow\; f \circ \mathsf{in}_F = \phi \circ F\,f$$

The following laws of cata cancellation, reflection, and fusion are straightforward corollaries. If ϕ is an F-algebra, then

$$(\!|\,\phi\,|\!)_F \circ \mathsf{in}_F = \phi \circ F\,(\!|\,\phi\,|\!)_F$$

$$\mathsf{id} = (\!|\,\mathsf{in}_F\,|\!)_F$$

If ψ, ϕ are F-algebras, then

$$h \circ \psi = \phi \circ F\,h \;\Rightarrow\; h \circ (\!| \psi |\!)_F = (\!| \phi |\!)_F$$

The cancellation law, directed left-to-right, gives the natural computation rule for catamorphisms.

Example 1. Define $\mathsf{N}\,C = 1 + C$. Then $\mathsf{Nat} = \mu\mathsf{N}$ behaves as the type of natural numbers. The constructors $\mathsf{zero} : 1 \rightarrow \mathsf{Nat}$ and $\mathsf{succ} : \mathsf{Nat} \rightarrow \mathsf{Nat}$ can be defined by stating $\mathsf{zero} = \mathsf{in}_{\mathsf{N}} \circ \mathsf{inl}$, $\mathsf{succ} = \mathsf{in}_{\mathsf{N}} \circ \mathsf{inr}$.

The function $\mathsf{n2i} : \mathsf{Nat} \rightarrow \mathsf{Int}$ converting naturals to integers admits the following catamorphic definition.

$$\mathsf{n2i} = (\!| [\,\bar{0}, (\bar{1}\bar{+})\,] |\!)_N$$

Define $\mathsf{L}E\,C = 1 + E \times C$. Then $\mathsf{List}E = \mu(\mathsf{L}E)$ behaves as the type of lists with element type E. The constructors $\mathsf{nil} : 1 \rightarrow \mathsf{List}E$ and $\mathsf{cons} : E \times \mathsf{List}E \rightarrow \mathsf{List}E$ are defined by letting $\mathsf{nil} = \mathsf{in}_{\mathsf{L}E} \circ \mathsf{inl}$, $\mathsf{cons} = \mathsf{in}_{\mathsf{L}E} \circ \mathsf{inr}$.

Coding iteration in the Mendler style leads to the definition of a different combinator, the m-cata combinator [UV97,UV00,UV99b]. Let F be a functor for which an initial algebra exists. An *F-m-algebra* with carrier C is a function Φ on morphisms $A \rightarrow C$ returning morphisms $F\,A \rightarrow C$ for any A such that, if α, β are morphisms to C, then

$$\alpha = \beta \circ g \;\Rightarrow\; \Phi\,\alpha = \Phi\,\beta \circ F\,g$$

$$\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[d, "g"'] & C \arrow[d, equal] & & F\,A \arrow[r, "\Phi\,\alpha"] \arrow[d, "F\,g"'] & C \arrow[d, equal] \\
B \arrow[r, "\beta"] & C & \Rightarrow & F\,B \arrow[r, "\Phi\,\beta"] & C
\end{tikzcd}$$

Equivalently, one might just require that, if α is a morphism to C, then

$$\Phi\,\alpha = \Phi\,\mathsf{id} \circ F\,\alpha$$

$$\begin{tikzcd}
F\,A \arrow[r, "\Phi\,\alpha"] \arrow[d, "F\,\alpha"'] & C \arrow[d, equal] \\
F\,C \arrow[r, "\Phi\,\mathsf{id}"] & C
\end{tikzcd}$$

(i.e., the above condition for $\beta := \mathsf{id}$ only). The side condition on Φ states that Φ is *parametric*. For any Φ that is definable, its satisfaction is therefore a "theorem for free" in the sense of Wadler [Wad89].

A *homomorphism* between two F-m-algebras Ψ, Φ with carriers D, C is a morphism $h : D \rightarrow C$ such that, if δ is a morphism to D and γ is a morphism to C, then

$$h \circ \delta = \gamma \;\Rightarrow\; h \circ \Psi\,\delta = \Phi\,\gamma$$

$$\begin{tikzcd}
 & D \arrow[dd, "h"] & & & D \arrow[dd, "h"] \\
A \arrow[ur, "\delta"] \arrow[dr, "\gamma"'] & & \Rightarrow & F\,A \arrow[ur, "\Psi\,\delta"] \arrow[dr, "\Phi\,\gamma"'] & \\
 & C & & & C
\end{tikzcd}$$

An equivalent condition is

$$h \circ \Psi\,\mathsf{id} = \Phi\,h$$

$$\begin{tikzcd}
 & D \arrow[dd, "h"] \\
F\,D \arrow[ur, "\Psi\,\mathsf{id}"] \arrow[dr, "\Phi\,h"'] & \\
 & C
\end{tikzcd}$$

(the above for $\delta := \mathsf{id}$ only).

Assume now that there is an initial F-algebra. Then, for any F-m-algebra Φ with carrier C, a unique morphism $f : \mu F \to C$ exists such that, if m is a morphism to μF and γ is a morphism to C, then

$$f \circ m = \gamma \;\Rightarrow\; f \circ (\mathsf{in}_F \circ F\, m) = \Phi\, \gamma$$

$$\begin{tikzcd}
& \mu F \arrow[dd, "f"] \\
A \arrow[ur, "m"] \arrow[dr, "\gamma"'] & \\
& C
\end{tikzcd}
\quad\Rightarrow\quad
\begin{tikzcd}
& \mu F \arrow[dd, "f"] \\
F\, A \arrow[ur, "\mathsf{in}_F \circ F\, m"] \arrow[dr, "\Phi\, \gamma"'] & \\
& C
\end{tikzcd}$$

or, equivalently,

$$f \circ \mathsf{in}_F = \Phi\, f$$

$$\begin{tikzcd}
& \mu F \arrow[dd, "f"] \\
F\, \mu F \arrow[ur, "\mathsf{in}_F"] \arrow[dr, "\Phi\, f"'] & \\
& C
\end{tikzcd}$$

(the above for $m := \mathsf{id}$). This shows that $\lambda m.\ \mathsf{in}_F \circ F\, m$ is an initial F-m-algebra with carrier μF. The f is called the *F-m-catamorphism* of Φ and denoted by $(\!|\, \Phi \,|\!)^{\mathsf{m}}_F$. The equation for m-catamorphisms is a Mendler-style coding of iteration. It is best thought of as saying that, if γ is the restriction of f to a subtype A of μF containing the predecessors of a given argument value, then the result value of f for this argument value is equal to $\Phi\, \gamma$ applied to it (m should be thought of as the "natural" embedding of A in μF). The statement that the equation has a unique solution says that Mendler-style iteration defines functions.

According to the definition, the m-cata combinator is characterized by the following law. If Φ is an F-m-algebra, then

$$f = (\!|\, \Phi \,|\!)^{\mathsf{m}}_F \;\Leftrightarrow\; f \circ \mathsf{in}_F = \Phi\, f$$

From this, the m-cata cancellation, reflection, and fusion laws follow straightforwardly. If Φ is an F-m-algebra, then

$$(\!|\, \Phi \,|\!)^{\mathsf{m}}_F \circ \mathsf{in}_F = \Phi\, (\!|\, \Phi \,|\!)^{\mathsf{m}}_F$$

$$\mathsf{id} = (\!|\, \lambda m.\ \mathsf{in}_F \circ F\, m \,|\!)^{\mathsf{m}}_F$$

If Ψ, Φ are F-m-algebras, then

$$(\forall \delta, \gamma.\ h \circ \delta = \gamma \;\Rightarrow\; h \circ \Psi\, \delta = \Phi\, \gamma) \;\Rightarrow\; h \circ (\!|\, \Psi \,|\!)^{\mathsf{m}}_F = (\!|\, \Phi \,|\!)^{\mathsf{m}}_F$$

Directing the cancellation law again gives the appropriate computation rule.

Example 2. The definition of the function $\mathsf{n2i}$ as a m-catamorphism is the following.

$$\mathsf{n2i} = (\!|\, \lambda \gamma.\ [\, \bar{\mathsf{0}}, (\bar{\mathsf{1}} \bar{+}) \circ \gamma \,] \,|\!)_N$$

Any catamorphism is an m-catamorphism and vice versa. If ϕ is an F-algebra, then

$$(\!|\, \phi \,|\!)_F = (\!|\, \lambda \gamma.\ \phi \circ F\, \gamma \,|\!)^{\mathsf{m}}_F$$

If Φ is an F-m-algebra, then

$$(\!|\, \Phi \,|\!)^{\mathsf{m}}_F = (\!|\, \Phi\, \mathsf{id} \,|\!)_F$$

3 Histomorphisms

The constructions of conventional and Mendler-style iteration combinators are almost equally trivial. A useful recursion scheme not lending itself too easily for conventional-style coding but admitting a neat coding in the Mendler style is course-of-value iteration: if no references are allowed in an equation to inputs of the function defined, then the equation has to refer to candidate "courses" of its outputs.

In the conventional style, course-of-value iteration admits the following coding [UV97,UV00,UV99a]. Let F again be a functor. Define $(C \ltimes F)A = C \times F\,A$ for any A and let $\mathsf{out}_{\_\ltimes F}$ stand for a final $C \ltimes F$-coalgebra, $\nu(C \ltimes F)$ for its carrier, and $[\!(\,\cdot\,)\!]_{\_\ltimes F}$ for the corresponding ana combinator. $\nu(C \ltimes F)$ is the coinductive type of F-*streams* for an element type C, $\mathsf{out}_{\_\ltimes F}$ the value destructor, and $[\!(\,\cdot\,)\!]_{\_\ltimes F}$ the conventional-style coiterator (generator) for F-streams. (So $\mathsf{out}_{\_\ltimes F}$ is a morphism $\nu(C \ltimes F) \to C \times F\,\nu(C \ltimes F)$ natural in C, and $[\!(\,\cdot\,)\!]_{C\ltimes F}$ is a function parametric in C sending any $\gamma : A \to C \times F\,A$, i.e., any $(C \ltimes F)$-coalgebra with carrier A, to the unique morphism $f : A \to \nu(C \ltimes F)$ such that $\mathsf{out}_{\_\ltimes F} \circ f = (\mathsf{id} \times F\,f) \circ \gamma$.)

A F-*cv-algebra* with carrier C is a morphism $\phi : F\,\nu(C \ltimes F) \to C$. A *homomorphism* between F-cv-algebras ψ, ϕ with carriers D, C is a morphism $h : D \to C$ such that

$$h \circ \psi = \phi \circ F\,\nu(h \ltimes F)$$

$$\begin{tikzcd}
F\,\nu(D \ltimes F) \arrow[r, "\psi"] \arrow[d, "F\,\nu(h\ltimes F)"'] & D \arrow[d, "h"] \\
F\,\nu(C \ltimes F) \arrow[r, "\phi"] & C
\end{tikzcd}$$

Assume the existence of an initial F-algebra. Then, for any F-cv-algebra ϕ with carrier C, there turns out to exist exactly one morphism $f : \mu F \to C$ such that

$$f \circ \mathsf{in}_F = \phi \circ F\,[\!(\,\langle\, f, \mathsf{in}_F^{-1}\,\rangle\,)\!]_{\_\ltimes F}$$

$$\begin{tikzcd}
F\,\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "F{[\!(\,\langle f, \mathsf{in}_F^{-1}\rangle\,)\!]_{C\ltimes F}}"'] & \mu F \arrow[d, "f"] \\
F\,\nu(_ \ltimes F) \arrow[r, "\phi"] & C
\end{tikzcd}$$

This f is called the F-*histomorphism* of ϕ and denoted by $\{\!|\,\phi\,|\!\}_F$. The characteristic equation of f says that the result value of f for any given argument value is equal to ϕ applied to the "course" of the result value of f for its ancestral. The intuitive reason why the "course" has to be coinductive and cannot be inductive is that, within the recursion scheme, we cannot assess the argument value and hence cannot set any bound on its depth.

The important calculation laws about the histo combinator are the following. If ϕ is an F-cv-algebra, then

$$f = \{\!|\,\phi\,|\!\}_F \;\Leftrightarrow\; f \circ \mathsf{in}_F = \phi \circ F\,[\!(\,\langle\, f, \mathsf{in}_F^{-1}\,\rangle\,)\!]_{\_\ltimes F}$$

If ϕ is an F-cv-algebra, then

$$\{\!|\,\phi\,|\!\}_F \circ \mathsf{in}_F = \phi \circ F\,[\!(\,\langle\, \{\!|\,\phi\,|\!\}_F, \mathsf{in}_F^{-1}\,\rangle\,)\!]_{\_\ltimes F}$$

$$\mathsf{id} = \{\!|\,\mathsf{in}_F \circ F\,(\mathsf{fst} \circ \mathsf{out}_{\_\ltimes F})\,|\!\}_F$$

If ψ, ϕ are F-cv-algebras, then

$$h \circ \psi = \phi \circ F\,\nu(h \ltimes F) \;\Rightarrow\; h \circ \{\!|\,\psi\,|\!\}_F = \{\!|\,\phi\,|\!\}_F$$

Any catamorphism is a histomorphism. If ϕ is an F-algebra, then

$$(\!|\,\phi\,|\!)_F = \{\!|\,\phi \circ F\,(\mathsf{fst} \circ \mathsf{out}_{\_\ltimes F})\,|\!\}_F$$

The histo combinator is reducible to the cata combinator. If ϕ is an F-cv-algebra, then

$$\{\!|\, \phi \,|\!\}_F = \mathsf{fst} \circ \mathsf{out}_{\_\ltimes F} \circ (\!|\, \mathsf{out}^{-1}_{\_\ltimes F} \circ \langle\, \phi, F\,\mathsf{id} \,\rangle \,|\!)_F$$

Example 3. Using the histo combinator, the following definitions can be given to the Fibonacci function $\mathsf{fibo} : \mathsf{Nat} \to \mathsf{Int}$ (from naturals to integers) and the function $\mathsf{evens} : \mathsf{List}E \to \mathsf{List}E$ selecting from a given list every second element, functions, which are "naturally" course-of-value-iterative.

$$\mathsf{fibo} = \{\!|\; \begin{bmatrix} \bar{0}, \\ \begin{bmatrix} \bar{1} \circ \mathsf{snd}, \\ \bar{+} \circ (\mathsf{id} \times (\mathsf{fst} \circ \mathsf{out}_{\_\ltimes \mathsf{N}})) \end{bmatrix} \circ \mathsf{distl} \circ \mathsf{out}_{\_\ltimes \mathsf{N}} \end{bmatrix} \;|\!\}_{\mathsf{N}}$$

$$\mathsf{evens} = \{\!|\; \begin{bmatrix} \mathsf{nil}, \\ \begin{bmatrix} \mathsf{nil}, \\ \mathsf{cons} \circ (\mathsf{id} \times (\mathsf{fst} \circ \mathsf{out}_{\_\ltimes \mathsf{L}E})) \end{bmatrix} \circ (\mathsf{snd} \circ \mathsf{out}_{\_\ltimes \mathsf{L}E}) \circ \mathsf{snd} \end{bmatrix} \;|\!\}_{\mathsf{L}E}$$

(Here, distl stands for the unique isomorphism $A \times (B_1 + B_2) \to (A \times B_1) + (A \times B_2)$ natural in A, B_1, B_2; $\mathsf{distl} = \mathsf{uncurry}' [\, \mathsf{curry}'\, \mathsf{inl}, \mathsf{curry}'\, \mathsf{inr} \,]$. distr, used in a later example, is the natural isomorphism $(A_1 + A_2) \times B \to (A_1 \times B) + (A_2 \times B)$; $\mathsf{distr} = \mathsf{uncurry} [\, \mathsf{curry}\, \mathsf{inl}, \mathsf{curry}\, \mathsf{inr} \,]$.)

Note that, computationally, course-of-value iterative function definitions are disastrous (they compute same function values over and over again); as *programs*, "memoizing" definitions combining simple iteration with tupling/projecting are preferable. But course-of-value iterative function definitions win in "declarative performance"; as *specifications*, they do better.

In the Mendler style, working with "courses" of result values is not a necessity for coding course-of-value iteration. There is a construction of a Mendler-style analog to the histo combinator that avoids this [UV97,UV00]. Let F be a functor. Say that a *F-m-cv-algebra* with carrier C is a function Φ on morphisms $A \to C \times F\,A$ returning morphisms $F\,A \to C$ for any A such that, if α, β are $(C \ltimes F)$-coalgebras, then

$$(\mathsf{id} \times F\,g) \circ \alpha = \beta \circ g \;\Rightarrow\; \Phi\,\alpha = \Phi\,\beta \circ F\,g$$

$$\begin{tikzcd} A \arrow[r, "\alpha"] \arrow[d, "g"'] & C \times F\,A \arrow[d, "\mathsf{id} \times F\,g"] \\ B \arrow[r, "\beta"] & C \times F\,B \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} F\,A \arrow[r, "\Phi\,\alpha"] \arrow[d, "F g"'] & C \arrow[d, equal] \\ F\,B \arrow[r, "\Phi\,\beta"] & C \end{tikzcd}$$

or, which is equivalent, that, if α is a $(C \ltimes F)$-coalgebra, then

$$\Phi\,\alpha = \Phi\,\mathsf{out}_{\_\ltimes F} \circ F\,[\!(\, \alpha \,)\!]_{\_\ltimes F}$$

$$\begin{tikzcd} F\,A \arrow[r, "\Phi\,\alpha"] \arrow[d, "F\,[\!(\alpha)\!]_{\_\ltimes F}"'] & C \arrow[d, equal] \\ F\,\nu(C \ltimes F) \arrow[r, "\Phi\,\mathsf{out}_{\_\ltimes F}"] & C \end{tikzcd}$$

(the above for $\beta := \mathsf{out}_{\_\ltimes F}$ only). Just as the side condition for m-algebras, this side condition on Φ is a parametricity condition and its satisfaction can be taken for granted, if Φ is definable.

A *homomorphism* between two F-m-cv-algebras Ψ, Φ with carriers D, C is a morphism $h : D \to C$ such that, if δ is a $(D \ltimes F)$-coalgebra and γ is a $(C \ltimes F)$-coalgebra, then

$$(h \times F\,\mathsf{id}) \circ \delta = \gamma \;\Rightarrow\; h \circ \Psi\,\delta = \Phi\,\gamma$$

$$\begin{tikzcd} & D \times F\,A \arrow[dd, "h \times F\,\mathsf{id}"] \\ A \arrow[ur, "\delta"] \arrow[dr, "\gamma"'] & \\ & C \times F\,A \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} & D \arrow[dd, "h"] \\ F\,A \arrow[ur, "\Psi\,\delta"] \arrow[dr, "\Phi\,\gamma"'] & \\ & C \end{tikzcd}$$

or, equivalently,

$$h \circ \Psi\,\mathsf{out}_{-\ltimes F} = \Phi\,((h \times F\,\mathsf{id}) \circ \mathsf{out}_{-\ltimes F})$$

$$\begin{tikzcd}
& D \arrow[dd, "h"] \\
F\,\nu(D \ltimes F) \arrow[ur, "\Psi\,\mathsf{out}_{-\ltimes F}"] \arrow[dr, "\Phi\,((h\times F\,\mathsf{id})\circ\mathsf{out}_{-\ltimes F})"'] & \\
& C
\end{tikzcd}$$

(the above condition for $\delta := \mathsf{out}_{-\ltimes F}$ only).

Assume now the existence of an initial F-algebra. For any F-m-cv-algebra Φ with carrier C, then a unique morphism $f : \mu F \to C$ exists such that, if m_1 is a morphism to μF, γ_1 is a morphism to C, and $m_2 = \gamma_2$ is an F-coalgebra, then

$$F\,m_1 \circ \gamma_2 = \mathsf{in}_F^{-1} \circ m_1 \;\;\wedge\;\; f \circ m_1 = \gamma_1 \;\;\Rightarrow\;\; f \circ (\mathsf{in}_F \circ F\,m_1) = \Phi\,\langle\, \gamma_1, \gamma_2 \,\rangle$$

$$\begin{tikzcd}
A \arrow[r, "\gamma_2"] \arrow[d, "m_1"'] & F\,A \arrow[d, "F\,m_1"] \\
\mu F \arrow[r, "\mathsf{in}_F^{-1}"] & F\,\mu F
\end{tikzcd}
\;\;\wedge\;\;
\begin{tikzcd}
& \mu F \arrow[dd, "f"] \\
A \arrow[ur, "m_1"] \arrow[dr, "\gamma_1"'] & \\
& C
\end{tikzcd}
\;\;\Rightarrow\;\;
\begin{tikzcd}
& \mu F \arrow[dd, "f"] \\
F\,A \arrow[ur, "\mathsf{in}_F \circ F\,m_1"] \arrow[dr, "\Phi\,\langle\, \gamma_1, \gamma_2 \,\rangle"'] & \\
& C
\end{tikzcd}$$

or, equivalently,

$$f \circ \mathsf{in}_F = \Phi\,\langle\, f, \mathsf{in}_F^{-1} \,\rangle$$

$$\begin{tikzcd}
& \mu F \arrow[dd, "f"] \\
F\,\mu F \arrow[ur, "\mathsf{in}_F"] \arrow[dr, "\Phi\,\langle\, f, \mathsf{in}_F^{-1} \,\rangle"'] & \\
& C
\end{tikzcd}$$

(the above condition for $\langle\, m_1, m_2 \,\rangle := \langle\, \mathsf{id}, \mathsf{in}_F^{-1} \,\rangle$ only). Let us call this f the *F-m-histomorphism* of Φ and denote it by $\{\!|\, \Phi \,|\!\}_F^{\mathsf{m}}$. It is best thought of as saying that, if γ_1 is the restriction of f to a subtype A of μF that (1) is downward closed (contains the predecessors of any value inside it), and (2) contains the predecessors of a given argument value, and γ_2 is the restriction of the uninjection function in_F^{-1} to A, then the result value of f for the argument value is equal to $\Phi\,\langle\, \gamma_1, \gamma_2 \,\rangle$ applied to it. (m_1 should be thought of as the "natural" embedding of A in μF.)

The important laws about m-histo combinator are the following. If Φ is an F-m-cv-algebra, then

$$f = \{\!|\, \Phi \,|\!\}_F^{\mathsf{m}} \;\;\Leftrightarrow\;\; f \circ \mathsf{in}_F = \Phi\,\langle\, f, \mathsf{in}_F^{-1} \,\rangle$$

If Φ, is an F-m-cv-algebra, then

$$\{\!|\, \Phi \,|\!\}_F^{\mathsf{m}} \circ \mathsf{in}_F = \Phi\,\langle\, \{\!|\, \Phi \,|\!\}_F^{\mathsf{m}}, \mathsf{in}_F^{-1} \,\rangle$$

$$\mathsf{id} = \{\!|\, \lambda m.\ \mathsf{in}_F \circ F\,(\mathsf{fst} \circ m) \,|\!\}_F^{\mathsf{m}}$$

If Ψ, Φ are F-m-cv-algebras, then

$$(\forall \delta, \gamma.\ (h \times F\,\mathsf{id}) \circ \delta = \gamma \;\;\Rightarrow\;\; h \circ \Psi\,\delta = \Phi\,\gamma) \;\;\Rightarrow\;\; h \circ \{\!|\, \Psi \,|\!\}_F^{\mathsf{m}} = \{\!|\, \Phi \,|\!\}_F^{\mathsf{m}}$$

Any m-catamorphism is a m-histomorphism. If Φ is an F-m-algebra, then

$$(\!|\, \Phi \,|\!)_F^{\mathsf{m}} = \{\!|\, \lambda\gamma.\ \Phi(\mathsf{fst} \circ \gamma) \,|\!\}_F^{\mathsf{m}}$$

The m-histo combinator reduces to the cata combinator. If Φ is an F-m-cv-algebra, then

$$\{\!|\, \Phi \,|\!\}_F^{\mathsf{m}} = \mathsf{fst} \circ \mathsf{out}_{-\ltimes F} \circ (\!|\, \mathsf{out}_{-\ltimes F}^{-1} \circ \langle\, \Phi\,\mathsf{out}_{-\ltimes F}, F\,\mathsf{id} \,\rangle \,|\!)_F$$

Example 4. Using the m-histo combinator, fibo and evens are course-of-value-iteratively definable as follows.

$$\mathsf{fibo} = \{\!|\ \lambda\gamma.\ \begin{bmatrix} \bar{\mathsf{0}}, \\ \begin{bmatrix} \bar{\mathsf{1}} \circ \mathsf{snd}, \\ \bar{+} \circ (\mathsf{id} \times (\mathsf{fst} \circ \gamma)) \end{bmatrix} \circ \mathsf{distl} \circ \gamma \end{bmatrix}\ |\!\}^{\mathsf{m}}_{\mathsf{N}}$$

$$\mathsf{evens} = \{\!|\ \lambda\gamma.\ \begin{bmatrix} \mathsf{nil}, \\ \begin{bmatrix} \mathsf{nil}, \\ \mathsf{cons} \circ (\mathsf{id} \times (\mathsf{fst} \circ \gamma)) \end{bmatrix} \circ (\mathsf{snd} \circ \gamma) \circ \mathsf{snd} \end{bmatrix}\ |\!\}^{\mathsf{m}}_{\mathsf{L}E}$$

These definitions look superficially similar to the definitions using the histo combinator, but how they work is quite different.

Any histomorphism is a m-histomorphism and the other way around. If ϕ is an F-cv-algebra, then

$$\{\!|\ \phi\ |\!\}_F = \{\!|\ \lambda\gamma.\ \phi \circ F\,[\![\ \gamma\ ]\!]_{-\ltimes F}\ |\!\}^{\mathsf{m}}_F$$

If Φ is an F-m-cv-algebra, then

$$\{\!|\ \Phi\ |\!\}^{\mathsf{m}}_F = \{\!|\ \Phi\,\mathsf{out}_{-\ltimes F}\ |\!\}_F$$

4 Multimorphisms

Another recursion scheme for which a simple conventional-style combinator cannot, but a Mendler-style combinator can be defined is simultaneous iteration or multi-iteration.

Similarly to the case of course-of-value iteration, constructing a conventional-style combinator for simultaneous iteration requires an introduction of an intermediate structure. The solution of [FSZ94,HIT97,Uus99] is the following.

Let F_1, F_2, and F be functors and let $\tau : F_1\,C_1 \times F_2\,C_2 \to F\,(C_1 \times C_2)$ be natural in C_1, C_2. Assume that initial algebras exist for F_1, F_2. Then, for any $\phi : F\,C \to C$, there is exactly one morphism $f : \mu F_1 \times \mu F_2 \to C$ such that

$$f \circ (\mathsf{in}_{F_1} \times \mathsf{in}_{F_2}) = \phi \circ F\,f \circ \tau$$

$$\begin{tikzcd}
F_1\,\mu F_1 \times F_2\,\mu F_2 \arrow[rr, "\mathsf{in}_{F_1} \times \mathsf{in}_{F_2}"] \arrow[d, "\tau"'] & & \mu F_1 \times \mu F_2 \arrow[dd, "f"] \\
F\,(\mu F_1 \times \mu F_2) \arrow[d, "F\,f"'] & & \\
F\,C \arrow[rr, "\phi"'] & & C
\end{tikzcd}$$

Call this f the $F_1, F_2; F$-τ-*multimorphism* of ϕ and denote it by $(\!|\ \phi\ |\!)^{\tau}_{F_1,F_2;F}$.

The multi combinator obeys the following laws. If ϕ is an F-algebra, then

$$f = (\!|\ \phi\ |\!)^{\tau}_{F_1,F_2;F} \iff f \circ (\mathsf{in}_{F_1} \times \mathsf{in}_{F_2}) = \phi \circ F\,f \circ \tau$$

If ϕ is an F-algebra, then

$$(\!|\ \phi\ |\!)^{\tau}_{F_1,F_2;F} \circ (\mathsf{in}_{F_1} \times \mathsf{in}_{F_2}) = \phi \circ F\,(\!|\ \phi\ |\!)^{\tau}_{F_1,F_2;F} \circ \tau$$

If ψ, ϕ are F-algebras, then

$$h \circ \psi = \phi \circ F\,h \;\Rightarrow\; h \circ (\!|\ \psi\ |\!)^{\tau}_{F_1,F_2;F} = (\!|\ \phi\ |\!)^{\tau}_{F_1,F_2;F}$$

If ψ_1 is an F_1-algebra, ψ_2 is an F_2-algebra, and ϕ is an F-algebra, then

$$h \circ (\psi_1 \times \psi_2) = \phi \circ F\,h \circ \tau \;\Rightarrow\; h \circ ((\!|\ \psi_1\ |\!)_{F_1} \times (\!|\ \psi_2\ |\!)_{F_2}) = (\!|\ \phi\ |\!)^{\tau}_{F_1,F_2;F}$$

The multi combinator is reducible to the cata combinator. If ϕ is an F-algebra, then

$$
\begin{aligned}
&(\!|\, \phi \,|\!)^{\tau}_{F_1,F_2;F} \\
&= \mathsf{uncurry}(\!|\, (\mathsf{in}_{F_2}^{-1} \to \phi) \circ \mathsf{curry}(F\,(\mathsf{uncurry}\,\mathsf{id}) \circ \tau) \,|\!)_{F_1} \\
&= \mathsf{uncurry}'(\!|\, (\mathsf{in}_{F_1}^{-1} \to \phi) \circ \mathsf{curry}'(F\,(\mathsf{uncurry}'\,\mathsf{id}) \circ \tau) \,|\!)_{F_2}
\end{aligned}
$$

Example 5. Define $\mathsf{NN}C = (1+1)+(1+C)$ and

$$\mathsf{nn} = ((\mathsf{fst}+\mathsf{fst})\circ\mathsf{distl}) + ((\mathsf{snd}+\mathsf{id})\circ\mathsf{distl})\circ\mathsf{distr}$$

The less-than function $\mathsf{ltnat} : \mathsf{Nat} \to \mathsf{Bool}$ on naturals is defined as a $\mathsf{N},\mathsf{N};\mathsf{NN}$-$\mathsf{nn}$-multimorphism as follows.

$$\mathsf{ltnat} = (\!|\, [\,[\,[\,\mathsf{false},\mathsf{true}\,],[\,\mathsf{false},\mathsf{id}\,]\,]\,|\!)^{\mathsf{nn}}_{\mathsf{N},\mathsf{N};\mathsf{NN}}$$

Define $\mathsf{NL}E\,C = (1+1)+(1+E\times C)$ and

$$\mathsf{nl} = ((\mathsf{fst}+\mathsf{fst})\circ\mathsf{distl}) + ((\mathsf{snd}+\langle\,\mathsf{fst}\circ\mathsf{snd},\langle\,\mathsf{fst},\mathsf{snd}\circ\mathsf{snd}\,\rangle\,\rangle)\circ\mathsf{distl})\circ\mathsf{distr}$$

Then the function $\mathsf{take} : \mathsf{Nat}\times\mathsf{List}E \to \mathsf{List}E$ that selects a given number of first elements from a given list is defined as a $\mathsf{N},\mathsf{L}E;\mathsf{NL}E$-$\mathsf{nl}$-multimorphism in the following way.

$$\mathsf{take} = (\!|\, [\,[\,[\,\mathsf{nil},\mathsf{nil}\,],[\,\mathsf{nil},\mathsf{cons}\,]\,]\,|\!)^{\mathsf{nl}}_{\mathsf{N},\mathsf{L}E;\mathsf{NL}E}$$

In the Mendler style, resorting to any intermediate structure is unnecessary. Let F_1, F_2 be functors. The concept of m-algebra is readily generalizable to apply to a tuple of functors, not just one functor. Say that a F_1,F_2-m-algebra with carrier C is a function Φ on morphisms $A_1\times A_2\to C$ returning morphisms $F_1\,A_1\times F_2\,A_2\to C$ for any A_1,A_2 such that, if α, β are morphisms from products to C, then

$$\alpha = \beta\circ(g_1\times g_2) \;\Rightarrow\; \Phi\,\alpha = \Phi\,\beta\circ(F_1\,g_1\times F_2\,g_2)$$

$$
\begin{tikzcd}
A_1\times A_2 \arrow[r, "\alpha"] \arrow[d, "g_1\times g_2"'] & C \arrow[d, equal] \\
B_1\times B_2 \arrow[r, "\beta"] & C
\end{tikzcd}
\quad\Rightarrow\quad
\begin{tikzcd}
F_1\,A_1\times F_2\,A_2 \arrow[r, "\Phi\,\alpha"] \arrow[d, "F_1\,g_1\times F_2\,g_2"'] & C \arrow[d, equal] \\
F_1\,B_1\times F_2\,B_2 \arrow[r, "\Phi\,\beta"] & C
\end{tikzcd}
$$

Assume that initial algebras exist for F_1, F_2. Then, for any F_1,F_2-m-algebra Φ with carrier C, there exists a unique morphism $f : \mu F_1\times\mu F_2\to C$ such that, if m_1 is a morphism to μF_1 and m_2 is a morphism to μF_2, and γ is a morphism to C, then

$$f\circ(m_1\times m_2) = \gamma \;\Rightarrow\; f\circ((\mathsf{in}_{F_1}\circ F_1\,m_1)\times(\mathsf{in}_{F_2}\circ F_2\,m_2)) = \Phi\,\gamma$$

$$
\begin{tikzcd}
 & \mu F_1\times\mu F_2 \arrow[d, "f"] \\
A_1\times A_2 \arrow[ur, "m_1\times m_2"] \arrow[r, "\gamma"'] & C
\end{tikzcd}
\quad\Rightarrow\quad
\begin{tikzcd}
 & \mu F_1\times\mu F_2 \arrow[d, "f"] \\
F_1\,A_1\times F_2\,A_2 \arrow[ur, "(\mathsf{in}_{F_1}\circ F_1\,m_1)\times(\mathsf{in}_{F_2}\circ F_2\,m_2)"] \arrow[r, "\Phi\,\gamma"'] & C
\end{tikzcd}
$$

or, equivalently,

$$f\circ(\mathsf{in}_{F_1}\times\mathsf{in}_{F_2}) = \Phi\,f$$

$$
\begin{tikzcd}
 & \mu F_1\times\mu F_2 \arrow[d, "f"] \\
F_1\,\mu F_1\times F_2\,\mu F_2 \arrow[ur, "\mathsf{in}_{F_1}\times\mathsf{in}_{F_2}"] \arrow[r, "\Phi\,f"'] & C
\end{tikzcd}
$$

(the above condition for $m_1 := \mathsf{id}$, $m_2 := \mathsf{id}$ only). Call this f the *F_1,F_2-m-multimorphism* of Φ and denote it by $(\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2}$.

The m-multi combinator obeys the following laws. If Φ is an F_1, F_2-m-algebra, then

$$f = (\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2} \;\Leftrightarrow\; f \circ (\mathsf{in}_{F_1} \times \mathsf{in}_{F_2}) = \Phi\, f$$

If Φ is an F_1, F_2-m-algebra, then

$$(\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2} \circ (\mathsf{in}_{F_1} \times \mathsf{in}_{F_2}) = \Phi\, (\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2}$$

If Ψ, Φ are F_1, F_2-m-algebras, then

$$(\forall \delta, \gamma.\ h \circ \delta = \gamma \;\Rightarrow\; h \circ \Psi\, \delta = \Phi\, \gamma) \;\Rightarrow\; h \circ (\!|\, \Psi \,|\!)^{\mathsf{m}}_{F_1,F_2} = (\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2}$$

If Ψ_1 is an F_1-m-algebra, Ψ_2 is an F_2-m-algebra, and Φ is an F_1, F_2-m-algebra, then

$$(\forall \delta_1, \delta_2, \gamma.\ h \circ (\delta_1 \times \delta_2) = \gamma \;\Rightarrow\; h \circ (\Psi_1\, \delta_1 \times \Psi_2\, \delta_2) = \Phi\, \gamma) \;\Rightarrow\; h \circ ((\!|\, \Psi_1 \,|\!)^{\mathsf{m}}_{F_1} \times (\!|\, \Psi_2 \,|\!)^{\mathsf{m}}_{F_2}) = (\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2}$$

The reduction of m-multi to cata is the following. If Φ is an F_1, F_2-algebra, then

$$\begin{aligned}
&(\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2} \\
&= \mathsf{uncurry}(\!|\, (\mathsf{in}_{F_2}^{-1} \to \mathsf{id}) \circ \mathsf{curry}(\Phi(\mathsf{uncurry}\,\mathsf{id})) \,|\!)_{F_1} \\
&= \mathsf{uncurry}'(\!|\, (\mathsf{in}_{F_1}^{-1} \to \mathsf{id}) \circ \mathsf{curry}'(\Phi(\mathsf{uncurry}'\,\mathsf{id})) \,|\!)_{F_2}
\end{aligned}$$

Example 6. Recast using the m-multi combinator, the natural, i.e., simultaneous-iterative, definitions of ltnat and take take the following form.

$$\mathsf{ltnat} = (\!|\, \lambda\gamma.\ \begin{bmatrix} \begin{bmatrix} \mathsf{false} \circ \mathsf{fst}, \\ \mathsf{true} \circ \mathsf{fst} \end{bmatrix} \circ \mathsf{distl}, \\ \begin{bmatrix} \mathsf{false} \circ \mathsf{snd}, \\ \gamma \end{bmatrix} \circ \mathsf{distl} \end{bmatrix} \circ \mathsf{distr} \,|\!)^{\mathsf{m}}_{\mathsf{N},\mathsf{N}}$$

$$\mathsf{take} = (\!|\, \lambda\gamma.\ \begin{bmatrix} \begin{bmatrix} \mathsf{nil} \circ \mathsf{fst}, \\ \mathsf{nil} \circ \mathsf{fst} \end{bmatrix} \circ \mathsf{distl}, \\ \begin{bmatrix} \mathsf{nil} \circ \mathsf{snd}, \\ \mathsf{cons} \circ \langle\, \mathsf{fst} \circ \mathsf{snd}, \gamma \circ \langle\, \mathsf{fst}, \mathsf{snd} \circ \mathsf{snd} \,\rangle \,\rangle \end{bmatrix} \circ \mathsf{distl} \end{bmatrix} \circ \mathsf{distr} \,|\!)^{\mathsf{m}}_{\mathsf{N},\mathsf{L}E}$$

Any multimorphism is an m-multimorphism. If ϕ is an F-algebra, then

$$(\!|\, \phi \,|\!)^{\tau}_{F_1,F_2;F} = (\!|\, \lambda\gamma.\ \phi \circ F\, \gamma \circ \tau \,|\!)^{\mathsf{m}}_{F_1,F_2}$$

The opposite claim, that any m-multimorphism is a multimorphism, holds, if certain tensors and coends can be assumed to exist. If Φ is an F_1, F_2-m-algebra, then

$$(\!|\, \Phi \,|\!)^{\mathsf{m}}_{F_1,F_2} = (\!|\, [\, \Phi \,] \,|\!)^{\tau}_{F_1,F_2;F}$$

where $F\,C = \int^{A_1,A_2} \mathrm{hom}(A_1 \times A_2, C) \otimes (F_1\, A_1 \times F_2\, A_2)$ and $\tau = \mathsf{In}\,(\mathsf{id} \times \mathsf{id})$. ($\mathsf{In}$ is a function parametric in A_1, A_2 on morphisms $A_1 \times A_2 \to C$ returning morphisms $F_1\, A_1 \times F_2\, A_2 \to F\, C$, and $[\, \Theta \,]$ is, for any function Θ parametric in A_1, A_2 on morphisms $A_1 \times A_2 \to C$ returning morphisms $F_1\, A_1 \times F_2\, A_2 \to C'$, the unique morphism $f : F\, C \to C'$ such that $f \circ \mathsf{In}\, \alpha = \Theta\, \alpha$.)

5 Conclusion

This paper aimed to demostrate that the key to a reasonable coding of a (not entirely trivial) terminating recursion scheme as a combinator may sometimes lie in the Mendler style. We find that course-of-value iteration and simultaneous iteration are good witnesses to this claim. The claim is also supported by the fact that the same idea has recently been applied (although not under the same name) in a number of contexts. Bird and Paterson's ho-algebras [BP99] for the analysis of nested, or irregular, datatypes, for instance, can be seen as an application of Mendler's technique. In his study of the principle of guarded-by-destructors recursion (originally guarded-by-constructors corecursion), Giménez switched from a very syntactically defined combinator [Gim95] to a more semantically motivated one [Gim98], which seems to be very closely related to what we consider to be the prime Mendler-style encoding of course-of-value primitive recursion. He however works in a system with subtyping, which means that certain castings between types that we have to do explicitly in our framework, happen silently in his setting. The exact connection has to be spelled out.

Within our own setting, we have to clarify the exact interrelation of F-(m-)algebras and F-(m-)cv-algebras. F-m-cv-algebras are particularly interesting also because of their equivalence to m-algebras for a mixed-variant functor.

Acknowledgements

We are grateful to Gilles Barthe for very helpful discussions and insight into Eduardo Giménez', his, and Maria João Frade's current work.

The work reported here was partially supported by the Estonian Science Foundation under grant no. 4155. The first author also received support from the Portuguese Foundation for Science and Technology under grant no. PRAXIS XXI/C/EEI/14172/98.

References

[BP99] R Bird, R Paterson. Generalized folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.

[FSZ94] L Fegaras, T Sheard, and T Zhou. Improving programs which recurse over multiple inductive structures. In *Proceedings ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'94, Orlando, FL, USA, June 1994*, pp 21–32. ACM Press, New York, 1994.

[Geu92] H Geuvers. Inductive and coinductive types with iteration and recursion. In B Nordström, K Pettersson, and G Plotkin, eds, *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*, pp 193–217. Dept of Computing Science, Chalmers Univ of Technology and Göteborg Univ, 1992. ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z.

[Gim95] E Giménez. Codifying guarded definitions with recursion schemes. In P Dybjer and B Nordström, eds, *Selected Papers 2nd Int Workshop on Types for Proofs and Programs, TYPES'94, Båstad, Sweden, 6–10 June 1994*, vol 996 of *Lecture Notes in Computer Science*, pp 39–59. Springer-Verlag, Berlin, 1995.

[Gim98] E Giménez. Structural recursive definitions in type theory. In K G Larsen, S Skyum, and G Winskel, eds, *Proceedings 25th Int Coll on Automata, Languages and Programming, ICALP'98, Aalborg, Denmark, 13-17 July 1998*, vol 1443 of *Lecture Notes in Computer Science*, pp 397–408. Springer-Verlag, Berlin, 1998.

[HIT97] Z Hu, H Iwasaki, and M Takeichi. An extension of the acid rain theorem. In T Ida, A Ohori, and M Takeichi, eds, *Proceedings 2nd Fuji Int Workshop on Functional and Logic Programming, Shonan Village Center, Japan, 1–4 Nov 1996*, pp 91–105. World Scientific, Singapore, 1997.

[Lei90] D Leivant. Contracting proofs to programs. In P Odifreddi, ed, *Logic and Computer Science*, vol 31 of *APIC Studies in Data Processing*, pp 279–327. Academic Press, London, 1990.

[Mal90] G Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[Mat98] R Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. PhD thesis, Fachbereich Mathematik, Ludwig-Maximilians-Univ München, 1998.

[Mat00] R Matthes. Tarski's fixed-point theorem and lambda calculi with monotone inductive types. In B Löwe and F Rudolph, eds, *Refereed Papers of Research Coll on Foundations of the Formal Sciences, Berlin, Germany, 7–9 May 1999*, pp 91–112. Kluwer Acad Publ, Dordrecht, 2000.

[Mee92] L Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[Men87] N P Mendler. Recursive types and type constraints in second-order lambda-calculus. In *Proceedings 2nd Annual IEEE Symp on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987*, pp 30–36. IEEE CS Press, Washington, DC, 1987.

[Men91] N P Mendler. Inductive types and type constraints in the second-order lambda-calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[Spł93] Z Spławski. *Proof-Theoretic Approach to Inductive Definitions in ML-Like Programming Languages versus Second-Order Lambda Calculus.* PhD thesis, Wrocław Univ, 1993.

[SU99] Z Spławski and P Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Proceedings 4th ACM SIGPLAN Int Conf on Functional Programming, ICFP'99, Paris, France, 27–29 Sept 1999*, pp 102–113. ACM Press, New York, 1999.

[Uus98] T Uustalu. *Natural Deduction for Intuitionistic Least and Greatest Fixedpoint Logics, with an Application to Program Construction.* PhD thesis, Dissertation TRITA-IT AVH 98:03, Dept of Teleinformatics, Royal Inst of Technology, Stockholm, May 1998.

[Uus99] T Uustalu. Multi-(co)iteration, categorically. In J Penjam, ed, *Proceedings 6th Fenno-Ugric Symp on Software Technology, FUSST'99, Sagadi, Estonia, 19–21 Aug 1999*, Report CS 104/99, Inst of Cybern, Tallinn Technical Univ, pp 259–267. August 1999.

[UV97] T Uustalu and V Vene. A cube of proof systems for the intuitionistic predicate μ, ν-logic. In M Haveraaen and O Owe, eds, *Selected Papers 8th Nordic Workshop on Programming Theory, NPWT'96, Oslo, Norway, 4–6 Dec 1996*, Research Report 248, Dept of Informatics, Univ of Oslo, pp 237–246. May 1997.

[UV99a] T Uustalu and V Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *INFORMATICA*, 10(1):5–26, 1999.

[UV99b] T Uustalu and V Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, 1999.

[UV00] T Uustalu and V Vene. Least and greatest fixedpoints in intuitionistic natural deduction. *Theoretical Computer Science*, to appear (accepted March 2000).

[Wad89] P Wadler Theorems for free! In *Proceedings 4th Int Conf on Funct Prog and Computer Arch, FPCA'89, London, UK, 11–13 Sept 1989*, pp 347–359. ACM Press, New York, 1989.

A Initial algebras and catamorphisms, in Haskell

In Haskell, like in type theory, functors arise from the association of a morphism mapping to an object mapping. A functor in Haskell is a type constructor from the class `Functor` defined in the Haskell Prelude as follows:

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

The type constructor `f`, in itself, is the object mapping part of a functor. The morphism mapping is the function `fmap`. The class definition forces `fmap` to have the correct typing, but cannot force it to preserve identities and composition, so at each time the programmer defines the `fmap` function for a particular type constructor `f`, it is his responsibility to ensure that these conditions are met.

An initial algebra is "created" for any functor `f` by a recursive newtype definition producing a data constructor `In` for the initial algebra and a type `Mu f` for its carrier. From this basis, the inverse of `In` is definable directly.

```
> newtype Mu f = In (f (Mu f))

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

That `In` is an algebra is stated in the definition of `Mu f`. Its initiality follows from the existence of a matching cata combinator. This function, `cata`, is definable by turning the cata computation rule into a recursive function definition. Note that the definition uses `fmap`.

```
> cata :: Functor f => (f c -> c) -> Mu f -> c
> cata phi (In x) = phi (fmap (cata phi) x)
```

The type of natural numbers, the number zero and the successor function are defined as follows.

```
> data N c = Z | S c
> instance Functor N  where
>   fmap g Z     = Z
>   fmap g (S c) = S (g c)

> type Nat = Mu N

> zer :: Nat
> zer = In Z

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

The type of lists, the empty list and the cons function are defined similarly, for any element type.

```
> data L e c  = N | C e c
> instance Functor (L e)  where
>   fmap g N       = N
>   fmap g (C e c) = C e (g c)

> type List e = Mu (L e)

> nil :: List e
> nil = In N

> cons :: e -> List e -> List e
> cons e es = In (C e es)
```

Using `cata`, the function converting a natural to an integer is defined as follows.

```
> n2i' :: Nat -> Int
> n2i' = cata phi
>   where phi Z         = 0                    -- n2i^_Z
>         phi (S n2i_n) = 1 + n2i_n            -- n2i^_(S n)
```

Note that `phi` is a function computing the result value of the conversion function for an unspecified argument value from the result value of the conversion function, `n2i_n`, for the predecessor, if it exists.

The m-cata combinator is defined as the cata combinator, by making its computation rule a definition.

```
> cataM :: (forall a. (a -> c) -> f a -> c) -> Mu f -> c
> cataM phiM (In x) = phiM (cataM phiM) x
```

The type of `mcata` involves a non-top-level universal quantification, permitted in the version of Haskell with "rank-2 type signatures". The quantified type, the expected type of `phiM`, is the type of a m-algebra. The parametricity of `phiM` is a "theorem for free", provided that `phiM` is a defined function.

With `cataM`, the function for the naturals-to-integers conversion is defined as follows.

```
> n2i'' :: Nat -> Int
> n2i'' = cataM phiM
>   where phiM n2i Z     = 0                   -- n2i^ Z
>         phiM n2i (S n) = 1 + n2i n           -- n2i^ (S n)
```

The function `phiM` computes the result value of the conversion function from an explicitly given argument value and a restriction, `n2i`, of the conversion function to a domain containing at least the predecessor, `n`, if it exists.

B Histomorphisms, in Haskell

In order to define the histo combinator, we first have to define the type `Strf c f` of `f`-streams for any given functor `f` and element type `c`, together with its destructors, `hdf` and `tlf`. The latter two functions form a final coalgebra for the functor sending a type `a` to the type `(c, f a)`. The carrier is `Strf c f`.

```
> data Strf c f = Consf c (f (Strf c f))

> hdf :: Strf c f -> c
> hdf (Consf c _) = c

> tlf :: Strf c f -> f (Strf c f)
> tlf (Consf _ fcs) = fcs
```

We also need the generator function, `genStrf`, for `f`-streams. This is the corresponding ana combinator.

```
> genStrf :: Functor f => (a -> c) -> (a -> f a) -> a -> Strf c f
> genStrf gamma1 gamma2 a =
>          Consf (gamma1 a) (fmap (genStrf gamma1 gamma2) (gamma2 a))
```

These preparations made, the histo combinator is now definable as follows, guided by the computation rule.

```
> histo :: Functor f => (f (Strf c f) -> c) -> Mu f -> c
> histo phi (In x) = phi (fmap (genStrf (histo phi) unIn) x)
```

This definition, however, is computationally unfeasible. But luckily enough, we can, at will, switch to another, very reasonable definition, which is based on the histo-to-cata reduction.

```
> -- histo phi = hdf . cata (\fcs -> Consf (phi fcs) fcs)
```

Using `histo`, the Fibonacci function is definable as follows.

```
> fibo' :: Nat -> Int
> fibo' = histo phi
>   where phi Z        = 0                              -- fibo^_Z
>         phi (S (Consf fibo_n fiboHist_n))             -- fibo^_(S n)
>                      = case fiboHist_n of
>                          Z          -> 1
>                          S (Consf fibo_n' _) -> fibo_n + fibo_n'
```

Here, `phi` computes the result value of Fibonacci for an unspecified argument value from both the result value of Fibonacci, `fibo_n`, for the predecessor and the "history" of "earlier" result values of Fibonacci, `fiboHist_n`, if the predecessor exists.

The function taking from a given list every second element is definable in a similar fashion.

```
> evens' :: List e -> List e
> evens' = histo phi
>   where phi N           = nil                         -- evens^_nil
>         phi (C _ (Consf _ evensHist_es))              -- evens^_(C _ es)
>                         = case evensHist_es of
>                             N            -> nil
>                             C e' (Consf evens_es' _) -> cons e' evens_es'
```

In sharp contrast to what we saw when paving the way for defining the histo combinator, no preparations whatsoever are needed in order to define the m-histo combinator.

```
> histoM :: (forall a. (a -> c) -> (a -> f a) -> f a -> c) -> Mu f -> c
> histoM phiM (In x) = phiM (histoM phiM) unIn x
```

Using `histoM`, the Fibonacci function is defined in the following way.

```
> fibo'' :: Nat -> Int
> fibo'' = histoM phiM
>   where phiM fibo unIn Z     = 0                      -- fibo^ Z
>         phiM fibo unIn (S n) = case unIn n of         -- fibo^ (S n)
>                                  Z  -> 1
>                                  S n' -> fibo n + fibo n'
```

Here, `phiM` computes the result value of Fibonacci from an explicitly given argument and the restrictions, `fibo` and `unIn`, of the Fibonacci function and the uninjection function (the other, globally defined and overriden `unIn`), to a domain that is downward closed and contains at least the predecessor, `n`, if it exists.

The function for taking from a list every second element is definable as follows.

```
> evens'' :: List e -> List e
> evens'' = histoM phiM
>   where phiM evens unIn N        = nil                -- evens^ N
>         phiM evens unIn (C _ es) = case unIn es of    -- evens^ (C _ es)
>                                      N      -> nil
>                                      C e' es'  -> cons e' (evens es')
```

C Multimorphisms, in Haskell

Defining the multi combinator, again, requires extra definitions to be made. For any given two functors `f1`, `f2`, we have to have a means to associate a function `tau` to a third functor `f`, if `f` is to be the "zip" of `f1`, `f2`. For this purpose, we introduce a type class `Multifunc`.

```
> class Functor f => MultiFunc f1 f2 f where
>   tau :: (f1 a1, f2 a2) -> f (a1, a2)
```

A rewrite into Haskell of the multi computation rule gives a definition for the multi combinator.

```
> multi :: MultiFunc f1 f2 f => (f c -> c) -> (Mu f1, Mu f2) -> c
> multi phi (In x1, In x2) = phi (fmap (multi phi) (tau (x1, x2)))
```

In order to define the less-than function for naturals using `multi`, we have to first define a type constructor `NN`, make it an instance of `Functor` and then make `N`, `N`, `NN` together an instance of `MultiFunc`.

```
> data NN c = ZZ | ZS | SZ | SS c
> instance Functor NN  where
>   fmap g ZZ     = ZZ
>   fmap g ZS     = ZS
>   fmap g SZ     = SZ
>   fmap g (SS c) = SS (g c)
> instance MultiFunc N N NN  where
>   tau (Z, Z)       = ZZ
>   tau (Z, S _)     = ZS
>   tau (S _, Z)     = SZ
>   tau (S c1, S c2) = SS (c1, c2)
```

These preparations made, giving the definition for the function itself is an easy task.

```
> lt' :: (Nat, Nat) -> Bool
> lt' = multi phi
>   where phi ZZ             = False           -- lt^_ZZ
>         phi ZS             = True            -- lt^_ZS
>         phi SZ             = False           -- lt^_SZ
>         phi (SS lt_n1_n2)  = lt_n1_n2        -- lt^_(SS n1 n2)
```

Defining the take function for lists proceeds similarly.

```
> data NL e c = ZN | ZC | SN | SC e c
> instance Functor (NL e) where
>   fmap g ZN   = ZN
>   fmap g ZC   = ZC
>   fmap g SN   = SN
>   fmap g (SC e c) = SC e (g c)
> instance MultiFunc N (L e) (NL e) where
>   tau (Z, N)     = ZN
>   tau (Z, C _ _)   = ZC
>   tau (S _, N)     = SN
>   tau (S c1, C e c2) = SC e (c1, c2)

> take' :: (Nat, List e) -> List e
> take' = multi phi
>   where phi ZN               = nil                    -- take^_ZN
>         phi ZC               = nil                    -- take^_ZC
>         phi SN               = nil                    -- take^_SN
>         phi (SC e take_n_es) = cons e take_n_es       -- take^_(SC e (n, es))
```

The m-multi combinator is definable right away.

```
> multiM :: (forall a1 a2. ((a1, a2) -> c) -> (f1 a1, f2 a2) -> c)
>      -> (Mu f1, Mu f2) -> c
> multiM phiM (In x1, In x2) = phiM (multiM phiM) (x1, x2)
```

Using `multiM`, the less-than function is defined as follows.

```
> lt'' :: (Nat, Nat) -> Bool
> lt'' = multiM phiM
>   where phiM lt (Z, Z)       = False                  -- lt^ (Z, Z)
>         phiM lt (Z, S _)     = True                   -- lt^ (Z, S _)
>         phiM lt (S _, Z)     = False                  -- lt^ (S _, Z)
>         phiM lt (S n1, S n2) = lt (n1, n2)            -- lt^ (S n1, S n2)
```

The take function is defined analogously as follows.

```
> take'' :: (Nat, List e) -> List e
> take'' = multiM phiM
>   where phiM take (Z, N)        = nil                 -- take^ (Z, N)
>         phiM take (Z, C _ _)    = nil                 -- take^ (Z, C _ _)
>         phiM take (S _, N)      = nil                 -- take^ (S _, N)
>         phiM take (S n, C e es) =                     -- take^ (S n, C e es)
>                                    cons e (take (n, es))
```