Functional Programming with Apomorphisms / Corecursion

Varmo Vene*
University of Tartu, Estonia

Tarmo Uustalu†
Royal Institute of Technology, Stockholm, Sweden

Abstract

In the mainstream categorical approach to typed (total) functional programming, functions with inductive source types defined by primitive recursion are called paramorphisms; the utility of primitive recursion as a scheme for defining functions in programming is well-known. We draw attention to the dual notion of apomorphisms—functions with coinductive target types defined by primitive corecursion—, and show on examples that primitive corecursion is useful in programming.

1 Introduction

This paper is about the categorical approach to typed (total) functional programming where datatypes and codatatypes are modelled as initial algebras and terminal coalgebras. Our object of study is the notion of apomorphisms, which are functions with coinductive target types defined by primitive corecursion. Apomorphisms are the dual of paramorphisms, functions with inductive source types defined by primitive corecursion. The widely used term paramorphism is due to Meertens [Mee92] ($\pi\alpha\rho\alpha$ — greek preposition meaning 'near to', 'at the side of', 'towards'), the term apomorphism is a novel invention of ours ($\alpha\pi o$ — greek preposition meaning 'apart from', 'far from', 'away from'). Our aim is to show that apomorphisms are a convenient tool for a declaratively thinking programmer, often more handy than anamorphisms, i.e. functions defined by coiteration, in much the same way as paramorphisms are frequently superior to catamorphisms, i.e. functions defined by iteration.

The tradition of categorical functional programming that we follow in this paper started off with the Bird-Meertens formalism [Bir87], which is a theory of the datatypes of lists. Malcolm [Mal90] and Fokkinga [Fok92] generalized the approach for arbitrary datatypes and codatatypes, inspired by the work of Hagino [Hag87].

Geuvers [Geu92] contains a thorough theoretical analysis of primitive recursion versus iteration and a demonstration that this readily dualizes into an analysis of primitive corecursion versus coiteration. In general, however, it appears that primitive corecursion has largely been overlooked in the theoretical literature, e.g. Fokkinga [Fok92] ignores it; neither do we know of any papers containing interesting programming examples, the most probable reason being the young age of programming with codatatypes.

We shall proceed as follows. In section 2, we give a short introduction to the categorical approach to (co)datatypes and (co)iteration and to its application to program calculation. In

*Arvutiteaduse Instituut, Tartu Ülikool, J. Liivi 2, EE-2484 Tartu, Estonia; varmo@cs.ut.ee
†Institutionen för Teleinformatik, Kungliga Tekniska Högskolan, Electrum 204, SE-164 40 Kista, Sweden; tarmo@it.kth.se

section 3, we introduce the scheme of primitive recursion for inductive types and prove several properties it has. Afterwards, we dualize these results for the coinductive types case, by introducing primitive corecursion and show the usefulness of this scheme on several examples. In section 4, we show possible directions for further work and conclude.

Proofs in this paper are carried out in the structured calculational proof style [Gru96].

2 (Co)datatypes and (co)iteration

Preliminaries

Throughout the paper $\mathcal{C}$ is the default category, in which we shall assume the existence of finite products $(\times, 1)$ and co-products $(+, 0)$, as well as the distributivity of products over coproducts (i.e. $\mathcal{C}$ is *distributive*). The typical example of a distributive category is **Sets** — the category of sets and total functions.

We make use of the following quite standard notation. Given two objects A, B, we write $\mathtt{fst} : A \times B \to A$ and $\mathtt{snd} : A \times B \to B$ to denote the left and right projections for the product $A \times B$. For $f : C \to A$ and $g : C \to B$, pairing is the unique morphism $\langle f, g \rangle : C \to A \times B$, such that $\mathtt{fst} \circ \langle f, g \rangle = f$ and $\mathtt{snd} \circ \langle f, g \rangle = g$. The left and right injections for the coproduct $A + B$ are $\mathtt{inl} : A \to A + B$ and $\mathtt{inr} : B \to A + B$. For $f : A \to C$ and $g : B \to C$, case analysis is the unique morphism $[f, g] : A + B \to C$, such that $[f, g] \circ \mathtt{inl} = f$ and $[f, g] \circ \mathtt{inr} = g$. Besides, given an object C, we have the unique morphism $!_C : C \to 1$. The inverse of the canonical map $[\mathtt{inl} \times \mathtt{id}_C, \mathtt{inr} \times \mathtt{id}_C] : (A \times C) + (B \times C) \to (A + B) \times C$ is denoted by $\mathtt{distr} : (A + B) \times C \to (A \times C) + (B \times C)$. Finally, given a predicate $p : A \to 1 + 1$, the guard $p? : A \to A + A$ is defined as $(\mathtt{snd} + \mathtt{snd}) \circ \mathtt{distr} \circ \langle p, \mathtt{id}_A \rangle$.

The identity functor is denoted by I. For a functor $F : \mathcal{C} \to \mathcal{C}$, its application to a morphism $f : A \to B$ is denoted by $F f : F A \to F B$.

Dialgebras

Categorically, datatypes (of natural numbers, lists, etc.) are traditionally modelled by initial algebras and codatatypes (of conatural numbers, colists, streams, etc.) by terminal coalgebras. Hagino [Hag87] introduced the notion of dialgebras, which generalises both concepts and is rich enough also to model many-sorted (co)algebras and bialgebras [Fok92].

Definition 1 *Let $F, G : \mathcal{C} \to \mathcal{C}$ be endofunctors.*

- F, G-dialgebra *is a pair (A, φ), where A is an object and $\varphi : F A \to G A$ is a morphism of the default category $\mathcal{C}$.*
- *Let (A, φ) and (B, ψ) be F, G-dialgebras. A morphism $f : A \to B$ of $\mathcal{C}$ is a F, G-*homomorphism *from (A, φ) to (B, ψ), if the following diagram commutes.*

$$\begin{tikzcd} F A \arrow[r, "\varphi"] \arrow[d, "F f"] & G A \arrow[d, "G f"] \\ F B \arrow[r, "\psi"] & G B \end{tikzcd}$$

(i.e. $G f \circ \varphi = \psi \circ F f$).

It is easy to verify that the composition of two F, G-homomorphisms is also a F, G-homomorphism. Moreover, the identity morphism $\mathtt{id}_A : A \to A$ is a F, G-homomorphism which is both a left and a right unit with respect to composition. It follows that dialgebras and homomorphisms form a category.

Definition 2 *Let $F, G : \mathcal{C} \to \mathcal{C}$ be endofunctors.*

- *A category of F,G-dialgebras $\mathcal{D}i\mathcal{A}lg(F,G)$ is a category where objects are F,G-dialgebras and morphisms are F,G-homomorphisms. Composition and identities are inherited from $\mathcal{C}$. The categories of F-algebras and G-coalgebras are defined as $\mathcal{A}lg(F) = \mathcal{D}i\mathcal{A}lg(F,I)$ and $\mathcal{C}o\mathcal{A}lg(G) = \mathcal{D}i\mathcal{A}lg(I,G)$, respectively.*

- *A F,G-dialgebra is* initial *(resp.* terminal*), if it is initial (resp. terminal) (as an object) in the category of F,G-dialgebras $\mathcal{D}i\mathcal{A}lg(F,G)$. The initial F-algebra is denoted by $(\mu F, \mathtt{in}_F)$, the terminal G-coalgebra is denoted by $(\nu G, \mathtt{out}_G)$.*

Note that initial and terminal dialgebras may or may not exist. In fact, we even do not know any simple criteria for their existence in the general dialgebras case. In the case of algebras, an initial F-algebra is guaranteed to exist if a functor F is ω-cocontinuous[1]. Dually, if a functor G is ω-continuous, then a terminal G-coalgebra is guaranteed to exist. All polynomial functors (i.e. functors which are built up from products, sums, identity and constant functors) are ω-(co)continuous and, consequently, the corresponding initial algebras and terminal coalgebras exist. We do not pursue this question any further here and refer the interested reader to [Fok92].

Initial algebras and catamorphisms

The existence of the initial F-algebra $(\mu F, \mathtt{in}_F)$ means that, for any F-algebra (C, ψ), there exists a unique homomorphism from $(\mu F, \mathtt{in}_F)$ to (C, ψ). Following Fokkinga [Fok92], we denote this homomorphism by $(\!|\, \psi \,|\!)_F$, so $(\!|\, \psi \,|\!)_F : \mu F \to C$ is characterized by the universal property

$$f \circ \mathtt{in}_F = \psi \circ F\, f \quad \equiv \quad f = (\!|\, \psi \,|\!)_F \qquad\qquad \text{cata-}\textsc{Charn}$$

The type information is summarized in the following diagram:

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

Morphisms of the form $(\!|\, \psi \,|\!)_F$ are called *F-catamorphisms* (derived from the greek preposition $\kappa\alpha\tau\alpha$ meaning 'downwards'); the construction $(\!|\, \cdot \,|\!)_F$ is an iterator.

Example 1 *Consider the datatype Nat of natural numbers, with constructor functions zero* $: 1 \to$ *Nat and succ* $:$ *Nat* $\to$ *Nat. It can be represented as an initial N-algebra $(\mathit{Nat}, [\,\mathit{zero}, \mathit{succ}\,]) = (\mu N, \mathtt{in}_N)$, where $N\,X = 1 + X$ and $N\,f = \mathtt{id}_1 + f$. Given any N-algebra $(C, [\,c, h\,])$, the catamorphism $f = (\!|\, [\,c, h\,] \,|\!)_N$ is the iteration $f(n) = h^n(c)$. For instance, the curried sum of two naturals can be defined as*

$$\mathit{add}\; n = (\!|\, [\,n, \mathit{succ}\,] \,|\!)_N .$$

Example 2 *The datatype List_A of finite lists over a given set A, with constructor functions nil* $: 1 \to \mathit{List}_A$ *and cons* $: A \times \mathit{List}_A \to \mathit{List}_A$*, can be represented as an initial algebra $(\mathit{List}_A, [\,\mathit{nil}, \mathit{cons}\,]) = (\mu_{L_A}, \mathtt{in}_{L_A})$, where $L_A\,X = 1 + (A \times X)$ and $L_A\,f = \mathtt{id}_1 + (\mathtt{id}_A \times f)$. Given any L_A-algebra $(C, [\,c, h\,])$, the catamorphism $f = (\!|\, [\,c, h\,] \,|\!)_{L_A}$ is an application of the standard fold function on lists. For any function $g : A \to B$, the function map $g : \mathit{List}_A \to \mathit{List}_B$, for instance, can be defined as*

$$\mathit{map}\; g = (\!|\, [\,\mathit{nil}, \mathit{cons} \circ (g \times \mathtt{id}_{\mathit{List}_A})\,] \,|\!)_{L_A} .$$

Catamorphisms obey several nice laws, of which the *cancellation* law, the *reflection* law (also known as the identity law) and the *fusion* (or, promotion) law are especially important for program

[1] A functor is ω-(co)continuous iff it preserves the (co)limits of ω-chains.

calculation (the cancellation law describing a certain program execution step):

$$
\begin{array}{lll}
(\!|\, \psi \,|\!)_F \circ \mathtt{in}_F = \psi \circ F(\!|\, \psi \,|\!)_F & & \text{cata-}\textsc{Cancel} \\
\mathtt{id}_{\mu F} = (\!|\, \mathtt{in}_F \,|\!)_F & & \text{cata-}\textsc{Refl} \\
f \circ \varphi = \psi \circ F\, f & \Rightarrow \quad f \circ (\!|\, \varphi \,|\!)_F = (\!|\, \psi \,|\!)_F & \text{cata-}\textsc{Fusion}
\end{array}
$$

All three follow directly from the characterization cata-CHARN. These laws (originally for the special case of lists) form the heart of the Bird-Meertens formalism.

To show the usefulness of above mentioned laws we prove the following lemma (first recorded by Lambek [Lam68]):

Lemma 1 *The morphism* $\mathtt{in}_F : F\,\mu F \to \mu F$ *is an isomorphism with the inverse* $\mathtt{in}_F^{-1} = (\!|\, F\,\mathtt{in}_F \,|\!)_F : \mu F \to F\,\mu F$.

Proof:

$$
\begin{array}{ll}
 & \mathtt{in}_F \circ (\!|\, F\,\mathtt{in}_F \,|\!)_F \\
= & \quad - \text{ cata-}\textsc{Fusion} - \\
 & (\!|\, \mathtt{in}_F \,|\!)_F \\
= & \quad - \text{ cata-}\textsc{Refl} - \\
 & \mathtt{id}_{\mu F}
\end{array}
$$

$$
\begin{array}{ll}
 & (\!|\, F\,\mathtt{in}_F \,|\!)_F \circ \mathtt{in}_F \\
= & \quad - \text{ cata-}\textsc{Cancel} - \\
 & F\,\mathtt{in}_F \circ F\,(\!|\, F\,\mathtt{in}_F \,|\!)_F) \\
= & \quad - F \text{ functor } - \\
 & F\,(\mathtt{in}_F \circ (\!|\, F\,\mathtt{in}_F \,|\!)_F) \\
= & \quad - \text{ above } - \\
 & F\,\mathtt{id}_{\mu F} \\
= & \quad - F \text{ functor } - \\
 & \mathtt{id}_{F\,\mu F}
\end{array}
$$

Terminal coalgebras and anamorphisms

The existence of the terminal G-coalgebra $(\nu G, \mathtt{out}_G)$ means that for any other G-coalgebra (C, φ), there exists a unique homomorphism from (C, φ) to $(\nu G, \mathtt{out}_G)$. This homomorphism is usually denoted by $[\!(\, \varphi \,)\!]_G$, so $[\!(\, \varphi \,)\!]_G : C \to \nu G$ is characterized by the universal property

$$
\mathtt{out}_G \circ f = G\, f \circ \varphi \quad \equiv \quad f = [\!(\, \varphi \,)\!]_G \qquad\qquad \text{ana-}\textsc{Charn}
$$

The type information is summarized in the following diagram:

$$
\begin{tikzcd}
C \arrow[r, "\varphi"] \arrow[d, "f"'] & G\,C \arrow[d, "G\,f"] \\
\nu G \arrow[r, "\mathtt{out}_G"] & G\,\nu G
\end{tikzcd}
$$

Morphisms of the form $[\!(\, \varphi \,)\!]_G$ are called G-*anamorphisms* (derived from the greek preposition $\alpha\nu\alpha$ meaning 'upwards'; the name is due to Meijer), and the construction $[\!(\, \cdot \,)\!]_G$ is a coiterator. Like catamorphisms, anamorphisms enjoy various properties including the following cancellation, reflection, and fusion laws:

$$
\begin{array}{lll}
\mathtt{out}_G \circ [\!(\, \varphi \,)\!]_G = G\,[\!(\, \varphi \,)\!]_G \circ \varphi & & \text{ana-}\textsc{Cancel} \\
\mathtt{id}_{\nu G} = [\!(\, \mathtt{out}_G \,)\!]_G & & \text{ana-}\textsc{Refl} \\
\psi \circ f = G\, f \circ \varphi & \Rightarrow \quad [\!(\, \psi \,)\!]_G \circ f = [\!(\, \varphi \,)\!]_G & \text{ana-}\textsc{Fusion}
\end{array}
$$

Also, $\mathtt{out}_G$ is an isomorphism with the inverse $\mathtt{out}_G^{-1} = [\!(\, G\,\mathtt{out}_G \,)\!]_G$.

Example 3 *Consider the codatatype* Stream_A *of streams (infinite lists) over* A*, with destructor functions* $\mathit{head} : \mathit{Stream}_A \to A$ *and* $\mathit{tail} : \mathit{Stream}_A \to \mathit{Stream}_A$*. It can be represented as a terminal coalgebra* $(\mathit{Stream}_A, \langle\, \mathit{head}, \mathit{tail} \,\rangle) = (\nu S_A, \mathtt{out}_{S_A})$*, where* $S_A\, X = A \times X$ *and* $S_A\, f = \mathtt{id}_A \times f$*. Anamorphisms are used to construct concrete streams. For instance the stream of natural numbers* *nats* $: 1 \to Stream_{Nat}$ *and 'zipping' of two streams* $zip : Stream_A \times Stream_B \to Stream_{A\times B}$ *can be defined as follows:*

$$
\begin{aligned}
nats &= [\!(\, \langle\, \mathtt{id}_{Nat}, succ \,\rangle \,)\!]_{S_{Nat}} \circ zero \\
zip &= [\!(\, \langle\, \mathtt{fst} \times \mathtt{fst}, \mathtt{snd} \times \mathtt{snd} \,\rangle \circ (\mathtt{out}_{S_A} \times \mathtt{out}_{S_B}) \,)\!]_{S_{A\times B}}
\end{aligned}
$$

Example 4 *The codatatype of colists (possibly infinite lists)* $List'_A$ *can be represented as a terminal coalgebra* $(\nu L_A, \mathtt{out}_{L_A})$ *of the functor* $L_A(X) = 1 + A \times X$. *Given a predicate* $empty : List'_A \to 1+1$ *and two partial functions* $head' : List'_A \to A$ *and* $tail' : List'_A \to List'_A$*, the terminal coalgebra morphism* $\mathtt{out}_{L_A}$ *can be defined as* $\mathtt{out}_{L_A} = (! + \langle\, head', tail' \,\rangle) \circ empty?$. *Anamorphisms for* $List'_A$ *correspond to applications of the unfold function from functional programming.*

3 Primitive (co-)recursion

Clearly, not every morphism $f : \mu F \to C$ is a catamorphism by itself, which means that the results of previous section cannot be applied as they stand. For instance, the factorial function specified by equations

$$
\begin{aligned}
fact(0) &= 1 \\
fact(n+1) &= (n+1) * fact(n)
\end{aligned}
$$

is not a catamorphism. The problem here is that, to compute the value of the factorial for a given argument, one needs not only its value for the previous argument, but also the previous argument itself. Meertens [Mee92] has shown that any function which can be characterized similarly to the factorial function is definable as the composition of left projection and a catamorphism. The relevant result is the following:

Lemma 2 *Given* $f : \mu F \to C$ *and* $\psi : F\,(C \times \mu F) \to C$,

$$
f \circ \mathtt{in}_F = \psi \circ F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle \qquad \equiv \qquad f = \mathtt{fst} \circ (\!|\, \langle\, \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \,\rangle \,|\!)_F .
$$

Proof:

$$
\begin{array}{ll}
\rhd & f \circ \mathtt{in}_F = \psi \circ F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle \\
\hline
 & f \\
= & \quad \text{– pairing –} \\
 & \mathtt{fst} \circ \langle\, f, \mathtt{id}_{\mu F} \,\rangle \\
= & \quad \text{– cata-}\textsc{Charn}\text{ –} \\
 & \begin{array}{ll}
 & \langle\, f, \mathtt{id}_{\mu F} \,\rangle \circ \mathtt{in}_F \\
 = & \quad \text{– pairing –} \\
 & \langle\, f \circ \mathtt{in}_F, \mathtt{in}_F \,\rangle \\
 = & \quad \text{– } F \text{ functor –} \\
 & \langle\, f \circ \mathtt{in}_F, \mathtt{in}_F \circ F\,\mathtt{id}_{\mu F} \,\rangle \\
 = & \quad \text{– pairing –} \\
 & \langle\, f \circ \mathtt{in}_F, \mathtt{in}_F \circ F\,(\mathtt{snd} \circ \langle\, f, \mathtt{id}_{\mu F} \,\rangle) \,\rangle \\
 = & \quad \text{– } \lhd,\ F \text{ functor –} \\
 & \langle\, \psi \circ F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle, \mathtt{in}_F \circ F\,\mathtt{snd} \circ F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle \,\rangle \\
 = & \quad \text{– pairing –} \\
 & \langle\, \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \,\rangle \circ F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle
 \end{array} \\
 & \mathtt{fst} \circ (\!|\, \langle\, \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \,\rangle \,|\!)_F
\end{array}
$$

$$
\begin{array}{ll}
\rhd & f = \mathtt{fst} \circ (\!|\, \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \,|\!)_F \\
\hline
 & f \circ \mathtt{in}_F \\
= & \quad - \lhd - \\
 & \mathtt{fst} \circ (\!|\, \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \,|\!)_F \circ \mathtt{in} \\
= & \quad - \text{cata-}\textsc{Cancel} - \\
 & \mathtt{fst} \circ \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \circ F\,(\!|\, \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \,|\!)_F \\
= & \quad - \text{pairing, 2x} - \\
 & \psi \circ F\,\langle\, \mathtt{fst} \circ (\!|\, \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \,|\!)_F, \mathtt{snd} \circ (\!|\, \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \,|\!)_F \,\rangle \\
= & \quad - \lhd, \text{cata-}\textsc{Fusion} - \\
 & \quad \left[\begin{array}{ll} & \mathtt{snd} \circ \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \\ = & \quad - \text{pairing} - \\ & \mathtt{in}_F \circ F\,\mathtt{snd} \end{array} \right. \\
 & \psi \circ F\,\langle\, f, (\!|\, \mathtt{in}_F \,|\!)_F \,\rangle \\
= & \quad - \text{cata-}\textsc{Refl} - \\
 & \psi \circ F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle
\end{array}
$$

The left hand side of the equivalence corresponds exactly to the specification by primitive recursion. For instance, for the factorial function specification above, it reads

$$
\begin{array}{ll}
 & \mathit{fact} \circ \mathtt{in}_N \\
= & [\, \mathit{succ} \circ \mathit{zero}, \mathit{mult} \circ (\mathtt{id}_{Nat} \times \mathit{succ}) \,] \circ N\,\langle\, \mathit{fact}, \mathtt{id}_{\mu N} \,\rangle \\
= & [\, \mathit{succ} \circ \mathit{zero}, \mathit{mult} \circ (\mathit{fact} \times \mathit{succ}) \,]
\end{array}
$$

and consequently we get the definition of factorial

$$
\mathit{fact} = \mathtt{fst} \circ (\!|\, \langle\, [\, \mathit{succ} \circ \mathit{zero}, \mathit{mult} \circ (\mathtt{id}_{Nat} \times \mathit{succ}) \,], \mathtt{in}_N \circ N\,\mathtt{snd} \,\rangle \,|\!)_N .
$$

From the lemma above, it follows that at least every primitive recursive function can be represented using catamorphism as the only recursive construction. In the presence of exponentials, one can even define Ackermann's function as a (higher-order) catamorphism, so the expressive power of the "language of catamorphisms" is bigger than the class of primitively recursive functions. In fact, Howard [How96] has shown that the functions expressible in simply typed λ-calculus extended with inductive and coinductive types are precisely those provably total in the logic $ID_{<\omega}$ (the first order arithmetic augmented by finitely-iterated inductive definitions).

Paramorphisms

To make programming and program reasoning easier, let us introduce a new construction and find out its properties. For any morphism $\psi : F\,(C \times \mu F) \to C$, define a morphism $\langle\!|\, \psi \,|\!\rangle_F : \mu F \to C$ by letting

$$
\langle\!|\, \psi \,|\!\rangle_F = \mathtt{fst} \circ (\!|\, \langle \psi, \mathtt{in}_F \circ F\,\mathtt{snd} \rangle \,|\!)_F \qquad \text{para-}\textsc{Def}
$$

Morphisms of the form $\langle\!|\, \psi \,|\!\rangle_F$ are called *F-paramorphisms* (the name is due to Meertens [Mee92]). The construction $\langle\!|\, \cdot \,|\!\rangle_F$ is a primitive recursor.

From lemma 2, we get the characterization of paramorphisms by the following universal property:

$$
f \circ \mathtt{in}_F = \psi \circ F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle \quad \equiv \quad f = \langle\!|\, \psi \,|\!\rangle_F \qquad \text{para-}\textsc{Charn}
$$

The type information is summarized in the following diagram:

$$
\begin{tikzcd}
F\,\mu F \arrow[r, "\mathtt{in}"] \arrow[d, "F\,\langle\, f, \mathtt{id}_{\mu F} \,\rangle"'] & \mu F \arrow[d, "f"] \\
F\,(C \times \mu F) \arrow[r, "\psi"] & C
\end{tikzcd}
$$

Example 5 *The factorial function can be defined as a paramorphism:*

$$fact = \langle\!| \, [\, succ \circ zero, mult \circ (\mathtt{id}_{Nat} \times succ) \,] \, |\!\rangle_N$$

The calculational properties of paramorphisms are similar to those of catamorphisms. In particular, we have "paramorphic" versions of the cancellation, reflection and fusion laws:

$$\begin{array}{llr}
\langle\!| \, \psi \, |\!\rangle_F \circ \mathtt{in}_F = \varphi \circ F \, \langle \, \langle\!| \, \psi \, |\!\rangle_F, \mathtt{id}_{\mu F} \, \rangle & & \text{para-}\textsc{Cancel} \\
\mathtt{id}_{\mu F} = \langle\!| \, \mathtt{in}_F \circ F \, \mathtt{fst} \, |\!\rangle_F & & \text{para-}\textsc{Refl} \\
f \circ \varphi = \psi \circ F \, (f \times \mathtt{id}_{\mu F}) & \Rightarrow \quad f \circ \langle\!| \, \varphi \, |\!\rangle_F = \langle\!| \, \psi \, |\!\rangle_F & \text{para-}\textsc{Fusion}
\end{array}$$

The reflection law is proved by the following calculation:

$$\left[\begin{array}{ll}
 & \mathtt{id}_{\mu F} \\
= & \quad - \text{ para-}\textsc{Charn} - \\
 & \left[\begin{array}{ll}
 & \mathtt{in}_F \\
 = & \quad - \; F \text{ functor} - \\
 & \mathtt{in}_F \circ F \, \mathtt{id}_{\mu F} \\
 = & \quad - \text{ pairing} - \\
 & \mathtt{in}_F \circ F \, (\mathtt{fst} \circ \langle \, \mathtt{id}_{\mu F}, \mathtt{id}_{\mu F} \, \rangle) \\
 = & \quad - \; F \text{ functor} - \\
 & \mathtt{in}_F \circ F \, \mathtt{fst} \circ F \, \langle \, \mathtt{id}_{\mu F}, \mathtt{id}_{\mu F} \, \rangle
 \end{array}\right. \\
 & \langle\!| \, \mathtt{in}_F \circ F \, \mathtt{fst} \, |\!\rangle_F
\end{array}\right.$$

The fusion law is proved as follows:

$$\left[\begin{array}{ll}
\rhd & f \circ \varphi = \psi \circ F \, (f \times \mathtt{id}_{\mu F}) \\
\hline
 & f \circ \langle\!| \, \varphi \, |\!\rangle_F \\
= & \quad - \text{ para-}\textsc{Charn} - \\
 & \left[\begin{array}{ll}
 & f \circ \langle\!| \, \varphi \, |\!\rangle_F \circ \mathtt{in}_F \\
 = & \quad - \text{ para-}\textsc{Cancel} - \\
 & f \circ \varphi \circ F \, \langle \, \langle\!| \, \varphi \, |\!\rangle_F, \mathtt{id}_{\mu F} \, \rangle \\
 = & \quad - \lhd - \\
 & \psi \circ F \, (f \times \mathtt{id}_{\mu F}) \circ F \, \langle \, \langle\!| \, \varphi \, |\!\rangle_F, \mathtt{id}_{\mu F} \, \rangle \\
 = & \quad - \; F \text{ functor} - \\
 & \psi \circ F \, ((f \times \mathtt{id}_{\mu F}) \circ \langle \, \langle\!| \, \varphi \, |\!\rangle_F, \mathtt{id}_{\mu F} \, \rangle) \\
 = & \quad - \text{ pairing} - \\
 & \psi \circ F \, \langle \, f \circ \langle\!| \, \varphi \, |\!\rangle_F, \mathtt{id}_{\mu F} \, \rangle
 \end{array}\right. \\
 & \langle\!| \, \psi \, |\!\rangle_F
\end{array}\right.$$

By definition, every paramorphism is the composition of left projection and a catamorphism. In converse, paramorphisms can be viewed as a generalization of catamorphisms, in the sense that every catamorphism is definable as a certain paramorphism:

$$(\!| \, \psi \, |\!)_F = \langle\!| \, \psi \circ F \, \mathtt{fst} \, |\!\rangle_F \qquad\qquad \text{para-}\textsc{Cata}$$

This property can be verified by the following calculation:

$$\left[\begin{array}{ll}
 & (\!| \, \psi \, |\!)_F \\
= & \quad - \text{ para-}\textsc{Charn} - \\
 & \left[\begin{array}{ll}
 & (\!| \, \psi \, |\!)_F \circ \mathtt{in}_F \\
 = & \quad - \text{ cata-}\textsc{Cancel} - \\
 & \psi \circ F \, (\!| \, \psi \, |\!)_F \\
 = & \quad - \text{ pairing} - \\
 & \psi \circ F \, (\mathtt{fst} \circ \langle \, (\!| \, \psi \, |\!)_F, \mathtt{id}_{\mu F} \, \rangle) \\
 = & \quad - \; F \text{ functor} - \\
 & \psi \circ F \, \mathtt{fst} \circ F \, \langle \, (\!| \, \psi \, |\!)_F, \mathtt{id}_{\mu F} \, \rangle
 \end{array}\right. \\
 & \langle\!| \, \psi \circ F \, \mathtt{fst} \, |\!\rangle_F
\end{array}\right.$$

Apomorphisms

Let us now dualize everything we know about paramorphisms. For any morphism $\varphi : C \to G\,(C + \nu G)$, define a morphism $[\![\,\varphi\,]\!] : C \to \nu G$ as the composition of a certain anamorphism and left injection:

$$[\![\,\varphi\,]\!]_G = [\![\,[\,\varphi, G\,\mathtt{inr} \circ \mathtt{out}_G\,]\,]\!]_G \circ \mathtt{inl} \qquad \text{apo-Def}$$

Let us agree to call morphisms of the form $[\![\,\varphi\,]\!]_G$ *G-apomorphisms.* The construction $[\![\,\cdot\,]\!]_G$ is, of course, a primitive corecursor. The characterizing universal property for apomorphisms is following:

$$\mathtt{out}_G \circ f = G\,[\,f, \mathtt{id}_{\nu G}\,] \circ \varphi \quad \equiv \quad f = [\![\,\varphi\,]\!]_G \qquad \text{apo-Charn}$$

The type information is summarized in the following diagram:

$$\begin{tikzcd} C \arrow[r, "\varphi"] \arrow[d, "f"'] & G\,(C + \nu G) \arrow[d, "G\,[\,f, \mathtt{id}_{\nu G}\,]"] \\ \nu G \arrow[r, "\mathtt{out}"] & G\,\nu G \end{tikzcd}$$

The laws for apomorphisms are just the duals of those for paramorphisms:

$$\begin{array}{lr}
\mathtt{out}_G \circ [\![\,\varphi\,]\!]_G = G\,[\,[\![\,\varphi\,]\!]_G, \mathtt{id}_{\nu G}\,] \circ \varphi & \text{apo-Cancel} \\
\mathtt{id}_{\nu G} = [\![\,G\,\mathtt{inl} \circ \mathtt{out}_G\,]\!]_G & \text{apo-Refl} \\
\psi \circ f = G\,(f + \mathtt{id}_{\nu G}) \circ \varphi \qquad \Rightarrow \quad [\![\,\psi\,]\!]_G \circ f = [\![\,\varphi\,]\!]_G & \text{apo-Fusion}
\end{array}$$

As paramorphisms generalized catamorphisms, apomorphisms are a generalization of anamorphisms:

$$[\![\,\varphi\,]\!]_G = [\![\,G\,\mathtt{inl} \circ \varphi\,]\!]_G \qquad \text{apo-Ana}$$

Example 6 *Consider the function maphd* $f : \mathit{Stream}_A \to \mathit{Stream}_A$*, which maps a function* $f : A \to A$ *onto the head of a stream and leaves the tail of the stream unchanged. Using anamorphisms, it can be defined as follows:*

$$\mathit{maphd}\;\; f = [\![\,[\,\langle\, f \circ \mathit{head}, \mathtt{inr} \circ \mathit{tail}\,\rangle, \langle\, \mathit{head}, \mathtt{inr} \circ \mathit{tail}\,\rangle\,]\,]\!]_{S_A} \circ \mathtt{inl}$$

Clearly, such definition is not very convenient, as the tail of the resulting stream, which is unchanged, is also constructed element by element. At the same time, using apomorphisms, the definition looks as follows:

$$\begin{array}{rcl}
\mathit{maphd}\;\; f & = & [\![\,(f \times \mathtt{inr}) \circ \mathtt{out}_{S_A}\,]\!]_{S_A} \\
 & = & [\![\,\langle\, f \circ \mathit{head}, \mathtt{inr} \circ \mathit{tail}\,\rangle\,]\!]_{S_A}
\end{array}$$

Now, the tail of the stream is returned directly without any inspection inside of it.

Example 7 *Assume that the set* A *is ordered linearly by a relation* $(\leq) : A \times A \to \mathtt{1} + \mathtt{1}$*. The function insert* $: A \times \mathit{Stream}_A \to \mathit{Stream}_A$ *that inserts a given element into a sorted stream so that the result will also be sorted is specified as follows:*

$$\langle\, \mathit{head}, \mathit{tail}\,\rangle \circ \mathit{insert} \circ \langle\, x, \mathit{ys}\,\rangle \quad = \quad \begin{cases} \langle\, x, \mathit{ys}\,\rangle & \text{if } x \leq \mathit{head} \circ \mathit{ys} \\ \langle\, \mathit{head} \circ \mathit{ys}, \mathit{insert} \circ \langle\, x, \mathit{tail} \circ \mathit{ys}\,\rangle\,\rangle & \text{otherwise} \end{cases}$$

In this specification, the input stream is traversed and copied to the output stream elementwise to the point at which the input element is inserted; the remainder of the input stream is entered to the output stream as one whole. Formally, this idea is captured by the following categorical definition:

$$\mathit{insert} \circ \langle\, x, \mathit{ys}\,\rangle = [\![\,[\,\langle\, x \circ !, \mathtt{inr}\,\rangle, (\mathtt{id}_A \times \mathtt{inl})\,] \circ \mathit{test}(x)\,]\!]_{S_A} \circ \mathit{ys}$$

where $\mathit{test}(x) : \mathit{Stream}_A \to \mathit{Stream}_A + S_A\,\mathit{Stream}_A$ *is*

$$\mathit{test}(x) = (\mathtt{fst} + \mathtt{snd}) \circ ((\leq) \circ \langle\, x \circ !, \mathtt{fst} \circ \mathtt{snd}\,\rangle)? \;\circ \langle\, \mathtt{id}_{\mathit{Stream}_A}, \mathtt{out}_{S_A}\,\rangle$$

Example 8 *Consider the function on colists concat* : $List'_A \times List'_A \to List'_A$*, which concatenates two colists. It is naturally representable as an apomorphism which copies the first colist and, while arriving at the end of it, returns the second one:*

$$concat \circ \langle x, y \rangle = [\![[L_A \mathtt{inr} \circ \mathtt{out}_{L_A} \circ y, \mathtt{inr}] \circ L_A \mathtt{inl} \circ \mathtt{out}_{L_A}]\!]_{L_A} \circ x$$

In order to illustrate the utility of the laws introduced above, we show that the concatenation of colists is associative:

$$concat \circ \langle\, concat \circ \langle x, y \rangle, z \rangle = concat \circ \langle x, concat \circ \langle y, z \rangle \rangle$$

First, let us introduce a short hand notation for the apomorphism occuring in the definition of the function *concat*:

$$c(y) = [\![[L_A \mathtt{inr} \circ \mathtt{out}_{L_A} \circ y, \mathtt{inr}] \circ L_A \mathtt{inl} \circ \mathtt{out}_{L_A}]\!]_{L_A}$$

So, the definition of *concat* is equivalent to $concat \circ \langle x, y \rangle = c(y) \circ x$, and the equation we want to prove looks as follows:

$$c(z) \circ c(y) \circ x = c(c(z) \circ y) \circ x$$

The proof is carried out according the following informal plan. First, we twice use the apomorphism characterization to "open" the bodies of both apomorphisms on the left hand side of the equation. Then we "push" all subexpressions as far to the right as possible and simplify the resulting expression. Finally, we "pull" the simplified subexpressions back to the left and use again the characterization of apomorphisms to "close" their bodies. When "pushing" subexpressions through the case operation, we shall repeatedly use the following intermediate result about the functor L_A:

$$L_A [f, \mathtt{id}_{List'_A}] \circ [L_A\, \mathtt{inr} \circ g, \mathtt{inr}] \circ L_A\, \mathtt{inl} = [g, \mathtt{inr}] \circ L_A\, f \qquad (*)$$

The validity of the equation $(*)$ follows directly from the definition of functor L_A and the properties of coproduct:

$$\begin{array}{cl}
 & L_A [f, \mathtt{id}_{List'_A}] \circ [L_A\, \mathtt{inr} \circ g, \mathtt{inr}] \circ L_A\, \mathtt{inl} \\
= & \quad \text{– case analysis –} \\
 & [L_A [f, \mathtt{id}_{List'_A}] \circ L_A \mathtt{inr} \circ g, L_A [f, \mathtt{id}_{List'_A}] \circ \mathtt{inr}] \circ L_A\, \mathtt{inl} \\
= & \quad \text{– } L_A \text{ functor, case analysis, } L_A \text{ functor –} \\
 & [g, L_A [f, \mathtt{id}_{List'_A}] \circ \mathtt{inr}] \circ L_A\, \mathtt{inl} \\
= & \quad \text{– def. of } L_A \text{, case analysis –} \\
 & [g, \mathtt{inr} \circ (\mathtt{id}_A \times [f, \mathtt{id}_{List'_A}])] \circ L_A\, \mathtt{inl} \\
= & \quad \text{– case analysis –} \\
 & [g, \mathtt{inr}] \circ (\mathtt{id}_1 + (\mathtt{id}_A \times [f, \mathtt{id}_{List'_A}])) \circ L_A\, \mathtt{inl} \\
= & \quad \text{– def. of } L_A \text{ –} \\
 & [g, \mathtt{inr}] \circ L_A [f, \mathtt{id}_{List'_A}] \circ L_A\, \mathtt{inl} \\
= & \quad \text{– } L_A \text{ functor, case analysis –} \\
 & [g, \mathtt{inr}] \circ L_A\, f
\end{array}$$

Now, the proof of associativity is the following:

$$
\begin{array}{ll}
 & c(z) \circ c(y) \circ x \\
= & \quad \text{– def. of } c \text{, apo-}\textsc{Charn} \text{ –} \\
 & \begin{array}{ll}
 & \mathtt{out}_{L_A} \circ c(z) \circ c(y) \\
= & \quad \text{– def. of } c \text{, apo-}\textsc{Cancel} \text{ –} \\
 & L_A\,[\,c(z), \mathtt{id}_{List'_A}\,] \circ [\,L_A\,\mathtt{inr} \circ \mathtt{out}_{L_A} \circ z, \mathtt{inr}\,] \circ L_A\,\mathtt{inl} \circ \mathtt{out}_{L_A} \circ c(y) \\
= & \quad \text{– (*) –} \\
 & [\,\mathtt{out}_{L_A} \circ z, \mathtt{inr}\,] \circ L_A\,(c(z)) \circ \mathtt{out}_{L_A} \circ c(y) \\
= & \quad \text{– def. of } c \text{, apo-}\textsc{Cancel} \text{ –} \\
 & [\,\mathtt{out}_{L_A} \circ z, \mathtt{inr}\,] \circ L_A\,(c(z)) \circ L_A[\,c(y), \mathtt{id}_{List'_A}\,] \circ \\
 & \qquad\qquad \circ\, [\,L_A\,\mathtt{inr} \circ \mathtt{out}_{L_A} \circ y, \mathtt{inr}\,] \circ L_A\,\mathtt{inl} \circ \mathtt{out}_{L_A} \\
= & \quad \text{– (*) –} \\
 & [\,\mathtt{out}_{L_A} \circ z, \mathtt{inr}\,] \circ L_A\,(c(z)) \circ [\,\mathtt{out}_{L_A} \circ y, \mathtt{inr}\,] \circ L_A\,(c(y)) \circ \mathtt{out}_{L_A} \\
= & \quad \text{– case analysis, def. of } L_A \text{ –} \\
 & [\,\mathtt{out}_{L_A} \circ z, \mathtt{inr}\,] \circ [\,L_A\,(c(z)) \circ \mathtt{out}_{L_A} \circ y, \mathtt{inr}\,] \circ L_A\,(c(z)) \circ L_A\,(c(y)) \circ \mathtt{out}_{L_A} \\
= & \quad \text{– case analysis, } L_A \text{ functor –} \\
 & [\,[\,\mathtt{out}_{L_A} \circ z, \mathtt{inr}\,] \circ L_A\,(c(z)) \circ \mathtt{out}_{L_A} \circ y, \mathtt{inr}\,] \circ L_A\,(c(z) \circ c(y)) \circ \mathtt{out}_{L_A} \\
= & \quad \text{– (*) –} \\
 & [\,L_A\,[\,c(z), \mathtt{id}_{List'_A}\,] \circ [\,L_A\,\mathtt{inr} \circ \mathtt{out}_{L_A} \circ z, \mathtt{inr}\,] \circ L_A\,\mathtt{inl} \circ \mathtt{out}_{L_A} \circ y, \mathtt{inr}\,] \circ \\
 & \qquad\qquad\qquad\qquad \circ\, L_A\,(c(z) \circ c(y)) \circ \mathtt{out}_{L_A} \\
= & \quad \text{– apo-}\textsc{Cancel}\text{, def. of } c \text{ –} \\
 & [\,\mathtt{out}_{L_A} \circ c(z) \circ y, \mathtt{inr}\,] \circ L_A\,(c(z) \circ c(y)) \circ \mathtt{out}_{L_A} \\
= & \quad \text{– (*) –} \\
 & L_A\,[\,c(z) \circ c(y), \mathtt{id}_{List'_A}\,] \circ [\,L_A\,\mathtt{inr} \circ \mathtt{out}_{L_A} \circ c(z) \circ y, \mathtt{inr}\,] \circ L_A \mathtt{inl} \circ \mathtt{out}_{L_A}
\end{array} \\
 & c(c(z) \circ y) \circ x
\end{array}
$$

4 Conclusion and further work

We have described the notion of primitive corecursion for coinductive types that is dual to primitive recursion for inductive types. Both of them are generalizations of more basic operations — iteration and coiteration respectively. While the value of an iterative function for a given argument depends solely on the values for its immediate subparts, the value of a primitive recursive function may additionally depend on these immediate subparts directly. Dually, the argument of a coiterative function for a value may only determine the arguments for the immediate subparts of the value, whereas the argument of a primitive corecursive function may alternatively determine these immediate subparts directly. Primitive corecursion, being more permissive than coiteration, makes it easier to write programs and to prove properties about programs.

Codatatypes and related notions such as coinduction and bisimulation are used widely in the analysis of processes specifiable by transition systems or state machines [JR97]. If processes are understood as functions from states to behaviors, sequential composition of two processes becomes a natural example of a function elegantly definable as an apomorphism. This leads us to believe that apomorphisms may turn out a viable construction in the modelling of processes. To check out this conjecture is one possible direction for continuing the work reported here.

In [UV97], we studied programming with (co)inductive types in the setting of intuitionistic natural deduction. Besides simple (co)iteration and primitive (co)recursion, we also considered course-of-value (co)iteration. A categorical treatment of these schemes is another subject for further work.

Acknowledgement

The authors are grateful to their anonymous referee for a very constructive and helpful review.

The work reported here was partially supported by the Estonian Science Foundation under grant no. 2976. The diagrams were produced using the Xy-pic macro package by Kristoffer H.

Rose.

References

[Bir87] Richard Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of NATO ASI Series F, pages 5–42. Springer-Verlag, Berlin, 1987.

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Dept. of Informatics, University of Twente, 1992.

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

[Gru96] Jim Grundy. A browsable format for proof presentation. *Mathesis Universalis*, 2, 1996.
URL `http://saxon.pip.com.pl/MathUniversalis/2/`.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis CST-47-87, Laboratory of Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh, Sept. 1987.

[How96] Brian T. Howard. Inductive, coinductive and pointed types. In *Proceedings of the 1st ACM SIGPLAN Intl. Conf. on Functional Programming, ICFP'96, Philadelphia, PA, USA, 24–26 May 1996*, pages 102–109. ACM Press, New York, 1996.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, 62:222–259, 1997.

[Lam68] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

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

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

[UV97] Tarmo Uustalu and Varmo Vene. A cube of proof systems for the intuitionistic predicate μ,ν-Logic. In M. Haveraaen, O. Owe, editors, *Selected Papers from the 8th Nordic Workshop on Programming Theory*, NWPT'96, Oslo, Norway, 4–6 Dec 1996, Research Report 248, Dept. of Informatics, Univ. of Oslo, pages 237–246, May 1997.