Nordic Journal of Computing 8(2001), 366–390.

RECURSION SCHEMES FROM COMONADS

TARMO UUSTALU*
Dep. de Informática, Universidade do Minho
Campus de Gualtar, P-4710-057 Braga, Portugal
tarmo@di.uminho.pt

VARMO VENE
Inst. of Computer Science, University of Tartu
J. Liivi 2, EE-50409 Tartu, Estonia
varmo@cs.ut.ee

ALBERTO PARDO
Instituto de Computación, Universidad de la República
Julio Herrera y Reissig 565, 11300 Montevideo, Uruguay
pardo@fing.edu.uy

Abstract. Within the setting of the categorical approach to total functional programming, we introduce a "many-in-one" recursion scheme that neatly unifies a variety of seemingly diverging strengthenings of the basic recursion scheme of iteration. The new scheme is doubly generic: in addition to being parametric in a functor capturing the signature of an inductive type, it is also parametric in a comonad and a distributive law (of the functor over the comonad) that together encode the recursive call pattern of a particular recursion scheme for this inductive type. Specializations of the scheme for particular comonads and distributive laws include (simple) iteration and mild generalizations of primitive recursion and course-of-value iteration.

CR Classification: D.1.1, D.3.3, F.3.3

Key words: inductive types, iteration, recursion schemes, initial functor-algebras, comonads, distributive laws, functional programming, genericity, category-theoretic

1. Introduction

Generic programming means programming using non-traditional forms of polymorphism. Its main expected benefit is enhanced code reusability because of more modular program structuring. In functional programming, the mainstream developments around genericity are focussing on polymorphism in type constructors; for a picture of the state-of-the-art, see [Backhouse *et al.* 1999].

In this paper, we are interested in total functional programming and study in the spirit of constructive algorithmics [Fokkinga 1992, Bird and

*On leave from Inst. of Cybernetics, Tallinn Technical University, Akadeemia tee 21, EE-12618 Tallinn, Estonia, tarmo@cs.ioc.ee.

Received December 29, 2000; revised May 16, 2001; accepted August 14, 2001.

de Moor 1997]. By total functional programming, we mean functional programming under a discipline that only allows total functions to be coded; cf. D. A. Turner's strong functional programming [Turner 1995] or programming in typed lambda calculi. We are concerned with genericity in recursion schemes, i.e., schemes of circular definition of functions with inductive domain. (Total functional programming implies a separation between inductive and coinductive types, so recursion is distinct from corecursion. The latter word is used to refer to circular definition of functions with coinductive codomain.)

Many recursion schemes are generic in the sense that they are definable and work uniformly for any inductive type. This means genericity also in accordance with the programming jargon: the combinators derived from these schemes are polymorphic in a functor inducing an inductive type. The most fundamental and useful such schemes are iteration and primitive recursion. In the constructive algorithmics community, the corresponding combinators are known as the catamorphism and paramorphism combinator. Some other generic schemes that are more powerful in terms of direct expressive power, e.g., the course-of-value strengthening of simple iteration, are nearly as fundamental and useful and are not too sophisticated in their generic formulation. Generic schemes that go considerably further, however, tend to be of too limited utility and overly sophisticated. To try to cover every imaginable recursion scheme by an individual library program is therefore probably not a good idea.

In this paper, we test an alternative idea of supporting different recursion schemes through one library program. We suggest a "many-in-one" recursion scheme that neatly unifies a variety of seemingly diverging strengthenings of iteration. This new scheme is doubly generic: in addition to being parametric in a functor capturing the signature of an inductive type, it is also parametric in another functor with a certain superstructure, more exactly, a comonad and a distributive law that together encode the recursive call pattern of a particular recursion scheme for this inductive type. Specializations of the scheme for particular comonads and distributive laws include (simple) iteration, a mild generalization of primitive recursion, and a comparable generalization of course-of-value iteration.

The paper is organized as follows. In Section 2, we recall some generalities about the categorical approach to functional programming and fix some notation. Section 3 contains a presentation of the standard initial functor-algebra treatment of inductive types and iteration, the starting point for all theory and examples of the paper. In Section 4, we present the concept of comonad and, in Section 5, we study the concept of comonad with a distributive law of a given functor over it. The new recursion scheme is introduced in Section 6, where we also present its key properties and show how it instantiates to some particular recursion schemes. In Section 7, we demonstrate a Haskell implementation of the schemes and the examples considered. In Sections 8 and 9, we briefly comment on the related work and conclude.

The paper assumes from the reader a minimal background in category theory. Concepts such as functor, natural transformation, product, initiality will not be defined; the reader may consult, e.g., [Barr and Wells 1990], for definitions, if necessary. Concepts such as functor-algebra, comonad, distributive law will be defined. For the appreciation of Section 7, some acquaintance with the functional programming language Haskell is beneficial, but not strictly necessary. A good introduction to functional programming in Haskell is [Bird 1998].

2. Types and functions in categories

In constructive algorithmics, it is common to approach functional programming with the methods of category theory. Good presentations of the methodology practised are [Fokkinga 1992, Bird and de Moor 1997]. We shall only recall a few generalities and proceed then with our concrete assumptions and notation.

In the category-theoretic approach to functional programming, the realm of programming is identified with some fixed category $\mathcal{C}$. The types are the objects and the functions are the morphisms of $\mathcal{C}$. Further, type constructors are object mappings etc. The role of values living in a given type A is assigned to morphisms $a : 1 \to A$. The motivation comes from $\mathcal{C} = \mathbf{Set}$, the category of sets and set-theoretic (i.e., total) functions. There, an object (set) A is isomorphic to the collection of all morphisms (functions) from 1 to A. The application of a function $f : A \to B$ to a value a of type A is identified with the composition $f \circ a : 1 \to B$. From the programming perspective, this approach of lifting elements to functions leads to a point-free style of programming, where functions are described exclusively in terms of function composition.

In the case of total functional programming, many things are simple compared to partial functional programming. In particular, product types are identifiable with (categorical) products, sum types are coproducts, function types are exponents etc. The reason is that, in this case, **Set** is indeed a meaningful candidate for $\mathcal{C}$. In mathematics, however, we may afford to be more general. In this paper, we only assume that $\mathcal{C}$ is distributive, i.e., a category with finite products, finite coproducts, and the products distributive over the coproducts. A number of further properties of **Set**, such as, for instance, the presence of exponents, well-pointedness, and local smallness, are not needed for the purposes here and therefore not assumed.

Our notation for the structure present in $\mathcal{C}$ is standard. The product of objects A and B is written $A \times B$, the projections being $\mathsf{fst}_{A,B} : A \times B \to A$ and $\mathsf{snd}_{A,B} : A \times B \to B$. The pairing of $g : C \to A$ and $h : C \to B$, i.e., the unique morphism $f : C \to A \times B$ such that $\mathsf{fst}_{A,B} \circ f = g$ and $\mathsf{snd}_{A,B} \circ f = h$, is written $\langle g, h \rangle$. The coproduct of A and B is written $A + B$; the injections are $\mathsf{inl}_{A,B} : A \to A + B$ and $\mathsf{inr}_{A,B} : B \to A + B$. The case analysis of $g : A \to C$ and $h : B \to C$, i.e., the unique morphism

$f : A + B \to C$ such that $f \circ \mathsf{inl}_{A,B} = g$ and $f \circ \mathsf{inr}_{A,B} = h$, is denoted by $[\,g, h\,]$. 1 is the final object and 0 is the initial object. The inverse of the morphism $[\,\mathsf{id}_A \times \mathsf{inl}_{B,C}, \mathsf{id}_A \times \mathsf{inr}_{B,C}\,] : (A \times B) + (A \times C) \to A \times (B + C)$ is written $\mathsf{distr}_{A,B,C}$. For $g : A \to B$, $h : C \to D$, $g \times h$ and $g + h$ abbreviate $\langle\, g \circ \mathsf{fst}_{A,C}, h \circ \mathsf{snd}_{A,C}\,\rangle : A \times C \to B \times D$ and $[\,\mathsf{inl}_{B,D} \circ g, \mathsf{inr}_{B,D} \circ h\,] : A + C \to B + D$, respectively.

The identity functor on $\mathcal{C}$ is denoted by Id. The composition of two functors F and G ("G after F") is denoted by GF using juxtaposition just as in the application of a functor to an object or a morphism. GFA may be parsed both as $(GF)A$ and $G(FA)$, the meaning is the same.

3. Inductive types and iteration

Inductive types are constructor-generated types and come together with a function definition scheme known as iteration[1], the simplest and most fundamental recursion scheme. Category-theoretically, inductive types are modelled by initial (functor-)algebras. The categorical concepts of (functor-) algebra, algebra homomorphism and initial algebra are generalizations of the corresponding concepts in universal algebra [Wechler 1992].

DEFINITION 1. *Given an endofunctor F on $\mathcal{C}$, an* F-algebra (algebra with signature F) *is a pair $\boldsymbol{A} = (A, \varphi)$ consisting of an object A of $\mathcal{C}$* (carrier) *and a morphism $\varphi : FA \to A$ of $\mathcal{C}$* (algebra structure). *An* F-algebra morphism (homomorphism between algebras with signature F) *from an F-algebra $\boldsymbol{A} = (A, \varphi)$ to an F-algebra $\boldsymbol{B} = (B, \psi)$ is a morphism $f : A \to B$ of $\mathcal{C}$ such that*

$$f \circ \varphi = \psi \circ Ff$$

$$\begin{array}{ccc} FA & \xrightarrow{\varphi} & A \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FB & \xrightarrow{\psi} & B. \end{array}$$

The collection of all F-algebras and F-algebra morphisms, with identities and composition as in $\mathcal{C}$, forms a category written $\mathbf{Alg}_F$. This category may or may not have initial objects. Any initial object of $\mathbf{Alg}_F$ is called an *initial F-algebra*. To initial algebras, we apply the usual jargon for entities determined uniquely up to isomorphism: we speak of them as determined truly uniquely. If initial F-algebras exist, then one of them is designated as representative (the choice may be arbitrary), said to be *the* initial F-algebra and written $\boldsymbol{\mu F} = (\mu F, \mathsf{in}_F)$.

[1] In this paper, the word 'iteration' denotes structural recursion and not tail-recursion.

Corollary 1. *Given an endofunctor F with an initial algebra. Then, for any F-algebra $\boldsymbol{A} = (A, \varphi)$, there exists a unique morphism $f : \mu F \to A$ such that*

$$f \circ \mathsf{in}_F = \varphi \circ Ff$$

$$\begin{tikzcd} F\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "Ff"'] & \mu F \arrow[d, "f"] \\ FA \arrow[r, "\varphi"] & A. \end{tikzcd}$$

In the constructive algorithmics community, this f is commonly called the *F-catamorphism* or *F-fold* of φ and written $(\!|\, \varphi \,|\!)_F$.

The relation of initial algebras to inductive types is the following. The endofunctor represents a signature. The initial algebra models the inductive type: both the type as a pure container and the constructors that structure it. Catamorphisms, witnesses of initiality, correspond to iteratively defined functions: the result type and the step functions are modelled by an algebra, the catamorphism models the function defined.

The two most popular examples of inductive types are the natural number type and the type of lists with a given element type. Iteration for the natural number type is the iteration of the theory of computability.

Example 1. Define an endofunctor N by setting, for any A, $\mathsf{N}A = 1 + A$ and, for any $f : A \to B$, $\mathsf{N}f = \mathsf{id}_1 + f$. The inductive type of natural numbers is naturally modelled by the initial N-algebra $(\mu\mathsf{N}, \mathsf{in}_\mathsf{N})$. The type of naturals is $\mathsf{Nat} = \mu\mathsf{N}$ and its two constructors constant zero and the successor function are $\mathsf{zero} = \mathsf{in}_\mathsf{N} \circ \mathsf{inl}_{1,\mathsf{Nat}} : 1 \to \mathsf{Nat}$, and $\mathsf{succ} = \mathsf{in}_\mathsf{N} \circ \mathsf{inr}_{1,\mathsf{Nat}} : \mathsf{Nat} \to \mathsf{Nat}$. The function defined by iteration from a result type A and step functions $g : 1 \to A$ and $h : A \to A$ is modelled by the catamorphism $\mathsf{natiter}(g, h) = (\!|\, [\, g, h \,] \,|\!)_\mathsf{N}$, which, as it ought to be, is a unique function $f : \mathsf{Nat} \to A$ such that

$$f \circ \mathsf{zero} = g \quad \wedge \quad f \circ \mathsf{succ} = h \circ f$$

$$\begin{tikzcd} 1 \arrow[r, "\mathsf{zero}"] \arrow[d, equal] & \mathsf{Nat} \arrow[d, "f"] & \mathsf{Nat} \arrow[l, "\mathsf{succ}"'] \arrow[d, "f"] \\ 1 \arrow[r, "g"] & A & A. \arrow[l, "h"'] \end{tikzcd}$$

Example 2. For an object E, define an endofunctor L_E by setting, for any A, $\mathsf{L}_E A = 1 + E \times A$ and, for any $f : A \to B$, $\mathsf{L}_E f = \mathsf{id}_1 + \mathsf{id}_E \times f$. The type of lists with element type E is obtained from the initial L_E-algebra $(\mu\mathsf{L}_E, \mathsf{in}_{\mathsf{L}_E})$. The list type by itself is $\mathsf{List}E = \mu\mathsf{L}_E$; its two structure-providing constructors are $\mathsf{nil}_E = \mathsf{in}_{\mathsf{L}_E} \circ \mathsf{inl}_{1,E\times\mathsf{List}E} : 1 \to \mathsf{List}E$ and $\mathsf{cons}_E = \mathsf{in}_{\mathsf{L}_E} \circ \mathsf{inr}_{1,E\times\mathsf{List}E} : E \times \mathsf{List}E \to \mathsf{List}E$. The list iteration (or, in Haskell terminology, the foldr function) given by a type A and functions $g : 1 \to A$, $h : E \times A \to A$ is the catamorphism $\mathsf{foldr}_E(g, h) = (\!|\, [\, g, h\,]\, |\!)_{\mathsf{L}_E}$, a unique function $f : \mathsf{List}E \to A$ such that

$$f \circ \mathsf{nil}_E = g \quad \wedge \quad f \circ \mathsf{cons}_E = h \circ (\mathsf{id}_E \times f)$$

$$\begin{tikzcd} 1 \arrow[r, "\mathsf{nil}_E"] \arrow[d, equal] & \mathsf{List}E \arrow[d, "f"] & E \times \mathsf{List}E \arrow[l, "\mathsf{cons}_E"'] \arrow[d, "\mathsf{id}_E \times f"] \\ 1 \arrow[r, "g"] & A & E \times A. \arrow[l, "h"'] \end{tikzcd}$$

Catamorphisms enjoy a number of calculational properties with relevance for programming. Given an endofunctor F with an initial algebra, the "is-a" part of the defining characterization of F-catamorphisms immediately entails the cata *cancellation* law stating how catamorphisms compute: for any F-algebra $\boldsymbol{A} = (A, \varphi)$,

$$(\!|\, \varphi\, |\!)_F \circ \mathsf{in}_F = \varphi \circ F(\!|\, \varphi\, |\!)_F. \tag{1}$$

The "unique" part decomposes into the cata *reflection* and *fusion* laws useful in program transformation:

$$\mathsf{id}_{\mu F} = (\!|\, \mathsf{in}_F\, |\!)_F \tag{2}$$

and, for any two F-algebras $\boldsymbol{A} = (A, \varphi)$, $\boldsymbol{B} = (B, \psi)$ and morphism $f : A \to B$,

$$f \circ \varphi = \psi \circ Ff \quad \Rightarrow \quad f \circ (\!|\, \varphi\, |\!)_F = (\!|\, \psi\, |\!)_F. \tag{3}$$

(For some authors, cancellation is evaluation, and reflection and fusion are identity and promotion, respectively.)

Given a natural transformation $\tau : F \dot{\to} F'$ between two endofunctors with initial algebras, set $\mu\tau = (\!|\, \mathsf{in}_{F'} \circ \tau_{\mu F'}\, |\!)_F : \mu F \to \mu F'$. The cata *absorption law*, a further law helpful in program transformation, describes a relationship between F- and F'-catamorphisms: for any F'-algebra $\boldsymbol{A} = (A, \varphi)$,

$$(\!|\, \varphi\, |\!)_{F'} \circ \mu\tau = (\!|\, \varphi \circ \tau_A\, |\!)_F. \tag{4}$$

4. Comonads

In this paper, our project is to show that making use of the concept of comonad, it is possible to view different recursion schemes as instances of one scheme. Monads and comonads are central notions of category theory. Monads have also become important as a presentation structuring tool in programming language semantics [Moggi 1991] and a program structuring device [Wadler 1992]. Similar uses of comonads [Kieburtz 1999] are a newer development. We shall only discuss comonads to the extent we need here; comprehensive treatments of monads and comonads can be found in [Manes 1976, Barr and Wells 1984].

Definition 2. *A* comonad *on $\mathcal{C}$ is a triple $\boldsymbol{N} = (N, \varepsilon, \delta)$ consisting of an endofunctor N on $\mathcal{C}$* (underlying functor) *and two natural transformations $\varepsilon : N \dot{\to} \mathsf{Id}$* (counit) *and $\delta : N \dot{\to} N^2$* (comultiplication) *such that, for any A,*

$$\varepsilon_{NA} \circ \delta_A = \mathsf{id}_{NA} = N\varepsilon_A \circ \delta_A \tag{5}$$

$$\delta_{NA} \circ \delta_A = N\delta_A \circ \delta_A \tag{6}$$

$$\begin{tikzcd} & NA \arrow[dl, equal] \arrow[d, "\delta_A"] \arrow[dr, equal] & \\ NA & N^2A \arrow[l, "\varepsilon_{NA}"'] \arrow[r, "N\varepsilon_A"] & NA \end{tikzcd} \qquad \begin{tikzcd} NA \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] & N^2A \arrow[d, "\delta_{NA}"] \\ N^2A \arrow[r, "N\delta_A"] & N^3A. \end{tikzcd}$$

An alternative means of conceptualizing the structure present in a comonad is given by the concept of coKleisli triple.

Definition 3. *A* coKleisli triple *on $\mathcal{C}$ is a triple $\boldsymbol{N} = (N, \varepsilon, -^\dagger)$ consisting of an endofunction N on $|\mathcal{C}|$ (underlying object mapping), a $|\mathcal{C}|$-indexed family ε of morphisms $\varepsilon_A : NA \to A$ and an operation $-^\dagger$* (coextension) *taking morphisms $f : NA \to B$ to morphisms $f^\dagger : NA \to NB$ such that, for any $f : NA \to B$,*

$$\varepsilon_B \circ f^\dagger = f \tag{7}$$

for any A,

$$\mathsf{id}_{NA} = {\varepsilon_A}^\dagger \tag{8}$$

and, for any $f : NA \to B$, $g : NB \to C$,

$$g^\dagger \circ f^\dagger = (g \circ f^\dagger)^\dagger. \tag{9}$$

Given a coKleisli triple $\boldsymbol{N} = (N, \varepsilon, -^\dagger)$, we can form a category called the *coKleisli category* of $\boldsymbol{N}$ and written $\mathbf{CoKl}^{\boldsymbol{N}}$. The objects of $\mathbf{CoKl}^{\boldsymbol{N}}$ are those of $\mathcal{C}$; a morphism from A to B is a morphism $f : NA \to B$ of $\mathcal{C}$; the identity on A is $\varepsilon_A : NA \to A$; and the composite of $f : NA \to B$ and $g : NB \to C$ is $g \circ f^\dagger : NA \to C$.

The concepts of comonad and coKleisli triple are equivalent.

Proposition 1. *Giving a comonad is the same as giving a coKleisli triple: between the comonads and the coKleisli triples on $\mathcal{C}$, there exists a bijection.*

Proof. Given a comonad (N, ε, δ), the corresponding coKleisli triple is $(N, \varepsilon, -^\dagger)$ where, for any $f : NA \to B$,

$$f^\dagger = Nf \circ \delta_A. \tag{10}$$

Given a coKleisli triple $(N, \varepsilon, -^\dagger)$, the corresponding comonad is (N, ε, δ) where, for any $f : A \to B$, $Nf = (f \circ \varepsilon_A)^\dagger$, and, for any A, $\delta_A = \mathsf{id}_{NA}{}^\dagger$. $\square$

Viewing a comonad as a coKleisli triple tends to make its structure more visible. Having an explicit notation $f^\dagger$ for morphisms $Nf \circ \delta_A$ makes it easy, for instance, to realize that (6) is just an instance of a more general law that, for any $f : NA \to B$,

$$\delta_B \circ f^\dagger = (f^\dagger)^\dagger = Nf^\dagger \circ \delta_A. \tag{11}$$

The simplest examples of comonads result from the identity functor and the product functor.

Example 3. The identity functor Id yields the comonad $\mathbf{Id} = (\mathsf{Id}, \varepsilon, \delta)$ where, for any A, $\varepsilon_A = \delta_A = \mathsf{id}_A$ and, for any $f : A \to B$, $f^\dagger = f$.

Example 4. For any object E, the product functor $\mathsf{Prod}^E = - \times E$ gives rise to the comonad $\mathbf{Prod}^E = (\mathsf{Prod}^E, \varepsilon, \delta)$ where, for any A, $\varepsilon_A = \mathsf{fst}_{A,E}$, $\delta_A = \langle\, \mathsf{id}_{A\times E}, \mathsf{snd}_{A,E}\,\rangle$ and, for any $f : \mathsf{Prod}^E A \to B$, $f^\dagger = \langle\, f, \mathsf{snd}_{A,E}\,\rangle$.

A more complex example is obtained from what we call the H-branching stream functor.

Example 5. Given an endofunctor H, the *H-branching stream object* given by an object A is the triple $(\mathsf{Str}^H A, \mathsf{hd}^H_A, \mathsf{tl}^H_A)$ consisting of an object $\mathsf{Str}^H A$ and morphisms $\mathsf{hd}^H_A : \mathsf{Str}^H A \to A$ and $\mathsf{tl}^H_A : \mathsf{Str}^H A \to H\mathsf{Str}^H A$ such that, for any object C and morphisms $g : C \to A$, $h : C \to HC$, there exists a unique function $f : C \to \mathsf{Str}^H A$, written $\mathsf{gen}^H(g, h)$, satisfying

$$\mathsf{hd}^H_A \circ f = g \quad \wedge \quad \mathsf{tl}^H_A \circ f = Hf \circ h$$

$$\begin{array}{ccccc} A & \xleftarrow{g} & C & \xrightarrow{h} & HC \\ \| & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle Hf} \\ A & \xleftarrow{\mathsf{hd}^H_A} & \mathsf{Str}^H A & \xrightarrow{\mathsf{tl}^H_A} & H\mathsf{Str}^H A. \end{array}$$

The object mapping Str^H is made a functor by setting, for any $f : A \to B$, $\mathsf{Str}^H f = \mathsf{gen}^H(f \circ \mathsf{hd}^H_A, \mathsf{tl}^H_A)$. This functor gives rise to the comonad $\mathbf{Str}^H = (\mathsf{Str}^H, \varepsilon, \delta)$ where, for any A, $\varepsilon_A = \mathsf{hd}^H_A$, $\delta_A = \mathsf{gen}^H(\mathsf{id}_{\mathsf{Str}^H A}, \mathsf{tl}^H_A)$ and, for any $f : \mathsf{Str}^H A \to B$, $f^\dagger = \mathsf{gen}^H(f, \mathsf{tl}^H_A)$.

5. Distributive comonads

In order for a comonad to yield a recursion scheme for an inductive type, the comonad and the base functor of the inductive type must interact in a useful way. Interactions of this sort are typically provided by distributive laws [Beck 1969; Barr and Wells 1984, Ch. 9; Lenisa *et al.* 2000]. For our purpose, it is useful to have the functor distribute over the comonad, so we shall now proceed to the analysis of comonads equipped with a distributive law of this kind. For brevity, we speak of distributive comonads.

DEFINITION 4. *Given an endofunctor F on $\mathcal{C}$. An F-*distributive comonad *on $\mathcal{C}$ is a pair $\mathbb{N} = (\boldsymbol{N}, \kappa)$ consisting of a comonad $\boldsymbol{N} = (N, \varepsilon, \delta)$ on $\mathcal{C}$ and a* distributive law *κ of F* over *$\boldsymbol{N}$, i.e., a natural transformation $\kappa : FN \dot{\to} NF$ such that, for any A,*

$$\varepsilon_{FA} \circ \kappa_A = F\varepsilon_A \tag{12}$$

$$\delta_{FA} \circ \kappa_A = N\kappa_A \circ \kappa_{NA} \circ F\delta_A \tag{13}$$

$$\begin{tikzcd} FNA \arrow[r, "\kappa_A"] \arrow[d, "F\varepsilon_A"'] & NFA \arrow[d, "\varepsilon_{FA}"] \\ FA \arrow[r, equal] & FA \end{tikzcd} \qquad \begin{tikzcd} FNA \arrow[rr, "\kappa_A"] \arrow[d, "F\delta_A"'] & & NFA \arrow[d, "\delta_{FA}"] \\ FNNA \arrow[r, "\kappa_{NA}"] & NFNA \arrow[r, "N\kappa_A"] & NNFA. \end{tikzcd}$$

Given a comonad $\boldsymbol{N} = (N, \varepsilon, \delta)$ on $\mathcal{C}$, a distributive law κ of F over $\boldsymbol{N}$ gives a lifting of $\boldsymbol{N}$ to a comonad on $\mathbf{Alg}_F$. The lifted comonad $\bar{\boldsymbol{N}} = (\bar{N}, \bar{\varepsilon}, \bar{\delta})$ is obtained by defining, for any F-algebra $\boldsymbol{A} = (A, \varphi)$, $\bar{N}\boldsymbol{A} = (NA, N\varphi \circ \kappa_A)$, $\bar{\varepsilon}_{\boldsymbol{A}} = \varepsilon_A$ and $\bar{\delta}_{\boldsymbol{A}} = \delta_A$, and, for any $f : A \to B$, $\bar{N}f = Nf$.

Another way to describe the same structure goes through what we here call distributive coKleisli triples.

DEFINITION 5. *An F-*distributive coKleisli triple *on $\mathcal{C}$ is pair $\mathbb{N} = (\boldsymbol{N}, -^{\ddagger})$ consisting of a coKleisli triple $\boldsymbol{N} = (N, \varepsilon, -^{\dagger})$ on $\mathcal{C}$ and an operation $-^{\ddagger}$ (*"coextension under F"*) that takes morphisms $\varphi : FNA \to B$ to morphisms $\varphi^{\ddagger} : FNA \to NB$ such that, for any $\varphi : FNA \to B$,*

$$\varepsilon_B \circ \varphi^{\ddagger} = \varphi \tag{14}$$

for any $\varphi : FNA \to B$, $f : NB \to C$,

$$f^{\dagger} \circ \varphi^{\ddagger} = (f \circ \varphi^{\ddagger})^{\ddagger} \tag{15}$$

and, for any $f : NA \to B$, $\psi : FNB \to C$,

$$\psi^{\ddagger} \circ Ff^{\dagger} = (\psi \circ Ff^{\dagger})^{\ddagger}. \tag{16}$$

From an F-distributive coKleisli triple $\mathbb{N} = (\boldsymbol{N}, -^{\ddagger})$, there is a short distance to a category defined thus: an object is an FN-algebra; a morphism between two FN-algebras $\boldsymbol{A} = (A, \varphi)$ and $\boldsymbol{B} = (B, \psi)$, at the same time, is not an FN-algebra morphism, but a morphism $f : NA \to B$ of $\mathcal{C}$ such that $f \circ \varphi^{\ddagger} = \psi \circ Ff^{\dagger}$; the identity on $\boldsymbol{A} = (A, \varphi)$ is ε_A; and the composite of $f : NA \to B$ and $g : NB \to C$ is $g \circ f^{\dagger}$.

PROPOSITION 2. *Giving an F-distributive comonad is equivalent to giving an F-distributive coKleisli triple.*

PROOF. We build on Proposition 1. An F-distributive comonad $(N, \varepsilon, \delta, \kappa)$ gives rise to an F-distributive coKleisli triple $(N, \varepsilon, -^\dagger, -^\ddagger)$ where, for any $\varphi : FNA \to B$,

$$\varphi^\ddagger = N\varphi \circ \kappa_{NA} \circ F\delta_A . \tag{17}$$

An F-distributive coKleisli triple $(N, \varepsilon, -^\dagger, -^\ddagger)$ determines an F-distributive comonad $(N, \varepsilon, \delta, \kappa)$ where, for any A, $\kappa_A = (F\varepsilon_A)^\ddagger$. □

We also record that, for any $f : FA \to B$,

$$Nf \circ \kappa_A = (f \circ F\varepsilon_A)^\ddagger \tag{18}$$

and, for any $\varphi : FNA \to B$,

$$\delta_B \circ \varphi^\ddagger = (\varphi^\ddagger)^\ddagger = N\varphi^\ddagger \circ \kappa_{NA} \circ F\delta_A . \tag{19}$$

The comonads of Examples 3–5 possess extensions to F-distributive comonads irrespective of what F happens to be.

EXAMPLE 6. The obvious sole possible extension of the identity comonad $\mathbf{Id}$ to an F-distributive comonad is $\mathbf{Id} = (\mathbf{Id}, \kappa)$ where, for any A, $\kappa_A = F\mathsf{id}_A$ and, for any $\varphi : FA \to B$, $\varphi^\ddagger = \varphi$.

EXAMPLE 7. Given an F-algebra $\boldsymbol{E} = (E, \chi)$, the product comonad $\mathbf{Prod}^E$ gives us an F-distributive comonad $\mathbf{Prod}^{\boldsymbol{E}} = (\mathbf{Prod}^E, \kappa)$ where, for any A, $\kappa_A = \langle F\mathsf{fst}_{A,E}, \chi \circ F\mathsf{snd}_{A,E} \rangle$ and, for any $\varphi : F\mathsf{Prod}^E A \to B$, $\varphi^\ddagger = \langle \varphi, \chi \circ F\mathsf{snd}_{A,E} \rangle$.

EXAMPLE 8. Say that an *F-distributive endofunctor* is a pair $\boldsymbol{H} = (H, \theta)$ where an endofunctor H is accompanied by a natural transformation $\theta : FH \dot{\to} HF$. Then, given an F-distributive endofunctor $\boldsymbol{H} = (H, \theta)$, the H-branching stream comonad $\mathbf{Str}^H$ extends to an F-distributive comonad $\mathbf{Str}^{\boldsymbol{H}} = (\mathbf{Str}^H, \kappa)$ where, for any A, $\kappa_A = \mathsf{gen}^H(F\mathsf{hd}^H_A, \theta_{\mathsf{Str}^H A} \circ F\mathsf{tl}^H_A)$ and, for any $\varphi : F\mathsf{Str}^H A \to B$, $\varphi^\ddagger = \mathsf{gen}^H(\varphi, \theta_{\mathsf{Str}^H A} \circ F\mathsf{tl}^H_A)$.

6. Recursion schemes from comonads

We are now ready to define a generic recursion scheme that organizes its structure of recursive calls in terms of a comonad and so instantiates to a variety of different particular recursion schemes. The scheme presents genericity in two levels. One is traditional: the scheme is parameterized by a functor F specifying an inductive type. The other level is a consequence of the presence of a F-distributive comonad $\mathbb{N} = (\boldsymbol{N}, \kappa)$. The comonad represents an abstraction of the different manners in which a recursion scheme makes use of the argument in each recursive step. For instance, some recursion schemes make a copy of the argument so as to make it available to the step function (primitive recursion). Others repeatedly deconstruct the argument so as to perform recursive calls on various subparts of it in a single step (course-of-value iteration).

To start with, let us record that, given an endofunctor F with an initial algebra, an F-distributive comonad $\mathbb{N} = (\boldsymbol{N}, \kappa)$ satisfies certain laws involving F-catamorphisms. From (5) and (6), noticing that κ is natural, we may by cata fusion (3) deduce that, for any F-algebra $\boldsymbol{A} = (A, \varphi)$,

$$\varepsilon_A \circ \llparenthesis\, N\varphi \circ \kappa_A \,\rrparenthesis_F = \llparenthesis\, \varphi \,\rrparenthesis_F \tag{20}$$

$$\delta_A \circ \llparenthesis\, N\varphi \circ \kappa_A \,\rrparenthesis_F = \llparenthesis\, N(N\varphi \circ \kappa_A) \circ \kappa_{NA} \,\rrparenthesis_F . \tag{21}$$

From (19), cata fusion (3) gives us the law that, for any FN-algebra $\boldsymbol{A} = (A, \varphi)$,

$$\delta_A \circ \llparenthesis\, \varphi^{\ddagger} \,\rrparenthesis_F = \llparenthesis\, N\varphi^{\ddagger} \circ \kappa_{NA} \,\rrparenthesis_F . \tag{22}$$

In the view of (18), this law is a generalization of (21).

Define now a special morphism $\mathsf{i}_F^{\mathbb{N}} : \mu F \to N\mu F$ by setting

$$\mathsf{i}_F^{\mathbb{N}} = \llparenthesis\, N\mathsf{in}_F \circ \kappa_{\mu F} \,\rrparenthesis_F = \llparenthesis\, (\mathsf{in}_F \circ F\varepsilon_{\mu F})^{\ddagger} \,\rrparenthesis_F . \tag{23}$$

This morphism obeys specific laws. By cata fusion (3) from cata cancellation (1) and the naturality of κ, we get that, for any F-algebra $\boldsymbol{B} = (B, \psi)$,

$$N\llparenthesis\, \psi \,\rrparenthesis_F \circ \mathsf{i}_F^{\mathbb{N}} = \llparenthesis\, N\psi \circ \kappa_B \,\rrparenthesis_F . \tag{24}$$

Combining (20) with cata reflection (2) gives

$$\varepsilon_{\mu F} \circ \mathsf{i}_F^{\mathbb{N}} = \mathsf{id}_{\mu F} . \tag{25}$$

Putting (21) and (24) together gives

$$\delta_{\mu F} \circ \mathsf{i}_F^{\mathbb{N}} = N\mathsf{i}_F^{\mathbb{N}} \circ \mathsf{i}_F^{\mathbb{N}} . \tag{26}$$

while (22) in conjunction with (24) yields that, for any FN-algebra $\boldsymbol{A} = (A, \varphi)$,

$$\delta_{\mu F} \circ \llparenthesis\, \varphi^{\ddagger} \,\rrparenthesis_F = N\llparenthesis\, \varphi^{\ddagger} \,\rrparenthesis_F \circ \mathsf{i}_F^{\mathbb{N}} . \tag{27}$$

These observations made, we are ready to prove our main result. The following theorem states that any F-distributive comonad yields a recursion scheme for the inductive type induced by F.

THEOREM 1. *Given an endofunctor F with an initial algebra and an F-distributive comonad $\mathbb{N} = (\boldsymbol{N}, \kappa)$. Then, for any FN-algebra $\boldsymbol{A} = (A, \varphi)$, we have a unique morphism $f : \mu F \to A$ such that*

$$f \circ \mathsf{in}_F = \varphi \circ F(Nf \circ \mathsf{i}_F^{\mathbb{N}})$$

$$\begin{tikzcd}
F\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "F\mathsf{i}_F^{\mathbb{N}}"'] & \mu F \arrow[dd, "f"] \\
FN\mu F \arrow[d, "FNf"'] & \\
FNA \arrow[r, "\varphi"] & A.
\end{tikzcd}$$

We shall speak of this f as the *F-$\mathbb{N}$-g-catamorphism* (generalized catamorphism) of φ and denote it by $\llparenthesis\, \varphi \,\rrparenthesis_F^{\mathbb{N}}$.

Proof. We show that the one and only f with the requested property is $\varepsilon_A \circ \llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F$. This means checking that

$$f \circ \mathsf{in}_F = \varphi \circ F(Nf \circ \mathsf{i}_F^{\mathbb{N}}) \quad \equiv \quad f = \varepsilon_A \circ \llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F .$$

The right-to-left direction is proved by the following calculation:

$$\left[\begin{array}{ll}
\rhd & f = \varepsilon_A \circ \llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F \\
\hline
 & f \circ \mathsf{in}_F \\
= & \quad - \lhd - \\
 & \varepsilon_A \circ \llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F \circ \mathsf{in}_F \\
= & \quad - \text{cata cancellation} - \\
 & \varepsilon_A \circ \varphi^\ddagger \circ F\llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F \\
= & \quad - (14) - \\
 & \varphi \circ F\llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F \\
= & \quad - (5) - \\
 & \varphi \circ F(N\varepsilon_A \circ \delta_A \circ \llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F) \\
= & \quad - (27) - \\
 & \varphi \circ F(N\varepsilon_A \circ N\llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F \circ \mathsf{i}_F^{\mathbb{N}}) \\
= & \quad - \lhd - \\
 & \varphi \circ F(Nf \circ \mathsf{i}_F^{\mathbb{N}}).
\end{array}\right.$$

For the left-to-right direction, we argue as follows:

$$\left[\begin{array}{ll}
\rhd & f \circ \mathsf{in}_F = \varphi \circ F(Nf \circ \mathsf{i}_F^{\mathbb{N}}) \\
\hline
 & f \\
= & \quad - (25) - \\
 & f \circ \varepsilon_{\mu F} \circ \mathsf{i}_F^{\mathbb{N}} \\
= & \quad - \text{naturality of } \varepsilon - \\
 & \varepsilon_A \circ Nf \circ \mathsf{i}_F^{\mathbb{N}} \\
= & \quad - \text{cata characterization} - \\
 & \left[\begin{array}{ll}
 & Nf \circ \mathsf{i}_F^{\mathbb{N}} \circ \mathsf{in}_F \\
= & \quad - (23), \text{cata cancellation} - \\
 & N(f \circ \mathsf{in}_F) \circ \kappa_{\mu F} \circ F\mathsf{i}_F^{\mathbb{N}} \\
= & \quad - \lhd - \\
 & N(\varphi \circ F(Nf \circ \mathsf{i}_F^{\mathbb{N}})) \circ \kappa_{\mu F} \circ F\mathsf{i}_F^{\mathbb{N}} \\
= & \quad - \text{naturality of } \kappa - \\
 & N\varphi \circ \kappa_{NA} \circ F(N(Nf \circ \mathsf{i}_F^{\mathbb{N}}) \circ \mathsf{i}_F^{\mathbb{N}}) \\
= & \quad - (26) - \\
 & N\varphi \circ \kappa_{NA} \circ F(NNf \circ \delta_{\mu F} \circ \mathsf{i}_F^{\mathbb{N}}) \\
= & \quad - \text{naturality of } \delta - \\
 & N\varphi \circ \kappa_{NA} \circ F(\delta_C \circ Nf \circ \mathsf{i}_F^{\mathbb{N}}) \\
= & \quad - (17) - \\
 & \varphi^\ddagger \circ F(Nf \circ \mathsf{i}_F^{\mathbb{N}})
\end{array}\right. \\
 & \varepsilon_A \circ \llparenthesis\, \varphi^\ddagger \,\rrparenthesis_F .
\end{array}\right.$$

$\square$

By specializing the new generic recursion scheme for particular comonads, we can find out what it really covers. The distributive comonads from Examples 6–8 turn out to yield iteration, a generalization of primitive recursion and a generalization of course-of-value iteration.

EXAMPLE 9. The simplest comonad, the identity comonad $\mathbf{Id}$, recovers iteration. Indeed, since $\mathsf{i}_F^{\mathbf{Id}} = \mathsf{id}_{\mu F}$, the F-$\mathbf{Id}$-g-catamorphism $(\!|\, \varphi \,|\!)_F^{\mathbf{Id}}$ given by an F-algebra $\boldsymbol{A} = (A, \varphi)$ is bound to be the unique morphism $f : \mu F \to A$ such that $f \circ \mathsf{in}_F = \varphi \circ F f$. But this morphism, as we know, is $(\!|\, \varphi \,|\!)_F$, the F-catamorphism of φ.

EXAMPLE 10. Consider next the F-distributive comonad $\mathbf{Prod}^{\boldsymbol{E}}$ for an F-algebra $\boldsymbol{E} = (E, \chi)$. From (23), we get that

$$\mathsf{i}_F^{\mathbf{Prod}^{\boldsymbol{E}}} = \langle\, \mathsf{id}_{\mu F}, (\!|\, \chi \,|\!)_F \,\rangle. \tag{28}$$

As an implication, the F-$\mathbf{Prod}^{\boldsymbol{E}}$-g-catamorphism $(\!|\, \varphi \,|\!)_F^{\mathbf{Prod}^{\boldsymbol{E}}}$ given by an $F\mathsf{Prod}^E$-algebra $\boldsymbol{A} = (A, \varphi)$ is the unique morphism $f : \mu F \to A$ such that

$$f \circ \mathsf{in}_F = \varphi \circ F\langle\, f, (\!|\, \chi \,|\!)_F \,\rangle$$

or, in other words, the first component of the unique pair (f, g) consisting of morphisms $f : \mu F \to A$ and $g : \mu F \to E$ such that

$$f \circ \mathsf{in}_F = \varphi \circ F\langle\, f, g \,\rangle \quad \wedge \quad g \circ \mathsf{in}_F = \chi \circ F g$$

$$\begin{tikzcd} F\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "F\langle f,g \rangle"'] & \mu F \arrow[d, "f"] \\ F(A \times E) \arrow[r, "\varphi"] & A \end{tikzcd} \qquad \begin{tikzcd} F\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "Fg"'] & \mu F \arrow[d, "g"] \\ FE \arrow[r, "\chi"] & E. \end{tikzcd}$$

The morphism so specified is what Malcolm [1990] calls the F-χ-*zygomorphism* of φ; we denote it by $\langle\!|\, \varphi \,|\!\rangle_F^{\chi}$. The recursion scheme that produces zygomorphisms is best thought of as a "semi-mutual" iteration where two functions are defined simultaneously, one being the function of interest and the other an auxiliary function.

An important special case of zygomorphisms results from choosing $\boldsymbol{E} = \boldsymbol{\mu} F = (\mu F, \mathsf{in}_F)$. Because of cata reflection, (28) specializes to

$$\mathsf{i}_F^{\mathbf{Prod}^{\boldsymbol{\mu} F}} = \langle\, \mathsf{id}_{\mu F}, \mathsf{id}_{\mu F} \,\rangle. \tag{29}$$

The consequence is that the F-$\mathbf{Prod}^{\boldsymbol{\mu} F}$-g-catamorphism given by an $F\mathsf{Prod}^{\mu F}$-algebra $\boldsymbol{A} = (A, \varphi)$ is the unique morphism $f : \mu F \to A$ such that

$$f \circ \mathsf{in}_F = \varphi \circ F\langle\, f, \mathsf{id}_{\mu F} \,\rangle$$

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

meaning that f is what is usually called the F-*paramorphism* of φ and written $\langle\!|\, \varphi \,|\!\rangle_F$. Paramorphisms, as introduced by Meertens [1992], are functions defined by primitive recursion. Primitive recursion is equivalent to "semi-mutual" iteration where the auxiliary function is the identity function (described as a copying function).

EXAMPLE 11. Given an F-distributive endofunctor $\boldsymbol{H} = (H, \theta)$, let us now study the F-distributive comonad $\mathbf{Str}^{\boldsymbol{H}}$. From (23), we derive

$$\mathsf{i}_F^{\mathbf{Str}^{\boldsymbol{H}}} = \mathsf{gen}^H(\mathsf{id}_{\mu F}, (\!|\, H\mathsf{in}_F \circ \theta_{\mu F} \,|\!)_F). \tag{30}$$

Hence, given an $F\mathsf{Str}^H$-algebra $\boldsymbol{A} = (A, \varphi)$, the F-$\mathbf{Str}^{\boldsymbol{H}}$-g-catamorphism $(\!|\, \varphi \,|\!)_F^{\mathbf{Str}^{\boldsymbol{H}}}$ is the unique $f : \mu F \to A$ such that

$$f \circ \mathsf{in}_F = \varphi \circ F\mathsf{gen}^H(f, (\!|\, H\mathsf{in}_F \circ \theta_{\mu F} \,|\!)_F)$$

or, which is the same, the first component in the unique pair (f, g) consisting of morphisms $f : \mu F \to A$ and $g : \mu F \to H\mu F$ such that

$$f \circ \mathsf{in}_F = \varphi \circ F\mathsf{gen}^H(f, g) \quad \wedge \quad g \circ \mathsf{in}_F = H\mathsf{in}_F \circ \theta_{\mu F} \circ Fg$$

$$\begin{tikzcd} F\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "F\mathsf{gen}^H(f,g)"'] & \mu F \arrow[d, "f"] \\ F\mathsf{Str}^H A \arrow[r, "\varphi"] & A \end{tikzcd} \qquad \begin{tikzcd} F\mu F \arrow[rr, "\mathsf{in}_F"] \arrow[d, "Fg"'] & & \mu F \arrow[d, "g"] \\ FH\mu F \arrow[r, "\theta_{\mu F}"] & HF\mu F \arrow[r, "H\mathsf{in}_F"] & H\mu F. \end{tikzcd}$$

We call such f the F-θ-g-*histomorphism* (generalized histomorphism) of φ and denote it $\{\!|\, \varphi \,|\!\}_F^{\theta}$, for a reason that will be obvious in a moment.

Consider the special case $\boldsymbol{H} = \boldsymbol{F} = (F, \mathsf{id}_{FF})$. Keeping in mind that $(\!|\, F\mathsf{in}_F \,|\!)_F = \mathsf{in}_F^{-1}$, by specializing (30), we infer

$$\mathsf{i}_F^{\mathbf{Str}^{\boldsymbol{F}}} = \mathsf{gen}^F(\mathsf{id}_{\mu F}, \mathsf{in}_F^{-1}) \tag{31}$$

from where it follows that, given an $F\mathsf{Str}^F$-algebra $\boldsymbol{A} = (A, \varphi)$, the F-$\mathbf{Str}^{\boldsymbol{F}}$-g-catamorphism $(\!|\, \varphi \,|\!)_F^{\mathbf{Str}^{\boldsymbol{F}}}$ is the unique $f : \mu F \to A$ such that

$$f \circ \mathsf{in}_F = \varphi \circ F\mathsf{gen}^F(f, \mathsf{in}_F^{-1})$$

$$\begin{tikzcd} F\mu F \arrow[r, "\mathsf{in}_F"] \arrow[d, "F\mathsf{gen}^F(f,\mathsf{in}_F^{-1})"'] & \mu F \arrow[d, "f"] \\ F\mathsf{Str}^F A \arrow[r, "\varphi"] & A. \end{tikzcd}$$

In [Uustalu and Vene 1999a, Vene 2000], such f was called the F-*histomorphism* of φ and written $\{\!|\, \varphi \,|\!\}_F$. Histomorphisms model course-of-value iteration; g-histomorphisms model a variation where the course associated to a given argument value need not amount to the natural organization of all of its recursive components into a branching stream, but may have a different content.

Just as catamorphisms, g-catamorphisms satisfy a number of calculational properties and the important ones are very close to those of catamorphisms. The fact that these are simple has the value of strengthening the grounds for the belief that the construction is not ad hoc. More important, however, is the fact that from each of these properties, we may, at will, derive a specialized property of the g-catamorphisms for any particular distributive comonad of interest.

Given an endofunctor F with an initial algebra and an F-distributive comonad $\mathbb{N} = (\boldsymbol{N}, \kappa)$. From the defining characterization of F-$\mathbb{N}$-g-catamorphisms, it immediately follows that they obey a *cancellation law* stating that, for any FN-algebra $\boldsymbol{A} = (A, \varphi)$,

$$(\!|\, \varphi \,|\!)_F^{\mathbb{N}} \circ \mathsf{in}_F = \varphi \circ F(N(\!|\, \varphi \,|\!)_F^{\mathbb{N}} \circ \mathsf{i}_F^{\mathbb{N}}) \tag{32}$$

and *reflection* and *fusion laws* stating that

$$\mathsf{id}_{\mu F} = (\!|\, \mathsf{in}_F \circ F\varepsilon_{\mu F} \,|\!)_F^{\mathbb{N}} \tag{33}$$

and, for any two FN-algebras $\boldsymbol{A} = (A, \varphi)$, $\boldsymbol{B} = (B, \psi)$ and morphism $f : A \to B$

$$f \circ \varphi = \psi \circ FNf \quad \Rightarrow \quad h \circ (\!|\, \varphi \,|\!)_F^{\mathbb{N}} = (\!|\, \psi \,|\!)_F^{\mathbb{N}}. \tag{34}$$

From the proof of the theorem, we may record that, for any FN-algebra $\boldsymbol{A} = (A, \varphi)$,

$$(\!|\, \varphi \,|\!)_F^{\mathbb{N}} = \varepsilon_A \circ (\!|\, \varphi^\ddagger \,|\!)_F \tag{35}$$

and

$$(\!|\, \varphi^\ddagger \,|\!)_F = N(\!|\, \varphi \,|\!)_F^{\mathbb{N}} \circ \mathsf{i}_F^{\mathbb{N}}. \tag{36}$$

F-catamorphisms are easily expressed as F-$\mathbb{N}$-g-catamorphisms: for any F-algebra $\boldsymbol{A} = (A, \varphi)$,

$$(\!|\, \varphi \,|\!)_F = (\!|\, \varphi \circ F\varepsilon_A \,|\!)_F^{\mathbb{N}}. \tag{37}$$

Given a natural transformation $\tau : F \dot{\to} F'$ between two endofunctors with initial algebras, an F-distributive comonad $\mathbb{N} = (\boldsymbol{N}, \kappa)$ and an F'-distributive comonad $\mathbb{N}' = (\boldsymbol{N}, \kappa')$ such that, for any A, $N\tau_A \circ \kappa_A = \kappa'_A \circ \tau_{NA}$, the interrelation between F-$\mathbb{N}$- and F'-$\mathbb{N}'$-g-catamorphisms is described by an *absorption law* stating that, for any $F'N$-algebra $\boldsymbol{A} = (A, \varphi)$,

$$(\!|\, \varphi \,|\!)_{F'}^{\mathbb{N}'} \circ \mu\tau = (\!|\, \varphi \circ \tau_{NA} \,|\!)_F^{\mathbb{N}}. \tag{38}$$

7. Implementation in Haskell

To illustrate the theory studied, we now show how to implement it in the functional programming language Haskell. More precisely, we use Haskell 98 [Peyton Jones and Hughes 1999] extended with multi-parameter classes [Jones 1995] and rank-2 type signatures [McCracken 1984]. These two features, though not part of the language standard yet, are both supported in the newer versions of the main implementations Hugs and GHC.

In the presentation of code, we use the "literate comment" convention: lines in which '`>`' is the first character are code, all other lines are comment. Speaking of classes, we refer to classes in the sense of Haskell, i.e., type-constructor classes.

7.1 Functors

Functors arise from the association of a morphism mapping to an object mapping. A functor in Haskell is a type constructor `f` 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` by itself is the object mapping part of a functor. The morphism mapping is the associated function `fmap`. The class declaration forces `fmap` to have the correct typing, but cannot force it to preserve identities and composition, so each time the programmer provides a definition for the `fmap` function for a particular type constructor `f`, it is his responsibility to ensure that these conditions are met.

Any particular type constructor is extended into a functor by declaring it to be an instance of `Functor` and supplying a definition for `fmap`. The list type constructor `[]`, for example, is made a functor in the Haskell Prelude as follows:

```
instance Functor [] where
   fmap = map
```

By this instance declaration, `[]` is stated to be a functor with the well-known list function `map` as the morphism mapping.

7.2 Inductive types and iteration

Inductive types, that is, initial algebras, are fixed-points of functorial type constructors. In Haskell, therefore, we can define them by a recursive type renaming declaration:

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

Given a type constructor `f`, this declaration produces a new type `Mu f` with the same representation as the type `f (Mu f)`; this is the carrier of the initial `f`-algebra. In addition, we also get a data constructor `In :: f (Mu f) -> Mu f` for explicit coercion between the two types in one direction; this is the structure of the initial algebra. Its inverse, i.e., the coercion in the other direction, is definable by pattern matching:

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

Catamorphisms are defined recursively by appealing to their cancellation law (1):

```
> cata :: Functor f => (f c -> c) -> Mu f -> c
> cata phi = phi . fmap (cata phi) . unIn
```

The `cata` combinator takes a function `phi :: f c -> c`, i.e., an algebra structure, into a function `cata phi :: Mu f -> c`, the algebra morphism determined by `phi`. In the type signature of `cata`, we need to constrain `f` to be of class `Functor`, as the defining equation uses `fmap`.

Example 12. The inductive type of natural numbers is implemented as follows. First, using a data type declaration, we introduce a new type constructor `N` corresponding to the object mapping of the underlying functor of the natural number type. Then, with an instance declaration, we make `N` an instance of the class `Functor` and define `fmap` as the appropriate morphism mapping.

```
> data N c = Z | S c

> instance Functor N where
>    fmap g Z     = Z
>    fmap g (S x) = S (g x)
```

The natural number type and the constructors constant zero and the successor function are now defined via `Mu` and `In`. For the definition the naturals number type, it suffices to use a type synonym declaration.

```
> type Nat = Mu N

> zero :: Nat
> zero  = In Z

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

The addition of two naturals is defined as a catamorphism as follows:

```
> add :: Nat -> Nat -> Nat
> add m = cata phi
>     where phi Z     = m
>           phi (S f) = succ f
```

Here the first equation for `phi` defines `add m zero` [this is `phi Z`] while the second defines `add m (succ n)` [`phi (S f)`] in terms of `add m n` [`f`].

Similarly, the multiplication of two naturals is also definable as a catamorphism:

```
> mult :: Nat -> Nat -> Nat
> mult m = cata phi
>     where phi Z     = zero
>           phi (S f) = add m f
```

7.3 *Comonads*

In Haskell, monads are extensively used as a device for disciplined introduction of impure features. Monads are defined in the Haskell Prelude as a type-constructor class, essentially in the Kleisli format, and the language has special syntax supporting their use. Comonads, however, are not standard part of the language. We might define them by dualizing the definition of monads in Haskell. For our purposes, however, a definition based on counit and comultiplication fits better.

```
> class Functor n => Comonad n where
>    extr :: n a -> a
>    dupl :: n a -> n (n a)
```

Similarly to the case of functors, with a class declaration we can only state the type signatures of the operations and it remains the programmer's responsibility to ensure that every instance he creates satisfies the required laws.

Example 13. Haskell has predefined products. In an ideal Haskell, therefore, the product type constructor as we need it ought to be definable by means of this type synonym declaration:

```
  type Prod e c = (c, e)
```

In reality, however, this declaration only defines a type `Prod e c` and does not define a type constructor `Prod e`. This means that we are forced to make a data type declaration:

```
> data Prod e c = Pair c e
```

To avoid any need for direct manipulations with the constructor `Pair`, we also define the standard projection functions and a pair-forming combinator:

```
> outl :: Prod e c -> c
> outl (Pair x _) = x

> outr :: Prod e c -> e
> outr (Pair _ y) = y

> fork :: (a -> c) -> (a -> e) -> a -> Prod e c
> fork g1 g2 z = Pair (g1 z) (g2 z)
```

The type constructor `Prod e` is extended into a functor and further into a comonad by means of the following instance declarations.

```
> instance Functor (Prod e) where
>    fmap g = fork (g . outl) outr

> instance Comonad (Prod e) where
>    extr = outl
>    dupl = fork id outr
```

EXAMPLE 14. The branching streams type constructor given by a functor `h` is implemented using a recursive data type declaration:

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

The destructor functions are defined from the constructor `Consf` with the help of pattern matching.

```
> hdf :: Strf h c -> c
> hdf (Consf x _ ) = x

> tlf :: Strf h c -> h (Strf h c)
> tlf (Consf _ xs) = xs
```

The definition of the generator combinator is based on its cancellation law. Since this involves the `fmap` function associated to `h`, we need to make a constraint that the type constructor `h` is a functor.

```
> genStrf :: Functor h =>
>                       (a -> c) -> (a -> h a) -> a -> Strf h c
> genStrf g1 g2 z = Consf (g1 z) (fmap (genStrf g1 g2) (g2 z))
```

All this done, we can extend the type constructor `Strf h` into a functor, and further into a comonad, using the instance declaration mechanism. Since here we need `genStrf`, we again need the constraint that `h` is a functor.

```
> instance Functor h => Functor (Strf h) where
>    fmap g = genStrf (g . hdf) tlf

> instance Functor h => Comonad (Strf h) where
>    extr = hdf
>    dupl = genStrf id tlf
```

7.4 Distributive comonads and the new scheme

In order to introduce distributive comonads, we have to use either multi-parameter classes or rank-2 type signatures. Since distributive comonads are comonads in a certain relation to a functor, it looks natural to define them as a multi-parameter class:

```
class (Functor f, Comonad n) => DistComonad f n where
  dist :: f (n a) -> n (f a)
```

And then we could implement g-catamorphisms as the cancellation law (32) teaches us:

```
gcata :: DistComonad f n => (f (n c) -> c) -> Mu f -> c
gcata phi
        = phi . fmap (fmap (gcata phi) . iota) . unIn
             where iota = cata (fmap In . dist)
```

This choice, however, implies that we can never simultaneously work with more than one distributive law relating a particular comonad to a particular functor unless we are willing to rename the comonad.

An alternative avoiding this not so pleasant restriction consists in supplying the distributive law in an additional argument of the `gcata` combinator. Here we have to use rank-2 type signatures in order to correctly state the polymorphic type signature of this additional argument:

```
> gcata :: (Functor f, Comonad n) =>
>                  (forall a. f (n a) -> n (f a))
>                       -> (f (n c) -> c) -> Mu f -> c
> gcata dist phi
>         = phi . fmap (fmap (gcata dist phi) . iota) . unIn
>              where iota = cata (fmap In . dist)
```

Instead of relying on the g-catamorphism cancellation law (32), we may, at will, use the explicit definition (35) of `gcata` via the `cata` combinator:

```
gcata dist phi = extr . cata (fmap phi . dist . fmap dupl)
```

While the two definitions are denotationally equivalent, their operational characteristics are different. Which definition is better in terms of efficiency depends on the concrete comonad. The rule of thumb is that the definition via `cata` is more efficient. An important exception, however, is the comonad `Prod (Mu f)`, which captures the recursive call structure of primitive recursion. For this comonad, the cancellation law is the right choice.

EXAMPLE 15. General g-catamorphisms already defined, zygomorphisms are most simply defined as specific g-catamorphisms:

```
> zygo :: Functor f => (f e -> e)
>                        -> (f (Prod e c) -> c) -> Mu f -> c
> zygo chi = gcata (fork (fmap outl) (chi . fmap outr))
```

This, however, is not the most efficient definition. The options are to rely on the zygomorphism cancellation law:

```
zygo chi phi
       = phi . fmap (fork (zygo chi phi) (cata chi)) . unIn
```

or on the explicit definition of the `zygo` combinator via the `cata` combinator:

```
zygo chi phi = outl . cata (fork phi (chi . fmap outr))
```

Paramorphisms are now most easily defined as special zygomorphisms:

```
> para :: Functor f => (f (Prod (Mu f) c) -> c) -> Mu f -> c
> para = zygo In
```

but again we might equivalently choose to rely on the paramorphism laws.

The factorial function is expressed as a paramorphism as follows:

```
> fact :: Nat -> Nat
> fact = para phi
>   where phi Z              = succ zero
>         phi (S (Pair f n)) = mult f (succ n)
```

The first equation in the definition of `phi` defines `fact zero` [`phi Z`] and the second defines `fact (succ n)` [`phi (S (Pair f n))`] in terms of `fact n` [`f`] and `n`.

EXAMPLE 16. To define g-histomorphisms, we again need rank-2 type signatures. The easiest option is defining g-histomorphisms as special g-catamorphisms:

```
> ghisto :: (Functor f, Functor h) =>
>                  (forall a. f (h a) -> h (f a))
>                        -> (f (Strf h c) -> c) -> Mu f -> c
> ghisto chi = gcata (genStrf (fmap hdf) (chi . fmap tlf))
```

Histomorphisms in their turn are most easily defined as a special case of g-histomorphisms:

```
> histo :: Functor f => (f (Strf f c) -> c) -> Mu f -> c
> histo = ghisto id
```

The Fibonacci function is defined as a histomorphism as follows:

```
> fibo :: Nat -> Int
> fibo = histo phi
>   where phi Z    = zero
>         phi (S (Consf f fs))
>                  = case fs of
>                      Z               -> succ zero
>                      S (Consf f' _) -> add f f'
```

Here, the first equation for `phi` defines `fibo zero` [`phi Z`] and the second defines `fibo (succ n)` [`phi (S (Consf f fs))`] in terms of `fibo n` [`f`] and, provided `n == succ n'` [`fs == S (Consf f' _)`], also `fibo n'` [`f'`].

8. Related work

The idea of combining monads or comonads with catamorphisms is not new. Fokkinga [1994] (see also [Meijer and Jeuring 1995]) studied a combination of monads with catamorphisms, termed monadic catamorphisms. Given a monad $\boldsymbol{M}$ with a distributive law of F over $\boldsymbol{M}$, their scheme takes a monadic step function $\varphi : FA \to MA$ and produces a monadic induced function $f : \mu F \to MA$. Pardo [2000] (dualizing the combination of monads with anamorphisms of [Pardo 2001]) combined comonads with catamorphisms. For a comonad $\boldsymbol{N}$ with a distributive law of $\boldsymbol{N}$ over F, his scheme takes comonadic step functions $\varphi : NFA \to A$ to comonadic induced functions $f : N\mu F \to A$, but has the drawback that the induced function is not, in general, determined uniquely. An example of a well-behaving comonad, again, is the product comonad, which yields iteration with parameters.

A step towards the scheme of the present paper was taken by Lenisa [1999]. She, similarly to Pardo [2001], studied monads in relation to corecursion. Modulo duality, she recorded the straightforward fact that, given a comonad $\boldsymbol{N}$, any morphism $\varphi' : FNA \to NA$ determines a morphism $f : \mu F \to A$, namely $f = \varepsilon_A \circ (\!|\, \varphi' \,|\!)_F$. She also pointed out the relation of the product comonad in this context to primitive recursion. She did not, however, require F to distribute over $\boldsymbol{N}$ and without this, it is not possible to give an explicit definition and characterization for f in terms of $\varphi = \varepsilon_A \circ \varphi' : FNA \to A$ as done here. In [Lenisa *et al.* 2000], modulo duality again, distributive laws were added, but comonads were weakened to copointed endofunctors.

ADDED IN PROOF: In continuation of Lenisa's line of work, the dual of the scheme discussed in this paper was recently (independently of us and approximately simultaneously) justified by Bartels [2001]. He also introduced an alternative version of the scheme where—at the expense of extra assumptions on the category—no (co)monad structure is needed.

9. Conclusion and future work

Using the notion of comonad, we have defined a "many-in-one" recursion scheme that covers a variety of different strengthenings of iteration. In this general scheme of recursion, comonads package different manners in which particular recursion schemes allow recursive calls in evaluation. We have found it very pleasant that all differences between schemes as distant as primitive recursion and course-of-value iteration may be neatly abstracted and the outcome is one single light-weight skeleton scheme. The scheme presents a novel use of comonads as a program structuring device. That the scheme is natural indeed and conduces modular programming is demonstrated by the Haskell code presented in Section 7.

The goals for future work include the following. To start with, we wish to obtain further confidence in the usefulness of the new scheme in its generality by working out its specializations for other comonads. Second, we also wish to check examples of the dual generalization of coiteration. This looks promising for the reason that in **Set**-like categories, where exponents exist but coexponents do not, there are more monads than comonads.

A third direction would be to find out if and how the present work may be restated for the so-called Mendler style of formulating recursion schemes [Uustalu and Vene 1999b, 2000]. As Mendler-style formulations tend to be simpler than formulations in the conventional style, there is hope that the constructions of the present paper, especially those related to distributivity, may admit a smoother restatement in the Mendler style. And finally, since the recursion scheme considered here and the comonadic folds of Pardo [2000] use distributive laws in opposite directions and also correspond to complementary dimensions along which iteration may be strengthened, it ought to be worthwhile to carry out a detailed comparison of the two approaches and see if these can be combined.

Acknowledgements

We are grateful to our two anonymous referees for valuable remarks.

The research of the two first authors was partially supported by the Estonian Science Foundation under grant No. 4155. The first author received support also from the Portuguese Foundation for Science and Technology under grant No. PRAXIS XXI/C/EEI/14172/98. The second author's participation at NWPT'00, where this work was first presented, was financed by the organization of the workshop.

References

Backhouse, R., Jansson, P., Jeuring, J., and Meertens, L. 1999. Generic Programming –An Introduction–. In *Revised Lectures 3rd Int. School on Advanced Functional Programming, AFP'98* (Braga, Sept. 1998), Swierstra, S.D., Henriques, P.R, and Oliveira, J.N., Editors, Volume 1608 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 28–115.

Barr, M. and Wells, C. 1984. *Toposes, Triples and Theories*, Volume 278 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, Berlin. (Revised Electronic Edition of 2000 available at `ftp://ftp.math.mcgill.ca/pub/barr/`.)

Barr, M. and Wells, C. 1990. *Category Theory for Computing Science*, *Prentice Hall Int. Series in Computer Science*. Prentice Hall, London. (2nd Edition, 1995, same publisher; 3rd Edition, 1999, Centre de Recherches Mathématiques, Montréal.)

Bartels, F. 2001. Generalized Coinduction. In *Proceedings 4th Workshop on Coalgebraic Methods in Computer Science, CMCS'01* (Genova, Apr. 2001), Corradini, A., Lenisa, M., Montanari, U., Editors, Volume 44(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam.

Beck, J. 1969. Distributive Laws. In *Seminar on Triples and Categorical Homology Theory* (ETH, 1966/67), Eckmann, B., Editor, Volume 80 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, 119–140.

Bird, R. and de Moor, O. 1997. *Algebra of Programming*, *Prentice Hall Int. Series in Computer Science*. Prentice Hall, London.

Bird, R. 1998. *Introduction to Functional Programming Using Haskell*, *Prentice Hall Int. Series in Computer Science*. Prentice Hall, London.

FOKKINGA, M. M. 1992. Law and Order in Algorithmics. PhD Thesis, Dept. of Informatics, Univ. of Twente.

FOKKINGA, M. M. 1994. Monadic Maps and Folds for Arbitrary Datatypes. Memoranda Informatica 94-28, Dept. of Informatics, Univ. of Twente.

JONES, M. P. 1995. Simplifying and Improving Qualified Types. In *Conf. Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, (La Jolla, CA, June 1995). ACM Press, New York, 160–169.

KIEBURTZ, R. 1999. Codata and Comonads in Haskell. Manuscript.

LENISA, M. 1999. From Set-Theoretic Coinduction to Algebraic Coinduction: Some Results, Some Problems. In *Proceedings 2nd Workshop on Coalgebraic Methods in Computer Science, CMCS'99* (Amsterdam, March 1999), Jacobs, B. and Rutten, J., Editors, Volume 19 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam.

LENISA, M., POWER, J., AND WATANABE, H. 2000. Distributivity for Endofunctors, Pointed and Co-pointed Endofunctors, Monads and Comonads. In *Proceedings 3rd Workshop on Coalgebraic Methods in Computer Science, CMCS'00* (Berlin, March 2000), Reichel, H., Editor, Volume 33 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam.

MALCOLM, G. R. 1990. Algebraic Data Types and Program Transformation. PhD Thesis, Dept. of Computer Science, Univ. of Groningen.

MANES, E.G. 1976. *Algebraic Theories*, Volume 26 of *Graduate Texts in Mathematics*, Springer-Verlag, New York.

MCCRACKEN, N.J. 1984. The Typechecking of Programs with Implicit Type Structure. In *Proceedings Int. Symp. on the Semantics of Data Types* (Sophia-Antipolis, June 1984), Kahn, G., MacQueen, D. B., and Plotkin, G., Editors, Volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 301–315.

MEERTENS, L. 1992. Paramorphisms. *Formal Aspects of Computing 4*, 5, 413–424.

MEIJER, E. AND JEURING, J. 1995. Merging Monads and Folds for Functional Programming. In *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, AFP'95* (Båstad, May 1995), Jeuring, J. and Meijer, E., Editors, Volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 228–266.

MOGGI, E. 1991. Notions of Computation and Monads. *Information and Computation 93*, 1, 55–92.

PARDO, A. 2000. Towards Merging Recursion and Comonads. In *Proceedings 2nd Workshop on Generic Programming, WGP'00* (Ponte de Lima, July 2000), Jeuring, J., Editor, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ., 50–68.

PARDO, A. 2001. Fusion of Recursive Programs with Computational Effects. *Theoretical Computer Science 260*, 1–2, 165–207.

PEYTON JONES, S. AND HUGHES, J., EDITORS. 1999. Report on the Programming Language Haskell 98, A Non-Strict Purely Functional Language. Research Report YALEU-DCS-RR-1106, Dept. of Computer Science, Yale Univ.

TURNER, D.A. 1995. Elementary Strong Functional Programming. In *Proceedings 1st Int. Symp. on Functional Programming Languages in Education, FPLE'95* (Nijmegen, Dec. 1995), Plasmeijer, R. and Hartel, P., Editors, Volume 1022 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1–13.

UUSTALU, T. AND VENE, V. 1999a. Primitive (Co)recursion and Course-of-Value (Co)iteration, Categorically. *INFORMATICA 10*, 1, 5–26.

UUSTALU, T. AND VENE, V. 1999b. Mendler-Style Inductive Types, Categorically. *Nordic Journal of Computing 6*, 3, 343–361.

UUSTALU, T. AND VENE, V. 2000. Coding Recursion a la Mendler. In *Proceedings 2nd Workshop on Generic Programming, WGP'00* (Ponte de Lima, July 2000), Jeuring, J., Editor, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ., 69–85.

Vene, V. 2000. Categorical Programming with Inductive and Coinductive Types. PhD Thesis, Diss. Math. Univ. Tartuensis 23, Inst. of Computer Science, Univ. of Tartu.

Wadler, P. 1992. The Essence of Functional Programming. In *Conf. Record 19th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'92* (Albuquerque, Jan. 1992). ACM Press, New York, 1–12.

Wechler, W. 1992. *Universal Algebra for Computer Scientists*, Volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin.