DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
23

CATEGORICAL PROGRAMMING WITH INDUCTIVE AND COINDUCTIVE TYPES

VARMO VENE

TARTU 2000

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
23

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS
23

CATEGORICAL PROGRAMMING WITH INDUCTIVE AND COINDUCTIVE TYPES

VARMO VENE

TARTU 2000

Faculty of Mathematics, University of Tartu, Estonia

Dissertation accepted for public defense of the degree of Doctor of Philosophy (PhD) on May 26, 2000 by the Council of the Faculty of Mathematics, University of Tartu.

Opponent:

PhD, University Lecturer	Jeremy Gibbons Oxford University Computing Laboratory Oxford, England

The public defense will take place on Sept. 3, 2000.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu.

Tartu Ülikooli Kirjastuse trükikoda
Tiigi 78, 50410 Tartu
Tellimus nr. 365

CONTENTS

LIST OF ORIGINAL PUBLICATIONS

1. Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings Estonian Academy of Sciences: Phys., Maths.*, 47(3):147–161. Sept. 1998.

2. Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *INFORMATICA*, 10(1):5–26, March 1999.

3. Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361. Autumn 1999.

4. Tarmo Uustalu and Varmo Vene. Coding recursion a la Mendler (extended abstract). In J. Jeuring, ed., *Proceedings 2nd Workshop on Generic Programming, WGP'2000, Ponte de Lima, Portugal, 6 July 2000*, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ., pp. 69–85. June 2000.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Data types are one of the key components of every program. They allow to organize values according to their purpose and properties. Already the very first programming languages had some concept of data types, containing at least a fixed collection of base types, like integers, reals, characters, but also means to form compound data types like records, arrays or lists. Soon it was realized (e.g. by Hoare [Hoa72]) that the structure of a program is intimately related with to the data structures it uses. Hence the ability to express and manipulate complex data structures in a flexible and intuitive way is an important measure of the usability and expressiveness of a programming language. Especially notable in this respect are modern functional languages like Haskell [PJH99] and ML [MTHM97] which possess rich type systems supporting algebraic data types, polymorphism, static type checking, etc.

In this thesis we explore two particular kinds of data types, inductive and coinductive types, and several programming constructs related to them. The characteristic property of inductive types (like natural numbers or lists) is that they provide very simple means for *construction* of data structures, but in order to use these values one often needs recursion. Coinductive types (like streams, possibly infinite lists) are *dual* to inductive ones. They come together with basic operations to *destruct* the values, however, their construction often involves recursion. General recursion can be quite difficult to reason about, and it is sometimes called the `goto` of functional programming.

In this thesis we use a categorical theory of initial algebras and terminal coalgebras as the abstract framework for inductive and coinductive types. This approach is attractive, as it equips (co)inductive types with generic (co)iteration operations. As these operations capture a very simple form of recursion, namely the structural (co)recursion, they are very easy to reason about. While the class of functions expressible easily in terms of (co)iteration is quite large, not all useful functions fall under it. The main objective of this thesis is to find new (co)recursive operations which capture some useful programming constructs, but still possess nice reasoning properties.

Algebraic data types in Haskell

Algebraic data types as provided by Haskell are intuitive yet powerful way to describe data structures. Essentially, new data types are defined by listing all possible canonical ways to construct its values. For instance, the following declaration in Haskell defines a new data type `Shape` together with two *data constructors*:

```
data Shape = Circle Float | Rectangle Float Float
```

Functions can manipulate such data types using *pattern matching* to "destruct" the data structure into its components:

```
perimeter :: Shape -> Float
perimeter (Circle r)      = 2 * pi * r
perimeter (Rectangle h w) = 2 * (h + w)
```

Data definitions can be *recursive* allowing to describe data structures of varying size. For instance, below are defined natural numbers and (polymorphic) lists as recursive data types:

```
data Nat    = Zero | Succ Nat
data List a = Nil  | Cons a (List a)
```

Functions which operate on recursive data types are often recursive too. For instance, below is defined a function which finds the sum of the elements in the argument list (here we use the standard notation for lists in Haskell, where `[]` denotes the empty list, i.e. the `Nil` constructor above, and `(:)` corresponds to the `Cons` constructor):

```
sum        :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

The definition can be read as follows: the sum of the empty list is 0; in the case of non-empty list, the sum of the whole list is obtained by adding the head of the list to the sum of the tail of the list.

Folds

The same recursion pattern, occurs so often when defining list processing functions, that Haskell provides a standard higher-order function which captures its essence:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f b []     = b
foldr f b (x:xs) = f x (foldr f b xs)
```

For instance, the `sum` function can be defined using `foldr` as follows:

```
sum = foldr (+) 0
```

Below are some other useful functions defined as instances of `foldr`:

```
length   = foldr (\x n -> 1+n) 0
xs ++ ys = foldr (:) ys xs
map f    = foldr (\x xs -> f x : xs) []
```

The first function computes the the length of the argument list, the second concatenates two lists, and finally, the third maps the given function to the every element of the argument list.

The function `foldr` has a very nice algebraic reading: it "replaces" the binary list constructor `(:)` by a binary function `f`, and the empty list `[]` by a constant `b`; i.e. `foldr f b` is a homomorphism between the algebras formed by list constructors and by `f` and `b`. This observation leads naturally to the generalization of `foldr` to other algebraic data types, and forms the basis of the categorical treatment of the inductive data types. For instance, the function which "replaces" constructors of natural numbers can be defined as follows:

```
foldNat :: (a -> a) -> a -> Nat -> a
foldNat f b Zero     = b
foldNat f b (Succ n) = f (foldNat f b n)
```

Calculating with folds

The function `foldr` satisfies two important laws. The first law, known as *identity law*, is rather obvious. It states that "replacing" the constructor functions with themselves gives the identity function:

$$\texttt{foldr}\,(:)\,[\,] = \texttt{id}$$

The second law, known as *fusion law*, gives conditions under which intermediate values produced by folding can be eliminated:

$$h\,(f\,a\,b) = g\,a\,(h\,b) \quad \Rightarrow \quad h \circ \texttt{foldr}\,f\,b = \texttt{foldr}\,g\,(h\,b)$$

To illustrate the use of these laws (and also the structured calculational proof style [Gru96] we use throughout the thesis) we give a proof of the fact, that map is a functor. First, the proof that map preserves identities:

$$\begin{array}{ll} & \mathtt{map}\,\mathtt{id} \\ = & \quad \text{– definition of } \mathtt{map} \text{ –} \\ & \mathtt{foldr}(\lambda x\, xs \to \mathtt{id}\, x : xs)\, [\,] \\ = & \quad \text{– definition of } \mathtt{id} \text{ –} \\ & \mathtt{foldr}(:)\, [\,] \\ = & \quad \text{– identity law –} \\ & \mathtt{id} \end{array}$$

Next, we use fusion to show that map preserves compositions:

$$\begin{array}{ll} & \mathtt{map}\, f \circ \mathtt{map}\, g \\ = & \quad \text{– definition of } \mathtt{map} \text{ –} \\ & \mathtt{map}\, f \circ \mathtt{foldr}(\lambda x\, xs \to g\, x : xs)\, [\,] \\ = & \quad \text{– fusion law –} \\ & \left[\begin{array}{ll} & \mathtt{map}\, f\, (g\, a : [\,]) \\ = & \quad \text{– definition of } \mathtt{map} \text{ –} \\ & \mathtt{foldr}(\lambda x\, xs \to f\, x : xs)\, [\,]\, (g\, a : [\,]) \\ = & \quad \text{– definition of } \mathtt{foldr} \text{ –} \\ & f(g\, a) : \mathtt{foldr}(\lambda x\, xs \to f\, x : xs)\, [\,]\, [\,] \\ = & \quad \text{– definition of } \mathtt{map} \text{ –} \\ & f(g\, a) : \mathtt{map}\, f\, [\,] \end{array}\right. \\ & \mathtt{foldr}(\lambda x\, xs \to f(g\, x) : xs)\, [\,] \\ = & \quad \text{– definition of } \mathtt{map} \text{ –} \\ & \mathtt{map}\, (f \circ g) \end{array}$$

The identity and fusion law for foldr can be proved by induction over lists. However, in categorical treatment of the inductive data types as initial algebras, these laws are simple corollaries of the initiality. Hence they are not specific to foldr and folds for any inductive data type satisfy similar laws.

1.2 Overview of the thesis

In this thesis we develop new recursion combinators that capture more complex recursion patterns than simple (co)iteration but still possess nice reasoning properties. In particular, we consider combinators for primitive (co)recursion and course-of-value (co)iteration.

It is well known that the primitive recursion can be simulated by a simple iteration which computes a value paired together with the argument, and that this construction leads to the notion of paramorphism which captures the primitive recursion directly. We will show, that the obvious dualization of this construction leads to notion of apomorphism which captures the recursion pattern known as primitive corecursion. More importantly, we will also show that a more involved generic simulation of memoization by iteration leads to the notion of histomorphism, a direct formalization of course-of-value iteration, and describe the dual notion of futumorphism, a formalization of course-of-value coiteration.

Inspired by type-theoretic work by N. P. Mendler [Men87, Men91], we will introduce the concepts of Mendler-style algebra and homomorphism and treat inductive types as initial Mendler-style algebras. From that basis, we will introduce Mendler-style analogs for the cata, para and histo combinators. From the theory developed, it appears that Mender-style recursion combinators are just as well-suited for program calculation as the conventional ones, but support a programming style more close to customary (general-)recursive programming.

The remainder of the thesis is organized as follows: Chapter 2 reviews the conventional treatment of inductive and coinductive types as initial algebras and terminal coalgebras of a functor. The calculational properties of basic iteration and coiteration are studied.

Chapter 3 studies the properties of operators corresponding to primitive recursion and corecursion. This is the first chapter which contains our original contribution. Namely, we formalize primitive corecursive functions as apomorphisms, and show their utility on several simple examples (the "standard" example being the concatenation of two colists).

The next three chapters contain our main contribution to the theory of categorical data types.

Chapter 4 is devoted to course-of-value iteration and coiteration. They are formalized respectively as histo- and futumorphisms, the latter being functions which generate several elements of codata type at once.

Chapter 5 presents an alternative treatment of inductive types as initial Mendler-style algebras. It shows that, in the case of covariant functor, the conventional treatment coincides with the Mendler-style one. However, Mendler-style inductive types can be defined also for mixed variant base functor. In this case, it is shown, that if certain restricted existential types are available, then Mendler-style inductive types are equivalent with the conventional ones, but for a different (covariant) functor.

Chapter 6 uses Mendler-style algebras to define recursion operators which operate on conventional inductive types. Mendler-style versions of cata-, para- and histomorphisms are formalized and their properties are studied.

The concluding chapter 7 outlines possible future work.

1.3 Notation

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

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

CHAPTER 2

INDUCTIVE AND COINDUCTIVE TYPES

In this chapter we review the traditional treatment of inductive and coinductive types as initial algebras and terminal coalgebras of a functor.

2.1 Initial algebras and catamorphisms

Definition 2.1 (algebra)
Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor on category $\mathcal{C}$. An F-*algebra* is a pair (C, φ), where C is an object and $\varphi : \mathsf{F}C \to C$ an arrow in the category $\mathcal{C}$. The object C is the *carrier* and the functor F is the signature of the algebra. □

Definition 2.2 (algebra homomorphism)
Let (C, φ) and (D, ψ) be two F-algebras. A *homomorphism* from (C, φ) to (D, ψ) is an arrow $f : C \to D$ in the category $\mathcal{C}$, such that

$$f \circ \varphi \;=\; \psi \circ \mathsf{F}\, f$$

i.e. makes the following diagram to commute:

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

□

For any F-algebra, the identity arrow on its carrier is a homomorphism from it to itself and also the composition of two homomorphisms is always a homomorphism, so we can define a category where objects are F-algebras and arrows are homomorphisms between them. However we have to be a little careful, as the same arrow from the base category can be homomorphism between more than one pair of F-algebras. For instance, the identity arrow id_C is a homomorphism from any F-algebra with carrier C to itself.

Definition 2.3 (category of algebras)
The *category of* F*-algebras over* $\mathcal{C}$ — $\mathcal{A}lg(\mathsf{F})$ — is defined by:

- Objects: F-algebras; i.e. arrows φ of $\mathcal{C}$ such that $\mathrm{dom}\,\varphi = \mathsf{F}(\mathrm{cod}\,\varphi)$.
- Arrows: triples $(f, \varphi, \psi) : \varphi \to \psi$ where φ and ψ are F-algebras and $f : \mathrm{cod}\,\varphi \to \mathrm{cod}\,\psi$ is a homomorphism from φ to ψ.
- Identity: $\mathrm{id}_\varphi = (\mathrm{id}_{\mathrm{cod}\,\varphi}, \varphi, \varphi)$.
- Composition: $(f, \varphi_2, \varphi_3) \circ (g, \varphi_1, \varphi_2) = (f \circ g, \varphi_1, \varphi_3)$.

□

Definition 2.4 (initial algebra)
A F-algebra $(\mu\mathsf{F}, \mathrm{in})$ is the *initial* F*-algebra* if for any F-algebra (C, φ) there exists a unique arrow $(\!|\varphi|\!) : \mu\mathsf{F} \to C$ making the following diagram commute:

$$\begin{tikzcd} \mathsf{F}\,\mu\mathsf{F} \arrow[r, "\mathrm{in}"] \arrow[d, "\mathsf{F}\,(\!|\varphi|\!)"'] & \mu\mathsf{F} \arrow[d, "(\!|\varphi|\!)"] \\ \mathsf{F}\,C \arrow[r, "\varphi"] & C \end{tikzcd}$$

i.e. satisfying the universal property:

$$f \circ \mathrm{in} = \varphi \circ \mathsf{F}\,f \quad \equiv \quad f = (\!|\varphi|\!) \qquad \text{cata-CHARN}$$

The arrows in form $(\!|\varphi|\!)$ are called *catamorphisms* (derived from the Greek preposition $\kappa\alpha\tau\alpha$ meaning ‘downwards’). □

In other words, the initial algebra $(\mu\mathsf{F}, \mathrm{in})$ is an initial object in the category $\mathcal{A}lg(\mathsf{F})$, and the catamorphism $(\!|\varphi|\!)$ is the mediating arrow out of it.

The initial F-algebra may or may not exist. It is guaranteed to exist if F is ω-cocontinuous (i.e. it preserves the colimits of ω-chains). All polynomial functors (i.e. functors built up from products, sums, the identity functor, and constant functors) are ω-cocontinuous and, hence, the initial algebras for them exist.

Corollary 2.1 *Let* $(\mu\mathsf{F}, \mathrm{in})$ *be an initial* F*-algebra.*

- ***Cancellation:*** *For any* F*-algebra* $\varphi : \mathsf{F}C \to C$

$$(\!|\,\varphi\,|\!) \circ \mathrm{in} = \varphi \circ \mathsf{F}\,(\!|\,\varphi\,|\!) \qquad \text{cata-}\textsc{Self}$$

- ***Reflection:***

$$\mathrm{id} = (\!|\,\mathrm{in}\,|\!) \qquad \text{cata-}\textsc{Refl}$$

- ***Fusion:*** *For any* F*-algebras* $\varphi : \mathsf{F}C \to C$*,* $\psi : \mathsf{F}D \to D$ *and an arrow* $f : C \to D$

$$f \circ \varphi = \psi \circ \mathsf{F}\,f \quad \Rightarrow \quad f \circ (\!|\,\varphi\,|\!) = (\!|\,\psi\,|\!) \qquad \text{cata-}\textsc{Fusion}$$

Intuitively, the initial algebra $\mathrm{in} : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ denotes the collection of constructor functions for inductive data type $\mu\mathsf{F}$, and the catamorphism is a simple iteration. When read from left to right, the cancellation law can be viewed as the reduction rule for terms where catamorphism is applied to a data constructor. The reduction proceeds recursively by systematically replacing data constructors with some algebra with same signature. If constructors are replaced by themselves nothing is changed. This is exactly what the reflection law claims.

The formal justification on the identification of inductive types with initial algebras is given by the following fundamental theorem, known as Lambek lemma. Its proof, albeit simple, provides a nice example of using above mentioned laws in action.

Theorem 2.2 (Lambek [Lam68]) *The initial algebra* $\mathrm{in}_{\mathsf{F}} : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ *is an isomorphism with the inverse defined as*

$$\mathrm{in}^{-1} \;=\; (\!|\,\mathsf{F}\ \mathrm{in}\,|\!) \qquad \text{in-inv-}\textsc{Def}$$

Proof. Note that in^{-1} has indeed the right typing; i.e. $\mathrm{in}^{-1} : \mu\mathsf{F} \to \mathsf{F}\mu\mathsf{F}$. We have to show that it is the pre- and post-inverse of the in. For the first we argue:

$$\begin{array}{ll} & \mathrm{in} \circ (\!|\,\mathsf{F}\ \mathrm{in}\,|\!) \\ = & \qquad \text{– cata-}\textsc{Fusion}\text{ –} \\ & (\!|\,\mathrm{in}\,|\!) \\ = & \qquad \text{– cata-}\textsc{Refl}\text{ –} \\ & \mathrm{id} \end{array}$$

To show that it is also the post-inverse, we make use of the just shown fact (in the step marked "see above"):

$$
\begin{array}{cl}
 & (\!|\, \mathsf{F}\ \mathrm{in} \,|\!) \circ \mathrm{in} \\
= & \quad \text{– cata-}\textsc{Self}\text{ –} \\
 & \mathsf{F}\ \mathrm{in} \circ \mathsf{F}\, (\!|\, \mathsf{F}\ \mathrm{in} \,|\!)) \\
= & \quad \text{– } \mathsf{F} \text{ functor –} \\
 & \mathsf{F}\, (\mathrm{in} \circ (\!|\, \mathsf{F}\ \mathrm{in} \,|\!)) \\
= & \quad \text{– see above –} \\
 & \mathsf{F}\ \mathrm{id} \\
= & \quad \text{– } \mathsf{F} \text{ functor –} \\
 & \mathrm{id}
\end{array}
$$

□

The theorem shows that the carrier of the initial algebra is (up to isomorphism) a fixed point of the functor. In fact, initial algebras generalize the notion of the least fixed point from lattice theory in the sense that if the base category is a preorder and thus an endofunctor is a monotonic function then the carrier of the initial algebra is the least fixed point of the given function.

Example 2.1 (empty type)
In the category $\mathcal{S}et$ of sets and functions, the pair $(\emptyset, \mathrm{id}_\emptyset)$ is the initial algebra of the identity functor with the unique arrow out of $\emptyset$ as the required unique homomorphism. More generally, in any category with initial object, the pair $(0, \mathrm{id}_0)$ is the initial Id-algebra. □

Example 2.2 (naturals)
Consider the set $Nat = \{0, 1, 2, \dots\}$ of natural numbers with its zero and successor function $zero : 1 \to Nat$ and $succ : Nat \to Nat$ defined by:

$$
\begin{array}{lcl}
zero\ () & = & 0 \\
succ\ n & = & n + 1.
\end{array}
$$

Using join, these functions combine into a single function $[\, zero, succ \,] : 1 + Nat \to Nat$, forming an algebra of the functor $\mathsf{N}(X) = 1 + X$. In fact, the pair $(Nat, [\, zero, succ \,])$ is the initial N-algebra; i.e. $\mu\mathsf{N} = Nat$ and $\mathrm{in} = [\, zero, succ \,]$. To show this, assume an arbitrary N-algebra (C, φ). We have to find a function $f : Nat \to C$ which is homomorphism and it should be unique. Because of every arrow out of sum is join, $\varphi = [\, c, h \,]$ for some constant $c : 1 \to C$ and arrow $h : C \to C$. So, the homomorphism condition for N-algebras states that f should make the following diagram commute

$$\begin{tikzcd}
1 \arrow[r, "zero"] \arrow[dr, "c"'] & Nat \arrow[d, "f"] & Nat \arrow[l, "succ"'] \arrow[d, "f"] \\
 & C & C \arrow[l, "h"]
\end{tikzcd}$$

i.e. satisfies two equations

$$\begin{array}{rcl} f \circ zero & = & c \\ f \circ succ & = & h \circ f. \end{array}$$

But this equation system has exactly one solution, namely the function defined by $n \mapsto h^n(c())$, which gives to us the required unique homomorphism.

For instance, the sum and product of two naturals can be defined as follows:

$$\begin{array}{rcl} add(n,m) & = & \llparenthesis\, [\, \lambda x.m, succ\,]\, \rrparenthesis(n) \\ mul(n,m) & = & \llparenthesis\, [\, \lambda x.m, \lambda x.add(m,x)\,]\, \rrparenthesis(n). \end{array}$$

The predecessor function $pred : Nat \to 1 + Nat$ which maps $0 \mapsto \mathrm{inl}()$ and $n+1 \mapsto \mathrm{inr}\ n$ can be defined by

$$pred = \llparenthesis\, \mathrm{id} + [\, zero, succ\,]\, \rrparenthesis$$

i.e. it is the inverse of the initial N-algebra. □

Parametric data types can be easily modeled by initial algebras using bifunctors as their signatures. Let $\mathsf{F} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ be a bifunctor, then for any object A we have an endofunctor $\mathsf{F}_A : \mathcal{C} \to \mathcal{C}$ defined as $\mathsf{F}_A(X) = \mathsf{F}(A, X)$.

Example 2.3 (lists)
The data type of lists over a given set A can be represented as the initial algebra $(\mu\mathsf{L}_A, \mathrm{in})$ of the functor L_A defined by $\mathsf{L}_A(X) = 1 + (A \times X)$. Denote $\mu\mathsf{L}_A$ by $List(A)$. The constructor functions $nil : 1 \to List(A)$ and $cons : A \times List(A) \to List(A)$ are defined by

$$\begin{array}{rcl} nil & = & \mathrm{in} \circ \mathrm{inl} \\ cons & = & \mathrm{in} \circ \mathrm{inr}, \end{array}$$

so $\mathrm{in} = [\, nil, cons\,]$. Given any two functions $c : 1 \to C$ and $h : A \times C \to C$, the catamorphism $f = \llparenthesis\, [\, c, h\,]\, \rrparenthesis : List(A) \to C$ is the unique solution of the equation system

$$\begin{array}{rcl} f \circ nil & = & c \\ f \circ cons & = & h \circ (\mathrm{id} \times f), \end{array}$$

i.e., $\mathit{foldr}(c,h)$ from functional programming. For instance, the function $\mathit{length} : \mathit{List}(A) \to \mathit{Nat}$ which finds the length of the list, can be defined as catamorphism

$$\mathit{length} \;=\; (\!|\, [\, \mathit{zero}, \lambda(a,n).\mathit{succ}(n)\,]\, |\!).$$

As another example, the function $\mathit{concat} : \mathit{List}(A) \times \mathit{List}(A) \to \mathit{List}(A)$, which concatenates two lists, can be defined as catamorphism

$$\mathit{concat}(xs, ys) \;=\; (\!|\, [\, \lambda x.ys, \mathit{cons}\,]\, |\!)(xs).$$

Finally, the function $\mathit{map}(f) : \mathit{List}(A) \to \mathit{List}_B$, which applies the function $f : A \to B$ to every element of the argument list, can be defined as follows

$$\mathit{map}(f) \;=\; (\!|\, [\, \mathit{nil}, \mathit{cons} \circ (f \times \mathrm{id})\,]\, |\!).$$

Lots of other examples about list catamorphisms (i.e. function *foldr*) can be found in any functional programming textbook (e.g. [Bir98]). □

Example 2.4 (binary trees)
Consider the bifunctor $\mathsf{B}(A, X) = A + X \times X$. The initial B_A-algebra defines the data type of binary (leaf) trees $\mathit{Btree}(A) = \mu \mathsf{B}_A$ with a constructor functions

$$\begin{array}{lll} \mathit{leaf} & = \mathrm{in} \circ \mathrm{inl} & : A \to \mathit{Btree}(A) \\ \mathit{branch} & = \mathrm{in} \circ \mathrm{inr} & : \mathit{Btree}(A) \times \mathit{Btree}(A) \to \mathit{Btree}(A) \end{array}$$

For instance, a binary tree of naturals with three leafs can be constructed as

$$\mathit{branch}(\mathit{branch}(\mathit{leaf}(1), \mathit{leaf}(2)), \mathit{leaf}(3)).$$

Given any functions $l : A \to C$ and $b : C \times C \to C$, the catamorphism $f = (\!|\, [\, l, b\,]\, |\!) : \mathit{Btree}(A) \to C$ is the unique solution of the equation system

$$\begin{array}{lcl} f \circ \mathit{leaf} & = & l \\ f \circ \mathit{branch} & = & b \circ (f \times f). \end{array}$$

For instance, the function $\mathit{flatten} : \mathit{Btree}(A) \to \mathit{List}(A)$, which collects elements in leaves into list in left-to-right order, can be defined as

$$\mathit{flatten} \;=\; (\!|\, [\, \mathit{unit}, \mathit{concat}\,]\, |\!),$$

where $\mathit{unit}(x) = \mathit{cons}(x, \mathit{nil}) : A \to \mathit{List}(A)$ is a function which converts an element into singleton list, and concat the list concatenation function from Example 2.3. □

It is well known from functional programming that the type constructor $List$ together with the function map form a functor. The next theorem shows that lists are not exceptional in this respect and every similarly defined parametric data type can be extended to a functor.

Theorem 2.3 *Let* $\mathsf{F} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ *be a bifunctor, such that and for any object* A *there exists initial* F_A*-algebra* $(\mu\mathsf{F}_A, \mathrm{in})$*. Then, the mapping* $\mathsf{T}(A) = \mu\mathsf{F}_A$ *can extended to the endofunctor on* $\mathcal{C}$ *by defining*

$$\mathsf{T}(f) \;=\; (\!|\, \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \,|\!) \qquad\qquad \text{data-map-DEF}$$

The functor $\mathsf{T} : \mathcal{C} \to \mathcal{C}$ *is called a* data functor *of* F.

Proof. Note that definition above has the right typing. We have to show that T preserves identities and composition. First, identities:

$$\begin{array}{ll}
 & \mathsf{T}\,\mathrm{id} \\
= & \quad \text{– data-map-DEF –} \\
 & (\!|\, \mathrm{in} \circ \mathsf{F}(\mathrm{id}, \mathrm{id}) \,|\!) \\
= & \quad \text{– F bifunctor –} \\
 & (\!|\, \mathrm{in} \,|\!) \\
= & \quad \text{– cata-REFL –} \\
 & \mathrm{id}
\end{array}$$

For the composition, we show that $\mathsf{T}(f)$ is a homomorphism from $\mathrm{in} \circ \mathsf{F}(g, \mathrm{id})$ to $\mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id})$ and then use cata-FUSION:

$$\begin{array}{ll}
 & \mathsf{T}f \circ \mathsf{T}g \\
= & \quad \text{– data-map-DEF –} \\
 & \mathsf{T}f \circ (\!|\, \mathrm{in} \circ \mathsf{F}(g, \mathrm{id}) \,|\!) \\
= & \quad \text{– cata-FUSION –} \\
 & \left[\begin{array}{ll}
 & \mathsf{T}f \circ \mathrm{in} \circ \mathsf{F}(g, \mathrm{id}) \\
= & \quad \text{– data-map-DEF –} \\
 & (\!|\, \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \,|\!) \circ \mathrm{in} \circ \mathsf{F}(g, \mathrm{id}) \\
= & \quad \text{– cata-SELF –} \\
 & \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \circ \mathsf{F}(\mathrm{id}, (\!|\, \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \,|\!)) \circ \mathsf{F}(g, \mathrm{id}) \\
= & \quad \text{– F bifunctor –} \\
 & \mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id}) \circ \mathsf{F}(\mathrm{id}, (\!|\, \mathrm{in} \circ \mathsf{F}(f, \mathrm{id}) \,|\!)) \\
= & \quad \text{– data-map-DEF –} \\
 & \mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id}) \circ \mathsf{F}(\mathrm{id}, \mathsf{T}f)
\end{array}\right. \\
 & (\!|\, \mathrm{in} \circ \mathsf{F}(f \circ g, \mathrm{id}) \,|\!) \\
= & \quad \text{– data-map-DEF –} \\
 & \mathsf{T}(f \circ g)
\end{array}$$

□

Example 2.5 (bushes)
Consider the bifunctor $\mathsf{B}(A, X) = A \times List(\mathsf{B}(A, X))$. The initial B_A-algebra defines the data type of bushes (finitely branching trees) $Bush(A) = \mu\mathsf{B}_A$ with a constructor function $node = \mathrm{in} : A \times List(Bush(A)) \to Bush(A)$. Given any function $h : A \times List(C) \to C$, the catamorphism $f = (\!|\, h \,|\!) : Bush(A) \to C$ is the unique solution of the equation

$$f \circ node \;=\; h \circ (\mathrm{id} \times map(f)),$$

where $map(f) : List(Bush(A)) \to List(C)$ is the map function on lists defined in Example 2.3. □

2.2 Terminal coalgebras and anamorphisms

We now dualize the material about initial algebras and catamorphisms.

Definition 2.5 (coalgebra)
Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor on category $\mathcal{C}$. A F-*coalgebra* is a pair (C, φ), where C is an object and $\varphi : C \to \mathsf{F}C$ an arrow in the category $\mathcal{C}$. The object C is the *carrier* and the functor F is the signature of the coalgebra. □

Definition 2.6 (coalgebra homomorphism)
Let (C, φ) and (D, ψ) be two F-coalgebras. A *homomorphism* from (C, φ) to (D, ψ) is an arrow $f : C \to D$ in the category $\mathcal{C}$, such that

$$\psi \circ f \;=\; \mathsf{F}\,f \circ \varphi$$

i.e. makes the following diagram to commute:

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

□

Similarly to homomorphisms between algebras, homomorphisms between coalgebras compose with identity arrow as the identity homomorphism.

Definition 2.7 (category of coalgebras)
The *category of* F*-coalgebras over* $\mathcal{C}$ — $\mathcal{C}oAlg(\mathsf{F})$ — is defined by:

- Objects: F-coalgebras; i.e. arrows φ of $\mathcal{C}$ such that $\mathrm{cod}\,\varphi = \mathsf{F}(\mathrm{dom}\,\varphi)$.

- Arrows: triples $(f, \varphi, \psi) : \varphi \to \psi$ where φ and ψ are F-coalgebras and $f : \mathrm{dom}\,\varphi \to \mathrm{dom}\,\psi$ is a homomorphism from φ to ψ.
- Identity: $\mathrm{id}_\varphi = (\mathrm{id}_{\mathrm{dom}\,\varphi}, \varphi, \varphi)$.
- Composition: $(f, \varphi_2, \varphi_3) \circ (g, \varphi_1, \varphi_2) = (f \circ g, \varphi_1, \varphi_3)$.

□

Definition 2.8 (terminal coalgebra)
A F-coalgebra $(\nu\mathsf{F}, \mathrm{out})$ is the *terminal* F*-coalgebra* if for any F-coalgebra (C, φ) there exists unique arrow $\llbracket\, \varphi \,\rrbracket : C \to \nu\mathsf{F}$ making the following diagram commute:

$$\begin{tikzcd} C \arrow[r, "\varphi"] \arrow[d, "\llbracket \varphi \rrbracket"'] & \mathsf{F}\, C \arrow[d, "\mathsf{F}\llbracket \varphi \rrbracket"] \\ \nu\mathsf{F} \arrow[r, "\mathrm{out}"] & \mathsf{F}\,\nu\mathsf{F} \end{tikzcd}$$

i.e. satisfying the universal property:

$$\mathrm{out} \circ f = \mathsf{F}\, f \circ \varphi \quad \equiv \quad f = \llbracket\, \varphi \,\rrbracket \qquad \text{ana-CHARN}$$

The arrows in form $\llbracket\, \varphi \,\rrbracket$ are called *anamorphisms* (derived from the Greek preposition $\alpha\nu\alpha$ meaning 'upwards'; the name is due to Meijer). □

In other words, the terminal coalgebra $(\nu\mathsf{F}, \mathrm{out})$ is the terminal object in the category $\mathcal{C}o\mathcal{A}lg(\mathsf{F})$, and the anamorphism $\llbracket\, \varphi \,\rrbracket$ is the mediating arrow out of it.

Corollary 2.4 *Let* $(\nu\mathsf{F}, \mathrm{out})$ *be a terminal* F*-coalgebra.*

- ***Cancellation:*** *For any* F*-coalgebra* $\varphi : C \to \mathsf{F}C$

$$\mathrm{out} \circ \llbracket\, \varphi \,\rrbracket = \mathsf{F}\,\llbracket\, \varphi \,\rrbracket \circ \varphi \qquad \text{ana-SELF}$$

- ***Reflection:***

$$\mathrm{id} = \llbracket\, \mathrm{out} \,\rrbracket \qquad \text{ana-REFL}$$

- ***Fusion:*** *For any* F*-coalgebras* $\varphi : C \to \mathsf{F}C$*,* $\psi : D \to \mathsf{F}D$ *and an arrow* $f : C \to D$

$$\psi \circ f = \mathsf{F}\, f \circ \varphi \quad \Rightarrow \quad \llbracket\, \psi \,\rrbracket \circ f = \llbracket\, \varphi \,\rrbracket \qquad \text{ana-FUSION}$$

Terminal coalgebras satisfy the dual version of the Lambek lemma stating that their carriers are fixed points of F.

Corollary 2.5 *The terminal coalgebra* $\mathrm{out} : \nu\mathsf{F} \to \mathsf{F}\,\nu\mathsf{F}$ *is an isomorphism with the inverse* $\mathrm{out}^{-1} : \mathsf{F}\nu\mathsf{F} \to \nu\mathsf{F}$ *defined as follows*

$$\mathrm{out}^{-1} \;=\; [\![\, \mathsf{F}\ \mathrm{out}\,]\!] \qquad\qquad \text{out-inv-DEF}$$

□

Dually to initial algebras, terminal coalgebras generalize the notion of the greatest fixed point, as the carrier of the terminal coalgebra for a monotonic endofunction over preorder is the the greatest fixed point of the given function.

Example 2.6 (unit type)
In the category $\mathcal{S}et$, the pair $(\{\star\}, \mathrm{id}_{\{\star\}})$ is the terminal coalgebra of the identity functor, where $\{\star\}$ is a one element set. The unique arrow into $\{\star\}$ is the required unique homomorphism. More generally, in any category with terminal object, the pair $(1, \mathrm{id}_1)$ is the terminal Id-coalgebra. □

Example 2.7 (conaturals)
Consider the endofunctor $\mathsf{N}(X) = 1 + X$ from Example 2.2. Recall that its initial algebra is given by the set $Nat = \{0, 1, 2, \dots\}$ of natural numbers together with the join of zero and successor function as algebra structure $[\, zero, succ\,] : 1 + Nat \to Nat$. The inverse of the initial algebra $pred : Nat \to 1 + Nat$ is a N-coalgebra, but it is not terminal.

The terminal N-coalgebra is given by the pair $(CoNat, pred)$, where $CoNat = \{0, 1, 2, \dots\} \cup \{\infty\}$ is the set of natural numbers augmented with an extra element ∞, and $pred : CoNat \to 1 + CoNat$ is the predecessor function

$$\begin{array}{lcl} pred\ 0 & = & \mathrm{inl}\,() \\ pred\ (n+1) & = & \mathrm{inr}\ n \\ pred\ \infty & = & \mathrm{inr}\ \infty. \end{array}$$

Given an arbitrary N-coalgebra (C, f), there exists a unique function $g = [\![\, f\,]\!] : C \to CoNat$ satisfying

$$pred(g(x)) \;=\; \begin{cases} \mathrm{inl}\,() & \texttt{if}\ f(x) = \mathrm{inl}\,() \\ \mathrm{inr}(g(y)) & \texttt{if}\ f(x) = \mathrm{inr}\ y \end{cases}$$

For instance, consider the function $f : CoNat \times CoNat \to 1 + (CoNat \times CoNat)$ defined by

$$f(x, y) \;=\; \begin{cases} \mathrm{inl}\,() & \texttt{if}\ pred(x) = pred(y) = \mathrm{inl}\,() \\ \mathrm{inr}(x', y) & \texttt{if}\ pred(x) = \mathrm{inr}\ x' \\ \mathrm{inr}(x, y') & \texttt{if}\ pred(x) = \mathrm{inl}\,(),\ pred(y) = \mathrm{inr}\ y', \end{cases}$$

i.e. an N-coalgebra with carrier $CoNat \times CoNat$. The anamorphism $add = [\![\, f \,]\!] : CoNat \times CoNat \to CoNat$ defines the addition function on conaturals.

Because the sum appears in the target, we cannot decompose an N-coalgebra into simpler components in general. Often, however, the N-coalgebra is in the form $f = (!_C + h) \circ p?$ for some predicate $p : C \to Bool$ and function $h : C \to C$. In this case, the homomorphism condition translates to

$$pred(g(x)) \;=\; \begin{cases} \mathrm{inl}\,() & \texttt{if}\; p(x) \\ \mathrm{inr}(g(h(x))) & \texttt{otherwise}. \end{cases}$$

□

Parametric coinductive types can be modeled by terminal coalgebras using bifunctors as their signatures. Also, the resulting type constructor can be extended to a functor.

Corollary 2.6 *Let $\mathsf{F} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ be a bifunctor, such that and for any object A there exists terminal F_A-coalgebra $(\nu\mathsf{F}_A, \mathrm{out})$. Then, the mapping $\mathsf{T}(A) = \nu\mathsf{F}_A$ can extended to the endofunctor on $\mathcal{C}$ by defining*

$$\mathsf{T}(f) \;=\; [\![\, \mathsf{F}(f, \mathrm{id}) \circ \mathrm{out} \,]\!] \qquad \text{codata-map-DEF}$$

The functor $\mathsf{T} : \mathcal{C} \to \mathcal{C}$ is called a codata functor *of* F. □

Example 2.8 (streams)
The codata type of streams over a given set A is nicely represented by the terminal coalgebra $(\nu\mathsf{S}_A, \mathrm{out})$ of the bifunctor $\mathsf{S}(A, X) = A \times X$. Write $Stream(A)$ for νS_A. The functions $head : Stream(A) \to A$ and $tail : Stream(A) \to Stream(A)$ equal $\mathrm{fst} \circ \mathrm{out}$ and $\mathrm{snd} \circ \mathrm{out}$, respectively. Given any two functions $c : C \to A$ and $h : C \to C$, the anamorphism $[\![\, \langle c, h \rangle \,]\!]$ is the unique solution $f : C \to Stream_A$ of the equation system

$$\begin{array}{lcl} head \circ f & = & c \\ tail \circ f & = & f \circ h. \end{array}$$

The function $nats : Nat \to Stream(Nat)$, which returns the stream of all natural numbers starting with the natural number given as the argument, is the unique solution of the equation system

$$\begin{array}{lcl} head \circ nats & = & \mathrm{id} \\ tail \circ nats & = & nats \circ succ, \end{array}$$

and is thus definable as the anamorphism $[\![\, \langle \mathrm{id}, succ \rangle \,]\!]$.

The function $zip : Stream(A) \times Stream(B) \to Stream(A \times B)$ that zips the argument streams together is characterized as follows:

$$
\begin{array}{lcl}
head \circ zip & = & (\mathrm{fst} \times \mathrm{fst}) \circ (\mathrm{out} \times \mathrm{out}) \\
tail \circ zip & = & zip \circ (\mathrm{snd} \times \mathrm{snd}) \circ (\mathrm{out} \times \mathrm{out})
\end{array}
$$

This function can, therefore, be defined as $[\![\langle \mathrm{fst} \times \mathrm{fst}, \mathrm{snd} \times \mathrm{snd} \rangle \circ (\mathrm{out} \times \mathrm{out})]\!]$.

The function $iterate(f) : A \to Stream(A)$ builds the stream of all repeated applications of function $f : A \to A$ to the argument

$$iterate(f) = [\![\langle \mathrm{id}, f \rangle]\!]$$

□

Example 2.9 (colists)
The codata type of colists over a given set A can be represented as the terminal coalgebra $(\nu L_A, \mathrm{out})$ of the functor L_A. Write $List'_A$ for νL_A. Given any function $g : C \to 1 + (A \times C)$, the anamorphism $[\![g]\!]$ is the unique solution $f : C \to List'_A$ of the equation $\mathrm{out} \circ f = (\mathrm{id} + (\mathrm{id} \times f)) \circ g$, i.e. the function $unfold(g)$ from functional programming. □

2.3 Implementation in Haskell

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

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

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

Given some type constructor, it can be declared to be a functor by defining the `fmap` function for it using instance declaration. For example, the `fmap` function for the list type constructor is defined in the Haskell Prelude as follows:

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

The definition tells, that the `fmap` for lists is "ordinary" map function.

Inductive types, being carriers of initial algebras, are least fixed points of the corresponding functors. In Haskell, this can be modeled by the following declaration:

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

Given a type constructor `f`, this defines a new type `Mu f` which has the same representation as the type `f (Mu f)`; i.e. it defines `Mu f` as the least fixed point of `f`. In addition, it defines a data constructor `In :: f (Mu f) -> Mu f` for the explicit one-way coercion between the types. The coercion in the other way (i.e. the inverse of `In`) can be defined by pattern matching:

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

Coinductive types are carriers of terminal coalgebras, thus greatest fixed points of the corresponding functors. Because Haskell allows to use a general recursion, coinductive types are necessarily isomorphic to the inductive types with the same base functor. Hence we could use `Mu f` also for coinductive types. However, in order to make intended meaning of different usages explicit, we define them separately:

```
> newtype Nu f = Wrap (f (Nu f))

> out :: Nu f -> f (Nu f)
> out (Wrap x) = x
```

In order to implement cata- and anamorphisms, we make use the corresponding cancellation laws, but in a slightly modified form[1]. Namely, we eliminate the occurrences of the initial algebra in or terminal coalgebra out from the left-hand side of the equation, by pre- or postcomposing both sides with the corresponding inverse.

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

> ana  :: Functor f => (c -> f c) -> c -> Nu f
> ana phi  = Wrap . fmap (ana phi)  . phi
```

[1] In fact, this is not necessary for catamorphisms, as we could use pattern matching to implement the cancellation law directly. However, this does not work in the case of anamorphisms, as Haskell requires that the name of the defined function has to be the outermost in the left-hand side of the defining equation.

The combinator `cata` takes a function of type `f c -> c` (i.e. algebra) into function of type `Mu f -> c`. Dually, `ana` takes a function of type `c -> f c` (i.e. coalgebra) into function of type `c -> Nu f`. In both cases, the type constructor `f` has to belong into class `Functor`. The restriction on `f` is necessary, as the right-hand sides of the defining equations makes use of the function `fmap`. Note that there was no such restriction in the definitions of `Mu` or `Nu`.

Example 2.10 (naturals)
The data type of natural numbers, as given in example 2.2, is an initial algebra for the functor $\mathsf{N}(X) = 1 + X$. In Haskell, this can be implemented as follows:

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

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

> type Nat = Mu N
```

The first line defines a new type constructor `N`, which corresponds to the object mapping part of the functor N. Then, the instance declaration defines the function `fmap` for it; i.e. makes it a functor. Finally, the last line defines data type `Nat` as the least fixed point of `N`.

The constructor functions for naturals (the constant zero and successor function) can be defined as follows:

```
> zeroN :: Nat
> zeroN  = In Z

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

Below are listed some illustrative values of type `Nat` (naturals 1, 2 and 4):

```
In (S (In Z))
In (S (In (S (In Z))))
In (S (In (S (In (S (In (S (In Z))))))))
```

The sum of two naturals can be implemented as following catamorphism:

```
> addN :: Nat -> Nat -> Nat
> addN x y = cata phi x
>     where phi Z     = y
>           phi (S n) = succN n
```

Note that the algebra `phi` is defined by the case analysis over the structure of type constructor `N`, specifying the result separately depending whether the inductive argument (i.e. `x`) is zero or not. In the case of non-zero inductive argument, the result is specified in terms of the value on its predecessor.

Analogously, the product of two naturals can be implemented by a catamorphism:

```
> mulN :: Nat -> Nat -> Nat
> mulN x y = cata phi x
>     where phi Z     = y
>           phi (S n) = addN y n
```

□

Example 2.11 (lists)
The data type of lists can be implemented as follows:

```
> data L a x = N | C a x

> instance Functor (L a) where
>    fmap f N        = N
>    fmap f (C x xs) = C x (f xs)

> type List a   = Mu (L a)

> nilL :: List a
> nilL  = In N

> consL     :: a -> List a -> List a
> consL x xs = In (C x xs)
```

The functions `nilL` and `consL` are constructor functions for lists. The first corresponds to an empty list, and the second to the "ordinary" list constructor.

The functions $length$, $concat$ and map from the example 2.3 can be implemented as follows:

```
> lengthL :: List a -> Nat
> lengthL = cata phi
>     where phi N       = zeroN
>           phi (C _ n) = succN n
```

```
> concatL :: List a -> List a -> List a
> concatL xs ys = cata phi xs
>     where phi N         = ys
>           phi (C x xs') = consL x xs'

> mapList :: (a -> b) -> List a -> List b
> mapList f = cata phi
>     where phi N        = nilL
>           phi (C a bs) = consL (f a) bs
```

□

Example 2.12 (streams)
The codata type of streams can be implemented as follows:

```
> data S a x =  St a x

> instance Functor (S a) where
>    fmap f (St x xs) = St x (f xs)

> type Stream a = Nu (S a)

> headS :: Stream a -> a
> headS xs = case out xs of
>                St x _ -> x

> tailS :: Stream a -> Stream a
> tailS xs = case out xs of
>                St _ xs' -> xs'
```

Functions `headS` and `tailS` are stream destructors, returning the head and the tail of the given stream respectively.

```
> zipS :: (Stream a, Stream a) -> Stream (a,a)
> zipS = ana phi
>     where phi (xs, ys) = St (headS xs, headS ys)
>                             (tailS xs, tailS ys)
```

```
> iterateS :: (a -> a) -> a -> Stream a
> iterateS f = ana phi
>     where phi x = St x (f x)
```

□

Example 2.13 (colists)
Colists have the same base functor as lists, hence we can implement them as follows:

```
> type CoList a = Nu (L a)
```

The destructor function for conaturals, can not be decomposed in general. However, as Haskell allows to use partial functions, we define more intuitive "destructors" as follows:

```
> nullCL :: CoList a -> Bool
> nullCL xs = case out xs of
>                 N     -> True
>                 C _ _ -> False

> headCL :: CoList a -> a
> headCL xs = case out xs of
>                 C x _ -> x

> tailCL :: CoList a -> CoList a
> tailCL xs = case out xs of
>                 C _ xs' -> xs'
```

The function `nullCL` tests whether the colist is empty or not. Partial functions `headCL` and `tailCL` extract respectively the head and the tail of the given non-empty colist. □

2.4 Related work

The categorical treatment of inductive and coinductive types as initial algebras and terminal coalgebras for covariant functors comes from Hagino [Hag87], who designed a typed functional language CPL based on distributive categories and initial algebras and terminal coalgebras for strong covariant functors. The Charity language by Cockett et al. [CF92] is a similar programming language.

The program calculation community is rooted in the Bird-Meertens formalism or Squiggol [Bir87], which, originally, was an equational theory of programming with the parametric data type of lists. Malcolm [Mal90b, Mal90a] made the community aware of Hagino's work and much of the subsequent development followed the path he set. A classic reference in the area of theory is Fokkinga's [Fok92]. The excellent introduction into program calculation is the textbook [BdM97].

CHAPTER 3

PRIMITIVE (CO)RECURSION

This chapter, based on [VU98], is devoted to primitive recursion and primitive corecursion. Primitive recursion is a well known recursion scheme, where the value on the current argument is constructed using the value on the previous argument together with the previous argument itself. Its dualization, primitive corecursion, is not so well known, but provides an equally useful corecursive definition mechanism where a codata structure is generated either step by step (like in the case of coiteration) or in one big step. Both schemes are generalizations of the simple (co)iteration and can be embedded in a nice way into the categorical framework presented in the previous chapter.

3.1 Primitive recursion via tupling

Not every function with inductive type as source can be represented by a single catamorphism alone. For instance, the factorial function $fact : Nat \to Nat$ is neatly characterized as the unique solution of the equation system

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

However, the recursion pattern of the equations above does not follow that of catamorphisms but primitive recursion, i.e. the factorial of a given natural, depends not only on the factorial of its predecessor, but also on the predecessor itself. So, the catamorphic definition of factorial has to compute both in parallel as a pair and then project the factorial component out:

$$fact \quad = \quad \mathrm{fst} \circ (\!|\, [\, \lambda x.(1,0), \lambda(f,n).((n+1) * f, n+1)\,]\, |\!).$$

Meertens [Mee92] showed that the same trick of tupling can be also used for other inductive types. The relevant result is the following:

Lemma 3.1 *For any two arrows $f : \mu\mathsf{F} \to C$ and $\varphi : \mathsf{F}\,(C \times \mu\mathsf{F}) \to C$, we have*

$$f \circ \mathrm{in} = \varphi \circ \mathsf{F}\,\langle\, f, \mathrm{id}\,\rangle \quad \equiv \quad f = \mathrm{fst} \circ (\!|\, \langle\, \varphi, \mathrm{in} \circ \mathsf{F}(\mathrm{snd})\,\rangle \,|\!)$$

Proof. The left-hand equation essentially says that f follows the primitive recursion pattern for μF, while the right one gives its definition in terms of the composition of the left projection and a catamorphism.

The equivalence is proved by the following two calculations. First, from left to right:

$$
\left[\begin{array}{ll}
\triangleright & f \circ \mathrm{in} = \varphi \circ \mathsf{F}\,\langle\, f, \mathrm{id}\,\rangle \\ \hline
 & f \\
= & \quad \text{– pairing –} \\
 & \mathrm{fst} \circ \langle\, f, \mathrm{id}\,\rangle \\
= & \quad \text{– cata-}\textsc{Charn}\text{ –} \\
 & \left[\begin{array}{ll}
 & \langle\, f, \mathrm{id}\,\rangle \circ \mathrm{in} \\
= & \quad \text{– pairing –} \\
 & \langle\, f \circ \mathrm{in}, \mathrm{in}\,\rangle \\
= & \quad \text{– } \mathsf{F} \text{ functor –} \\
 & \langle\, f \circ \mathrm{in}, \mathrm{in} \circ \mathsf{F}\ \mathrm{id}\,\rangle \\
= & \quad \text{– pairing –} \\
 & \langle\, f \circ \mathrm{in}, \mathrm{in} \circ \mathsf{F}\,(\mathrm{snd} \circ \langle\, f, \mathrm{id}\,\rangle)\,\rangle \\
= & \quad \text{– } \triangleleft, \mathsf{F} \text{ functor –} \\
 & \langle\, \varphi \circ \mathsf{F}\,\langle\, f, \mathrm{id}\,\rangle, \mathrm{in} \circ \mathsf{F}\ \mathrm{snd} \circ \mathsf{F}\,\langle\, f, \mathrm{id}\,\rangle\,\rangle \\
= & \quad \text{– pairing –} \\
 & \langle\, \varphi, \mathrm{in} \circ \mathsf{F}\ \mathrm{snd}\,\rangle \circ \mathsf{F}\,\langle\, f, \mathrm{id}\,\rangle
\end{array}\right. \\
 & \mathrm{fst} \circ (\!|\, \langle\, \varphi, \mathrm{in} \circ \mathsf{F}\ \mathrm{snd}\,\rangle \,|\!)
\end{array}\right.
$$

Second, from right to left:

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

$\square$

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).

However, from the practical point of view, the situation is not very satisfactory. First, using tupling is clearly not the most natural way to program primitive recursive functions. Second, algorithms corresponding to the definitions obtained by the lemma above have additional penalty in terms of complexity, as they have to reconstruct the argument which is already there.

3.2 Paramorphisms

To make programming and program reasoning easier, let us introduce a new construction and study its properties.

Definition 3.1 (paramorphism)
Let $(\mu\mathsf{F}, \mathrm{in})$ be an initial F-algebra. For any arrow $\varphi : \mathsf{F}\,(C \times \mu\mathsf{F}) \to C$, the arrow $\langle\!| \varphi |\!\rangle : \mu\mathsf{F} \to C$ is defined by

$$\langle\!| \varphi |\!\rangle \;=\; \mathrm{fst} \circ (\!| \langle \varphi, \mathrm{in} \circ \mathsf{F}(\mathrm{snd}) \rangle |\!) \qquad \text{para-DEF}$$

The arrows in form $\langle\!| \varphi |\!\rangle$ are called *paramorphisms* (derived from the Greek preposition $\pi\alpha\rho\alpha$ meaning 'near to', 'at the side of', 'towards'; the name is due to Meertens [Mee92]). □

The definition made use of the right-hand side of the equivalence in Lemma 3.1. Exploiting the left-hand side, we get the characterization of paramorphisms in terms of universal property.

Corollary 3.2 *For any arrow $\varphi : \mathsf{F}\,(C \times \mu\mathsf{F}) \to C$, the paramorphism $f = \langle\!| \varphi |\!\rangle : \mu\mathsf{F} \to C$ is the unique arrow making the following diagram commute:*

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

i.e. satisfying the universal property:

$$f \circ \mathrm{in} = \varphi \circ \mathsf{F}\langle f, \mathrm{id} \rangle \quad \equiv \quad f = \langle\!| \varphi |\!\rangle \qquad \text{para-CHARN}$$

□

Example 3.1 (primitive recursion for naturals)
Consider the data type of natural numbers. Given any two functions $c : 1 \to C$ and $h : C \times Nat \to C$, the paramorphism $f = \langle\!| [\, c, h \,] |\!\rangle : Nat \to C$ is the unique solution of the equation system

$$\begin{array}{rcl} f \circ zero & = & c \\ f \circ succ & = & h \circ \langle f, \mathrm{id} \rangle, \end{array}$$

i.e. it captures the classical primitive recursion scheme. For instance, the factorial function $fact : Nat \to Nat$ can be defined as paramorphism

$$fact \;=\; \langle\!| [\, one, \lambda(f, n).mul(succ(n), f) \,] |\!\rangle,$$

where $one = succ \circ zero : 1 \to Nat$ and $mul : Nat \times Nat \to Nat$ is the multiplication of naturals defined in Example 2.2. □

Example 3.2 (primitive recursion for lists)
Consider the data type of lists $List(A)$. Given any two functions $c : 1 \to C$ and $h : A \times C \times List(A) \to C$, the paramorphism $f = \langle\!|\, [\, c, h \,] \,|\!\rangle : List(A) \to C$ is the unique solution of the equation system

$$\begin{array}{lcl} f(nil) & = & c() \\ f(cons(x, xs)) & = & h(x, f(xs), xs). \end{array}$$

For instance, the function $tails : List(A) \to List(List(A))$, which returns the list of all tail segments of argument list, can be defined as paramorphism

$$tails \;=\; \langle\!|\, [\, cons(nil, nil), \lambda(x, r, xs).cons\,(cons(x, xs), ys) \,] \,|\!\rangle.$$

Another example of list paramorphism is the function $dropWhile(p) : List(A) \to List(A)$, which for given predicate $p : A \to Bool$ drops the longest initial segment of the argument list such that all elements in this segment satisfy p

$$dropWhile(p) \;=\; \langle\!|\, [\, nil, \lambda(x, r, xs).\texttt{if}\ p(x)\ \texttt{then}\ r\ \texttt{else}\ cons(x, xs) \,] \,|\!\rangle.$$

$\square$

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

Proposition 3.3 *Let* $(\mu\mathsf{F}, \mathrm{in})$ *be an initial* F*-algebra.*

- ***Cancellation:*** *For any arrow* $\varphi : \mathsf{F}\,(C \times \mu\mathsf{F}) \to C$

$$\langle\!|\, \varphi \,|\!\rangle \circ \mathrm{in} = \varphi \circ \mathsf{F}\,\langle\, \langle\!|\, \varphi \,|\!\rangle, \mathrm{id} \,\rangle \qquad \text{para-SELF}$$

- ***Reflection:***

$$\mathrm{id} = \langle\!|\, \mathrm{in} \circ \mathsf{F}(\mathrm{fst}) \,|\!\rangle \qquad \text{para-REFL}$$

- ***Fusion:*** *For any arrows* $\varphi : \mathsf{F}\,(C \times \mu\mathsf{F}) \to C$*,* $\psi : \mathsf{F}\,(D \times \mu\mathsf{F}) \to D$ *and* $f : C \to D$

$$f \circ \varphi = \psi \circ \mathsf{F}(f \times \mathrm{id}) \quad \Rightarrow \quad f \circ \langle\!|\, \varphi \,|\!\rangle = \langle\!|\, \psi \,|\!\rangle \qquad \text{para-FUSION}$$

Proof. The cancellation law is directly obtained form the universal property of paramorphisms by substituting $f := \langle\!|\, \varphi \,|\!\rangle$ thus making the right-hand equation in para-CHARN trivially true. For the reflection law we argue:

$$
\left[\begin{array}{ll}
 & \mathrm{id} \\
= & \qquad \text{– para-}\textsc{Charn}\text{ –} \\
 & \left[\begin{array}{ll}
 & \mathrm{id} \circ \mathrm{in} \\
= & \qquad \text{– identity, } \mathsf{F} \text{ functor –} \\
 & \mathrm{in} \circ \mathsf{F}(\mathrm{id}) \\
= & \qquad \text{– pairing –} \\
 & \mathrm{in} \circ \mathsf{F}(\mathrm{fst} \circ \langle\, \mathrm{id}, \mathrm{id} \,\rangle) \\
= & \qquad \text{– } \mathsf{F} \text{ functor –} \\
 & \mathrm{in} \circ \mathsf{F}(\mathrm{fst}) \circ \mathsf{F}\langle\, \mathrm{id}, \mathrm{id} \,\rangle
\end{array}\right. \\
 & \langle\!|\, \mathrm{in} \circ \mathsf{F}(\mathrm{fst}) \,|\!\rangle
\end{array}\right.
$$

Finally, the fusion law is proved as follows:

$$
\left[\begin{array}{ll}
\rhd & f \circ \varphi = \psi \circ \mathsf{F}(f \times \mathrm{id}) \\
\hline
 & f \circ \langle\!|\, \varphi \,|\!\rangle \\
= & \qquad \text{– para-}\textsc{Charn}\text{ –} \\
 & \left[\begin{array}{ll}
 & f \circ \langle\!|\, \varphi \,|\!\rangle \circ \mathrm{in} \\
= & \qquad \text{– para-}\textsc{Self}\text{ –} \\
 & f \circ \varphi \circ \mathsf{F}\langle\, \langle\!|\, \varphi \,|\!\rangle, \mathrm{id} \,\rangle \\
= & \qquad \text{– } \lhd \text{ –} \\
 & \psi \circ \mathsf{F}(f \times \mathrm{id}) \circ \mathsf{F}\langle\, \langle\!|\, \varphi \,|\!\rangle, \mathrm{id} \,\rangle \\
= & \qquad \text{– } \mathsf{F} \text{ functor –} \\
 & \psi \circ \mathsf{F}((f \times \mathrm{id}) \circ \langle\, \langle\!|\, \varphi \,|\!\rangle, \mathrm{id} \,\rangle) \\
= & \qquad \text{– pairing –} \\
 & \psi \circ \mathsf{F}\langle\, f \circ \langle\!|\, \varphi \,|\!\rangle, \mathrm{id} \,\rangle
\end{array}\right. \\
 & \langle\!|\, \psi \,|\!\rangle
\end{array}\right.
$$

□

In addition to the laws above, paramorphisms satisfy several other useful properties some of which are listed below.

Proposition 3.4

1. *Every catamorphism* $(\!|\, \varphi \,|\!)$ *can be defined as a paramorphism which does not use the preceding value of the argument directly:*

$$(\!|\, \varphi \,|\!) = \langle\!|\, \varphi \circ \mathsf{F}(\mathrm{fst}) \,|\!\rangle \qquad \text{para-}\textsc{Cata}$$

2. *Every arrow whose source is the carrier of an initial algebra is a paramorphism:*

$$f = \langle\!|\, f \circ \mathrm{in} \circ \mathsf{F}(\mathrm{snd}) \,|\!\rangle \qquad \text{para-}\textsc{From-Init}$$

3. *The inverse of an initial algebra is a paramorphism:*

$$\mathrm{in}^{-1} = \langle\!|\, \mathsf{F}(\mathrm{snd}) \,|\!\rangle \qquad \text{para-In-Inv}$$

Proof. The first equality is proven as follows:

$$\left[\begin{array}{ll} & (\!|\, \varphi \,|\!) \\ = & \quad \text{– para-Charn –} \\ & \left[\begin{array}{ll} & (\!|\, \varphi \,|\!) \circ \mathrm{in} \\ = & \quad \text{– cata-Self –} \\ & \varphi \circ \mathsf{F}(\!|\, \varphi \,|\!) \\ = & \quad \text{– pairing –} \\ & \varphi \circ \mathsf{F}(\mathrm{fst} \circ \langle\, (\!|\, \varphi \,|\!), \mathrm{id} \,\rangle) \\ = & \quad \text{– F functor –} \\ & \varphi \circ \mathsf{F}(\mathrm{fst}) \circ \mathsf{F}\langle\, (\!|\, \varphi \,|\!), \mathrm{id} \,\rangle \end{array}\right. \\ & \langle\!|\, \varphi \circ \mathsf{F}(\mathrm{fst}) \,|\!\rangle \end{array}\right.$$

Similarly, the validity of para-From-Init is shown by:

$$\left[\begin{array}{ll} & f \\ = & \quad \text{– para-Charn –} \\ & \left[\begin{array}{ll} & f \circ \mathrm{in} \\ = & \quad \text{– F functor –} \\ & f \circ \mathrm{in} \circ \mathsf{F}(\mathrm{id}) \\ = & \quad \text{– pairing –} \\ & f \circ \mathrm{in} \circ \mathsf{F}(\mathrm{snd} \circ \langle\, f, \mathrm{id} \,\rangle) \\ = & \quad \text{– F functor –} \\ & f \circ \mathrm{in} \circ \mathsf{F}(\mathrm{snd}) \circ \mathsf{F}\langle\, f, \mathrm{id} \,\rangle \end{array}\right. \\ & \langle\!|\, f \circ \mathrm{in} \circ \mathsf{F}(\mathrm{snd}) \,|\!\rangle \end{array}\right.$$

Finally, para-In-Inv comes directly from the previous law:

$$\left[\begin{array}{ll} & \mathrm{in}^{-1} \\ = & \quad \text{– para-From-Init –} \\ & \langle\!|\, \mathrm{in}^{-1} \circ \mathrm{in} \circ \mathsf{F}(\mathrm{snd}) \,|\!\rangle \\ = & \quad \text{– in-inv-Charn –} \\ & \langle\!|\, \mathsf{F}(\mathrm{snd}) \,|\!\rangle \end{array}\right.$$

□

3.3 Apomorphisms

Let us now dualize everything we know about paramorphisms.

Definition 3.2 (apomorphism)
Let $(\nu\mathsf{F}, \mathrm{out})$ be a terminal F-coalgebra. For any arrow $\varphi : C \to \mathsf{F}(C + \nu\mathsf{F})$, define the arrow $\llbracket \varphi \rrbracket : C \to \nu\mathsf{F}$ as a composition of a certain anamorphism with the left injection:

$$\llbracket \varphi \rrbracket \;=\; \llbracket\, [\, \varphi, \mathsf{F}(\mathrm{inr}) \circ \mathrm{out} \,]\, \rrbracket \circ \mathrm{inl} \qquad \text{apo-DEF}$$

The arrows in form $\llbracket \varphi \rrbracket$ are called *apomorphisms* (derived from the Greek preposition $\alpha\pi o$ meaning ‘apart from’, ‘far from’, ‘away from’; the name was first used by Vos [Vos95][1]). □

Corollary 3.5 *For any arrow $\varphi : C \to \mathsf{F}(C + \nu\mathsf{F})$, the apomorphism $f = \llbracket \varphi \rrbracket : C \to \nu\mathsf{F}$ is the unique arrow making the following diagram commute:*

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

i.e. satisfying the universal property:

$$\mathrm{out} \circ f = \mathsf{F}\,[\, f, \mathrm{id} \,] \circ \varphi \quad \equiv \quad f = \llbracket \varphi \rrbracket \qquad \text{apo-CHARN}$$

□

The laws for apomorphisms are just dual to those for paramorphisms.

Corollary 3.6 *Let $(\nu\mathsf{F}, \mathrm{out})$ be a terminal F-coalgebra.*

- ***Cancellation:*** *For any arrow $\varphi : C \to \mathsf{F}(C + \nu\mathsf{F})$*

$$\mathrm{out} \circ \llbracket \varphi \rrbracket = \mathsf{F}[\, \llbracket \varphi \rrbracket, \mathrm{id} \,] \circ \varphi \qquad \text{apo-SELF}$$

- ***Reflection:***

$$\mathrm{id} = \llbracket\, \mathsf{F}(\mathrm{inl}) \circ \mathrm{out} \,\rrbracket \qquad \text{apo-REFL}$$

[1]It is interesting to note that Vene and Uustalu [VU98] unaware of the work by Vos happened to come up with exactly the same new name.

- ***Fusion:*** *For any arrows* $\varphi : C \to \mathsf{F}(C + \nu\mathsf{F})$, $\psi : D \to \mathsf{F}(D + \nu\mathsf{F})$ *and* $f : C \to D$

$$\psi \circ f = \mathsf{F}(f + \mathrm{id}) \circ \varphi \quad \Rightarrow \quad [\![\psi]\!] \circ f = [\![\varphi]\!] \qquad \text{apo-FUSION}$$

□

Corollary 3.7

1. *Every anamorphism* $[\![\varphi]\!]$ *is an apomorphism:*

$$[\![\varphi]\!] = [\![\mathsf{F}(\mathrm{inl}) \circ \varphi]\!] \qquad \text{apo-ANA}$$

2. *Every arrow whose target is the carrier of a terminal coalgebra is an apomorphism:*

$$f = [\![\mathsf{F}(\mathrm{inr}) \circ \mathrm{out} \circ f]\!] \qquad \text{apo-TO-TERM}$$

3. *The inverse of a terminal coalgebra is an apomorphism:*

$$\mathrm{out}^{-1} = [\![\mathsf{F}(\mathrm{inr})]\!] \qquad \text{apo-OUT-INV}$$

□

Example 3.3 (primitive corecursion for streams)
Consider the codata type of streams $\mathit{Stream}(A)$. Given any two functions $h : C \to A$ and $t : C \to C + \mathit{Stream}(A)$, the apomorphism $f = [\![\langle h, t \rangle]\!] : C \to \mathit{Stream}(A)$ is the unique solution of the equation system

$$\begin{array}{lcl} \mathit{head} \circ f & = & h \\ \mathit{tail} \circ f & = & [\, f, \mathrm{id} \,] \circ t. \end{array}$$

Like in the case of anamorphisms, the head of the stream is computed from the current seed value using the function h. However, the tail of the stream can be generated two different ways depending whether the function t computes the new seed value (in which case the generation process proceeds recursively) or the rest of stream as whole. For instance, the function $\mathit{maphd}(h) : \mathit{Stream}(A) \to \mathit{Stream}(A)$, which modifies any input stream by applying a function $h : A \to A$ to its head while leaving the tail unchanged, can be defined as following apomorphism:

$$\mathit{maphd}(h) \;=\; [\![\langle h \circ \mathit{head}, \mathrm{inr} \circ \mathit{tail} \rangle]\!].$$

The role of the function t is more explicit when it is in the form $t = [\, n, r\,] \circ p?$, where $p \to \mathit{Bool}$ is a predicate and $n : C \to C$ and $r : C \to \mathit{Stream}(A)$ are functions for computing next seed or rest of the stream respectively. Then the apomorphism $f = [\!(\langle h, [\, n, r\,] \circ p? \rangle)\!]$ is characterized by equations:

$$\begin{array}{rcll} \mathit{head}(f(x)) & = & h(x) & \\ \mathit{tail}(f(x)) & = & f(n(x)) & \texttt{if } p(x) \\ & = & r(x) & \texttt{otherwise}. \end{array}$$

For an example, assume that a given set A is ordered. The function $\mathit{insert}(a) : \mathit{Stream}(A) \to \mathit{Stream}(A)$ inserts the element $a : 1 \to A$ into a given stream immediately before the first element that is greater than or equal to a (so that the returned stream will be a sorted, if the argument was). It can be defined as an apomorphism:

$$\mathit{insert}(a) = [\!(\langle h, [\, \mathit{tail}, \mathrm{id}\,] \circ p? \rangle)\!],$$

where

$$\begin{array}{rcll} p(xs) & = & \mathit{head}(xs) \leq a() & \\ h(xs) & = & \mathit{head}(xs) & \texttt{if } p(xs) \\ & = & a() & \texttt{otherwise}. \end{array}$$

□

Example 3.4 (primitive corecursion for conaturals)
Given a function $h : C \to 1 + (C + \mathit{CoNat})$, the apomorphism $f = [\!(h)\!] : C \to \mathit{CoNat}$ is the unique solution of the equation system

$$\mathit{pred}(f(x)) = \begin{cases} \mathrm{inl}\,() & \texttt{if } h(x) = \mathrm{inl}\,() \\ \mathrm{inr}(f(x')) & \texttt{if } h(x) = \mathrm{inr}(\mathrm{inl}(x')) \\ \mathrm{inr}(y) & \texttt{if } h(x) = \mathrm{inr}(\mathrm{inr}(y)). \end{cases}$$

For instance, the addition function on conaturals $\mathit{add} = \mathit{CoNat} \times \mathit{CoNat} \to \mathit{CoNat}$, which was defined as an anamorphism in the Example 2.7, can be more succinctly defined as $\mathit{add} = [\!(f)\!]$, where

$$f(x, y) = \begin{cases} \mathrm{inl}\,() & \texttt{if } \mathit{pred}(x) = \mathit{pred}(y) = \mathrm{inl}\,() \\ \mathrm{inr}(\mathrm{inl}(x', y)) & \texttt{if } \mathit{pred}(x) = \mathrm{inr}\ x' \\ \mathrm{inr}(\mathrm{inr}(y')) & \texttt{if } \mathit{pred}(x) = \mathrm{inl}\,(),\ \mathit{pred}(y) = \mathrm{inr}\ y'. \end{cases}$$

The "structured" corecursion operator can be defined if the function h is in the form $h = [\, !_C, [\, n, r\,] \circ p_2?\,] \circ p_1?$, where $p_1 : C \to \mathit{Bool}$ and $p_2 : C \to \mathit{Bool}$ are predicates, $n : C \to C$ gives the next seed and $r : C \to CoNat$ gives the remainder of the conatural under construction. Then $f = [\!\langle [\, !_C, [\, n, r\,] \circ p_2?\,] \circ p_1? \rangle\!] : C \to CoNat$ is characterized by the equations:

$$pred(f(x)) = \begin{cases} \text{inl}\,() & \texttt{if}\; p_1(x) \\ \text{inr}(f(n(x))) & \texttt{if}\; \neg p_1(x) \;\wedge\; p_2(x) \\ \text{inr}(r(y)) & \texttt{otherwise}. \end{cases}$$

□

Example 3.5 (primitive corecursion for colists)
For instance, the function the function $append : CoList(A) \times CoList(A) \to CoList(A)$, which appends two colists is naturally definable as an apomorphism $append = [\!\langle f \rangle\!]$, where

$$\begin{array}{rcll} f(x,y) & = & \text{inl}() & \texttt{if}\; null(x) \wedge null(y) \\ & = & \text{inr}(head(y), \text{inr}(tail(y))) & \texttt{if}\; null(x) \wedge \neg(null(y)) \\ & = & \text{inr}(head(x), \text{inl}(tail(x), y)) & \texttt{otherwise}. \end{array}$$

Here, $null : CoList(A) \to Bool$ is a predicate which tests whether the colist is empty or not, i.e. $null = [\,\text{id}, !_A\,] \circ \text{out}$. □

3.4 Para- and apomorphisms in Haskell

Paramorphisms map arrows of type $\mathsf{F}\,(C \times \mu\mathsf{F}) \to C$ to the arrows of type $\mu\mathsf{F} \to C$. Thus, the type of paramorphism combinator can be expressed in Haskell as follows:

```
> para :: Functor f => (f (c, Mu f) -> c) -> Mu f -> c
```

For the defining equation of paramorphism combinator we have two possibilities. First, we can use the definition of paramorphism in terms of catamorphism:

```
para phi = fst . cata (fork phi (In . fmap snd))
```

where `fork` is pair forming function defined as follows:

```
> fork :: (a -> b) -> (a -> c) -> a -> (b,c)
> fork f g x = (f x, g x)
```

However, this definition is inefficent, as it recursively reconstructs the argument. The second possibility is to use the cancellation law to obtain the directly recursive definition:

```
> para phi = phi . fmap (fork (para phi) id) . unIn
```

This definition is more efficent, as the (previous) argument is used directly by phi.

Example 3.6 (factorial)
The factorial function can be implemented as paramorphism:

```
> fact :: Nat -> Nat
> fact = para phi
>     where phi Z          = succN zeroN
>           phi (S (r,x)) = mulN (succN x) r
```

In the second equation of phi, the result (i.e. factorial) on the previous argument is denoted by r, and the argument itself by x. □

Example 3.7 (dropwhile)
The function *dropWhile* from the example 3.2 can be implemented as follows:

```
> dropWhileL :: (a -> Bool) -> List a -> List a
> dropWhileL p = para phi
>     where phi N             = nilL
>           phi (C x (r,xs)) | p x        = r
>                            | otherwise = consL x xs
```

Here, again, r denotes the value on the previous argument (i.e. value on the tail of the list), while x and xs denote the head and tail of the original list. □

Dually to paramorphisms, apomorphisms map arrows of type $C \to \mathsf{F}(C + \nu\mathsf{F})$ to arrows of type $C \to \nu\mathsf{F}$. As Haskell does not provide a primitive type constructor for sums, we have to define it first:

```
> data Sum a b = InL a | InR b
```

We also define a combinator which does the case analysis on the sum:

```
> join :: (a -> c) -> (b -> c) -> Sum a b -> c
> join f g (InL x) = f x
> join f g (InR y) = g y
```

Now, the type of the apomorphism combinator can be expressed as follows:

```
> apo :: Functor f => (c -> f (Sum c (Nu f)))
>                   ->  c -> Nu f
```

Like in the case of paramorphisms, we have a choise between two possibilities to define the apomorphism combinator. First, the definition in terms of anamorphism:

```
apo phi = ana (join phi (fmap InR . out)) . InL
```

This definition is not very efficent, as it constructs the codata structure in a step-wise fashion even if the whole remaining structure is available (i.e. `phi` returns the right summand). The second possibility is to use directly recursive definition obtained from the cancellation law:

```
> apo phi = Wrap . fmap (join (apo phi) id) . phi
```

In the case of `phi` returns the right summand, this definition is more efficient, as the rest of the structure is returned by one step.

Example 3.8 (insert)
The function *insert* from the example 3.3 is defined as follows:

```
> insertS :: Ord a => a -> Stream a -> Stream a
> insertS a = apo phi
>     where phi xs | x < a     = St x (InL (tailS xs))
>                  | otherwise = St a (InR xs)
>                     where x = headS xs
```

□

Example 3.9 (append)
The concatenation of two colists can be implemented as apomorphism:

```
> appendCL :: (CoList a, CoList a) -> CoList a
> appendCL = apo phi
>     where phi (xs, ys)
>             | nullCL xs && nullCL ys = N
>             | nullCL xs  = C (headCL ys)
>                              (InR (tailCL ys))
>             | otherwise  = C (headCL xs)
>                              (InL (tailCL xs, ys))
```

□

3.5 Related work

Primitive recursion is universally recognized as an important generalization of iteration. Paramorphisms were introduced by Meertens [Mee92]. Geuvers [Geu92] contains a thorough category-theoretic 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. [Fok92] ignores it. The sole discussion of primitive corecursion in a programming context that we knew about when writing [VU98] was the laconic report in [Ves97] on a not very clean extension to the categorical functional language Charity in which it is possible to define functions by primitive recursion and primitive corecursion. Soon after we got to know of [Vos95]. Citations of [VU98] appear in [GH99, BBA00].

CHAPTER 4

COURSE-OF-VALUE (CO)ITERATION

In this chapter, which is based on [UV99b], we introduce categorical combinators for course-of-value iteration and coiteration. Course-of-value iteration is a recursion scheme, where the value on the current argument is constructed using the values for the subparts of the argument on arbitrary (but fixed) depth. Dually, course-of-value coiteration allows to generate several "levels" of a resulting codata structure in one step.

4.1 Course-of-value iteration via memoization

The famous Fibonacci function $\mathit{fibo} : \mathit{Nat} \to \mathit{Nat}$ is most smoothly characterized as the unique solution of the equation system

$$\begin{array}{rcl} \mathit{fibo}(0) & = & 1 \\ \mathit{fibo}(1) & = & 1 \\ \mathit{fibo}(n+2) & = & \mathit{fibo}(n+1) + \mathit{fibo}(n). \end{array}$$

This very nice characterization does not give us any definition of *fibo* in terms of catamorphisms. The problem is that the value of *fibo* for a given argument is defined not via the values for the immediate subparts of the argument, but via the values for its subparts of depth 2. But the characterization of *fibo* together with the function $\mathit{fibo}' : \mathit{Nat} \to \mathit{Nat} \times \mathit{Nat}$ (which, for any argument n, returns the pair formed of the value of *fibo* for n and either zero or the value of *fibo* for the predecessor of n) by a much trickier equation system, viz.,

$$\begin{array}{rcl} \mathit{fibo} & = & \mathrm{fst} \circ \mathit{fibo}' \\ \mathit{fibo}'(0) & = & (1, 0) \\ \mathit{fibo}'(n+1) & = & \langle\, \mathit{add}, \mathrm{fst}\,\rangle(\mathit{fibo}'(n)), \end{array}$$

leads to a definition of *fibo* as the composition of the left projection and a catamorphism:

$$\mathit{fibo} = \mathrm{fst} \circ \llparenthesis \langle\, [\, \lambda x.1, \mathit{add}\,], [\, \lambda x.0, \mathrm{fst}\,]\, \rangle \rrparenthesis .$$

Now, we could introduce a new construction that would capture the natural definition scheme of the Fibonacci function and closely similar functions, and start studying its properties. But this would only provide us with a partial solution to the problem manifested by the Fibonacci example, as one can imagine functions whose value for a given argument is naturally defined via the values for its subparts of depth three, four, etc. Instead of this, we introduce a construction that captures a general course-of-value iteration by collecting the values on all subparts into a certain codata structure.

Definition 4.1 (cv-algebra)
Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor for which there exists an initial algebra $(\mu\mathsf{F}, \mathrm{in})$. Define a bifunctor $\mathsf{F}^{\times} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ as follows:

$$\mathsf{F}^{\times}(A, X) = A \times \mathsf{F}(X).$$

Assume that for any object A there exists a terminal $\mathsf{F}^{\times}_A$-coalgebra $(\nu\mathsf{F}^{\times}_A, \mathrm{out})$, i.e. $\mathsf{F}^{\times}$ induces a codata functor $\mathsf{F}^{\nu}(X) = \nu\mathsf{F}^{\times}_X$. A F-*cv-algebra* is a pair (C, φ), where C is an object and $\varphi : \mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$ is an arrow. $\square$

Definition 4.2 (cv-algebra homomorphism)
Let (C, φ) and (D, ψ) be two F-cv-algebras. A *homomorphism* from (C, φ) to (D, ψ) is an arrow $f : C \to D$ in the category $\mathcal{C}$, such that

$$f \circ \varphi \;=\; \psi \circ \mathsf{F}\llbracket\, (f \times \mathrm{id}) \circ \mathrm{out}\, \rrbracket$$

i.e. makes the following diagram to commute:

$$\begin{tikzcd}
\mathsf{F}(\mathsf{F}^{\nu}(C)) \arrow[rr, "\varphi"] \arrow[d, "\mathsf{F}\llbracket (f\times\mathrm{id})\circ\mathrm{out} \rrbracket"'] & & C \arrow[d, "f"] \\
\mathsf{F}(\mathsf{F}^{\nu}(D)) \arrow[rr, "\psi"] & & D
\end{tikzcd}$$

$\square$

Note that any F-cv-algebra is an ordinary algebra for a functor $\mathsf{G}(X) = \mathsf{F}(\mathsf{F}^{\nu}(X))$, and homomorphisms between F-cv-algebras are ordinary homomorphisms between G-algebras.

The next result, analogous to Lemma 3.1 for primitive recursion, states that every function which can be specified using course-of-value iteration, can be defined in terms of catamorphism which builds a codata structure of values of the function on the all substructures of its argument (essentially, the catamorphism builds a memo-table [Mic68] for the function).

Lemma 4.1 *For any arrow* $f : \mu\mathsf{F} \to C$ *and* F*-cv-algebra* $\varphi : \mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$, *we have*

$$f \circ \mathrm{in} = \varphi \circ \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket \quad \equiv \quad f = \mathrm{fst} \circ \mathrm{out} \circ \llparenthesis\, \mathrm{out}^{-1} \circ \langle\, \varphi, \mathrm{id}\,\rangle\,\rrparenthesis$$

Proof. Proving it is quite tricky. From left to right we calculate:

$$
\begin{array}{ll}
\rhd & f \circ \mathrm{in} = \varphi \circ \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket \\
\hline
 & f \\
= & \quad \text{– pairing –} \\
 & \mathrm{fst} \circ \langle\, f, \mathrm{in}^{-1}\,\rangle \\
= & \quad \text{– pairing –} \\
 & \mathrm{fst} \circ (\mathrm{id} \times \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket) \circ \langle\, f, \mathrm{in}^{-1}\,\rangle \\
= & \quad \text{– ana-}\textsc{Self}\text{ –} \\
 & \mathrm{fst} \circ \mathrm{out} \circ \llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket \\
= & \quad \text{– cata-}\textsc{Charn}\text{ –} \\
 & \quad \begin{array}{ll}
 & \llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket \circ \mathrm{in} \\
= & \quad \text{– out-inv-}\textsc{Charn}\text{ –} \\
 & \mathrm{out}^{-1} \circ \mathrm{out} \circ \llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket \circ \mathrm{in} \\
= & \quad \text{– ana-}\textsc{Self}\text{ –} \\
 & \mathrm{out}^{-1} \circ (\mathrm{id} \times \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket) \circ \langle\, f, \mathrm{in}^{-1}\,\rangle \circ \mathrm{in} \\
= & \quad \text{– pairing –} \\
 & \mathrm{out}^{-1} \circ \langle\, f, \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket \circ \mathrm{in}^{-1}\,\rangle \circ \mathrm{in} \\
= & \quad \text{– pairing –} \\
 & \mathrm{out}^{-1} \circ \langle\, f \circ \mathrm{in}, \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket \circ \mathrm{in}^{-1} \circ \mathrm{in}\,\rangle \\
= & \quad \text{– } \lhd \text{, in-inv-}\textsc{Charn}\text{ –} \\
 & \mathrm{out}^{-1} \circ \langle\, \varphi \circ \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket, \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket\,\rangle \\
= & \quad \text{– pairing –} \\
 & \mathrm{out}^{-1} \circ \langle\, \varphi, \mathrm{id}\,\rangle \circ \mathsf{F}\,\llbracket\, \langle\, f, \mathrm{in}^{-1}\,\rangle\,\rrbracket
\end{array} \\
 & \mathrm{fst} \circ \mathrm{out} \circ \llparenthesis\, \mathrm{out}^{-1} \circ \langle\, \varphi, \mathrm{id}\,\rangle\,\rrparenthesis
\end{array}
$$

From right to left we argue:

$$
\begin{array}{ll}
\rhd & f = \mathrm{fst} \circ \mathrm{out} \circ (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \\
\hline
 & f \circ \mathrm{in} \\
= & \quad - \lhd - \\
 & \mathrm{fst} \circ \mathrm{out} \circ (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \circ \mathrm{in} \\
= & \quad - \text{cata-}\textsc{Self} - \\
 & \mathrm{fst} \circ \mathrm{out} \circ \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle \circ \mathsf{F}\, (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \\
= & \quad - \text{out-inv-}\textsc{Charn} - \\
 & \mathrm{fst} \circ \langle \varphi, \mathrm{id} \rangle \circ \mathsf{F}\, (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \\
= & \quad - \text{pairing} - \\
 & \varphi \circ \mathsf{F}\, (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \\
= & \quad - \text{ana-}\textsc{Charn} - \\
 & \quad \left[\begin{array}{ll}
 & \mathrm{out} \circ (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \\
= & \quad - \text{pairing} - \\
 & \langle \mathrm{fst} \circ \mathrm{out} \circ (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!), \mathrm{snd} \circ \mathrm{out} \circ (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \rangle \\
= & \quad - \lhd, \text{in-inv-}\textsc{Charn} - \\
 & \langle f, \mathrm{snd} \circ \mathrm{out} \circ (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \circ \mathrm{in} \circ \mathrm{in}^{-1} \rangle \\
= & \quad - \text{cata-}\textsc{Self} - \\
 & \langle f, \mathrm{snd} \circ \mathrm{out} \circ \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle \circ \mathsf{F}\, (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \circ \mathrm{in}^{-1} \rangle \\
= & \quad - \text{out-inv-}\textsc{Charn} - \\
 & \langle f, \mathrm{snd} \circ \langle \varphi, \mathrm{id} \rangle \circ \mathsf{F}\, (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \circ \mathrm{in}^{-1} \rangle \\
= & \quad - \text{pairing} - \\
 & \langle f, \mathsf{F}\, (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \circ \mathrm{in}^{-1} \rangle \\
= & \quad - \text{pairing} - \\
 & (\mathrm{id} \times \mathsf{F}\, (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!)) \circ \langle f, \mathrm{in}^{-1} \rangle
\end{array} \right. \\
 & \varphi \circ \mathsf{F}\, [\!(\langle f, \mathrm{in}^{-1} \rangle)\!]
\end{array}
$$

□

4.2 Histomorphisms

To make programming and program reasoning easier, let us introduce a new construction and study its properties.

Definition 4.3 (histomorphism)
Let $(\mu\mathsf{F}, \mathrm{in})$ be an initial F-algebra. For any F-cv-algebra $\varphi : \mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$, the arrow $\{\!| \varphi |\!\} : \mu\mathsf{F} \to C$ is defined by

$$\{\!| \varphi |\!\} \;=\; \mathrm{fst} \circ \mathrm{out} \circ (\!| \mathrm{out}^{-1} \circ \langle \varphi, \mathrm{id} \rangle |\!) \qquad \text{histo-}\textsc{Def}$$

The arrows in form $\{\!| \varphi |\!\}$ are called *histomorphisms*. □

By Lemma 4.1, we get the characterization of histomorphisms in terms of universal property.

Corollary 4.2 *For any* F*-cv-algebra* $\varphi : \mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$*, the histomorphism* $f = \{\!|\,\varphi\,|\!\} : \mu\mathsf{F} \to C$ *is the unique arrow making the following diagram commute:*

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, "\mathrm{in}"] \arrow[d, "\mathsf{F}\llbracket\langle f, \mathrm{in}^{-1}\rangle\rrbracket"'] & \mu\mathsf{F} \arrow[d, "f"] \\
\mathsf{F}(\mathsf{F}^{\nu}(C)) \arrow[r, "\varphi"] & C
\end{tikzcd}$$

i.e. satisfying the universal property:

$$f \circ \mathrm{in} = \varphi \circ \mathsf{F}\llbracket\langle f, \mathrm{in}^{-1}\rangle\rrbracket \quad \equiv \quad f = \{\!|\,\varphi\,|\!\} \qquad \text{histo-}\textsc{Charn}$$

□

From the universal property, we also get the cancellation, reflection and fusion laws for histomorphisms:

Proposition 4.3 *Let* $(\mu\mathsf{F}, \mathrm{in})$ *be an initial* F*-algebra.*

- ***Cancellation:*** *For any* F*-cv-algebra* $\varphi : \mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$

$$\{\!|\,\varphi\,|\!\} \circ \mathrm{in} = \varphi \circ \mathsf{F}\llbracket\langle \{\!|\,\varphi\,|\!\}, \mathrm{in}^{-1}\rangle\rrbracket \qquad \text{histo-}\textsc{Self}$$

- ***Reflection:***

$$\mathrm{id} = \{\!|\,\mathrm{in} \circ \mathsf{F}\,(\mathrm{fst} \circ \mathrm{out})\,|\!\} \qquad \text{histo-}\textsc{Refl}$$

- ***Fusion:*** *For any* F*-cv-algebra* $\varphi : \mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$*,* $\psi : \mathsf{F}(\mathsf{F}^{\nu}(D)) \to D$ *and an arrow* $f : C \to D$

$$f \circ \varphi = \psi \circ \mathsf{F}\llbracket(f \times \mathrm{id}) \circ \mathrm{out}\rrbracket \quad \Rightarrow \quad f \circ \{\!|\,\varphi\,|\!\} = \{\!|\,\psi\,|\!\} \qquad \text{histo-}\textsc{Fusion}$$

Proof. The cancellation law is directly obtained from the universal property of histomorphisms by substituting $f := \{\!|\,\varphi\,|\!\}$ thus making the right-hand equation in histo-CHARN trivially true. For the reflection law we argue:

$$
\left[\begin{array}{ll}
 & \mathrm{id} \\
= & \quad \text{– histo-CHARN –} \\
 & \left[\begin{array}{ll}
 & \mathrm{id} \circ \mathrm{in} \\
= & \quad \text{– identity, F functor –} \\
 & \mathrm{in} \circ \mathsf{F}(\mathrm{id}) \\
= & \quad \text{– pairing –} \\
 & \mathrm{in} \circ \mathsf{F}\,(\mathrm{fst} \circ \langle\, \mathrm{id}, \mathrm{in}^{-1} \,\rangle) \\
= & \quad \text{– pairing –} \\
 & \mathrm{in} \circ \mathsf{F}\,(\mathrm{fst} \circ (\mathrm{id} \times \mathsf{F}\,[\!(\, \langle\, \mathrm{id}, \mathrm{in}^{-1} \,\rangle \,)\!]) \circ \langle\, \mathrm{id}, \mathrm{in}^{-1} \,\rangle) \\
= & \quad \text{– ana-SELF –} \\
 & \mathrm{in} \circ \mathsf{F}\,(\mathrm{fst} \circ \mathrm{out} \circ [\!(\, \langle\, \mathrm{id}, \mathrm{in}^{-1} \,\rangle \,)\!]) \\
= & \quad \text{– F functor –} \\
 & \mathrm{in} \circ \mathsf{F}\,(\mathrm{fst} \circ \mathrm{out}) \circ \mathsf{F}\,[\!(\, \langle\, \mathrm{id}, \mathrm{in}^{-1} \,\rangle \,)\!]
\end{array}\right. \\
 & \{\!|\, \mathrm{in} \circ \mathsf{F}(\mathrm{fst} \circ \mathrm{out}) \,|\!\}
\end{array}\right.
$$

Finally, the fusion law is proved as follows:

$$
\left[\begin{array}{ll}
\rhd & f \circ \varphi = \psi \circ \mathsf{F}\,[\!(\, (f \times \mathrm{id}) \circ \mathrm{out} \,)\!] \\
\hline
 & f \circ \{\!|\, \varphi \,|\!\} \\
= & \quad \text{– histo-CHARN –} \\
 & \left[\begin{array}{ll}
 & f \circ \{\!|\, \varphi \,|\!\} \circ \mathrm{in} \\
= & \quad \text{– histo-SELF –} \\
 & f \circ \varphi \circ \mathsf{F}\,[\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!] \\
= & \quad - \lhd - \\
 & \psi \circ \mathsf{F}\,[\!(\, (f \times \mathrm{id}) \circ \mathrm{out} \,)\!] \circ \mathsf{F}\,[\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!] \\
= & \quad \text{– F functor –} \\
 & \psi \circ \mathsf{F}\,([\!(\, (f \times \mathrm{id}) \circ \mathrm{out} \,)\!] \circ [\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!]) \\
= & \quad \text{– ana-FUSION –} \\
 & \left[\begin{array}{ll}
 & (f \times \mathrm{id}) \circ \mathrm{out} \circ [\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!] \\
= & \quad \text{– ana-SELF –} \\
 & (f \times \mathrm{id}) \circ (\mathrm{id} \times \mathsf{F}\,[\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!]) \circ \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \\
= & \quad \text{– pairing –} \\
 & (f \times \mathsf{F}\,[\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!]) \circ \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \\
= & \quad \text{– pairing –} \\
 & \langle\, f \circ \{\!|\, \varphi \,|\!\}, \mathsf{F}\,[\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!] \circ \mathrm{in}^{-1} \,\rangle \\
= & \quad \text{– pairing –} \\
 & (\mathrm{id} \times \mathsf{F}\,[\!(\, \langle\, \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!]) \circ \langle\, f \circ \{\!|\, \varphi \,|\!\}, \mathrm{in}^{-1} \,\rangle
\end{array}\right. \\
 & \psi \circ \mathsf{F}\,[\!(\, \langle\, f \circ \{\!|\, \phi \,|\!\}, \mathrm{in}^{-1} \,\rangle \,)\!]
\end{array}\right. \\
 & \{\!|\, \psi \,|\!\}
\end{array}\right.
$$

$\square$

Read from left to right, the cancellation law can be treated as the reduction rule for histomorphisms. Informally it tells that, the value of the histomorphism on the given argument is computed by first building a certain "colist" and then using a cv-algebra to give the final result. The "colist" is generated using an anamorphism which gets the previous argument as the initial seed. On every step, the anamorphism computes (recursively) the value of the histomorphism on the current seed, and also a new seed by taking a "predecessor" of the current one.

The left-hand side of the fusion law states that f is a homomorphism between cv-algebras φ and ψ. Hence every cv-algebra homomorphism can be fused with a histomorphism.

Similarly to paramorphisms, histomorphisms can be viewed as a generalization of catamorphisms. Namely, every catamorphism is a histomorphism which uses only the value on the "predecessor" of the current argument (i.e. the "head" of the "colist").

Proposition 4.4 *For any* F*-algebra* $\varphi : \mathsf{F}(C) \to C$*,*

$$(\!|\, \varphi \,|\!) = \{\!|\, \varphi \circ \mathsf{F}\,(\mathrm{fst} \circ \mathrm{out}) \,|\!\} \qquad \text{histo-}\textsc{Cata}$$

Proof. It is verified by the following calculation:

$$\begin{array}{ll}
 & (\!|\, \varphi \,|\!) \\
= & \quad \text{– histo-}\textsc{Charn}\text{ –} \\
 & \left[\begin{array}{ll}
 & (\!|\, \varphi \,|\!) \circ \mathrm{in} \\
= & \quad \text{– cata-}\textsc{Self}\text{ –} \\
 & \varphi \circ \mathsf{F}\,(\!|\, \varphi \,|\!) \\
= & \quad \text{– pairing –} \\
 & \varphi \circ \mathsf{F}\,(\mathrm{fst} \circ \langle\, (\!|\, \varphi \,|\!), \mathrm{in}^{-1} \,\rangle) \\
= & \quad \text{– pairing –} \\
 & \varphi \circ \mathsf{F}\,(\mathrm{fst} \circ (\mathrm{id} \times \mathsf{F}\,[\!(\, \langle\, (\!|\, \varphi \,|\!), \mathrm{in}^{-1} \,\rangle \,)\!]) \circ \langle\, (\!|\, \varphi \,|\!), \mathrm{in}^{-1} \,\rangle) \\
= & \quad \text{– ana-}\textsc{Self}\text{ –} \\
 & \varphi \circ \mathsf{F}\,(\mathrm{fst} \circ \mathrm{out} \circ [\!(\, \langle\, (\!|\, \varphi \,|\!), \mathrm{in}^{-1} \,\rangle \,)\!]) \\
= & \quad \text{– }\mathsf{F}\text{ functor –} \\
 & \varphi \circ \mathsf{F}\,(\mathrm{fst} \circ \mathrm{out}) \circ \mathsf{F}\,[\!(\, \langle\, (\!|\, \varphi \,|\!), \mathrm{in}^{-1} \,\rangle \,)\!]
\end{array}\right. \\
 & \{\!|\, \varphi \circ \mathsf{F}\,(\mathrm{fst} \circ \mathrm{out}) \,|\!\}
\end{array}$$

□

Example 4.1 (course-of-value iteration for naturals)
Consider the data type of natural numbers; i.e. the initial N-algebra $(Nat, [\, zero, succ \,])$. The codata type $\mathsf{N}^{\nu}(C)$ consists of nonempty colists over C, and the terminal coalgebra structure is provided by $\mathrm{out} = \langle\, cur, prev \,\rangle$:

$\mathsf{N}^\nu(C) \to C \times (1 + \mathsf{N}^\nu(C))$, where $cur : \mathsf{N}^\nu(C) \to C$ gives the head and $prev : \mathsf{N}^\nu(C) \to 1 + \mathsf{N}^\nu(C)$ the (possible) tail of a colist.

Any N-cv-algebra $\varphi : 1 + \mathsf{N}^\nu(C) \to C$ can be decomposed using join $\varphi = [\, z_0, s_0 \,]$, where $z_0 : 1 \to C$ and $s_0 : \mathsf{N}^\nu(C) \to C$. The histomorphism $f = \{\!|\, [\, z_0, s_0 \,] \,|\!\} : Nat \to C$ is the unique solution of the equation system:

$$\begin{array}{rcl} f(zero()) &=& z_0() \\ f(succ(x)) &=& s_0([\![\, \langle f, pred \rangle \,]\!](x)), \end{array}$$

where $pred : Nat \to 1 + Nat$ is the predecessor function from the Example 2.2.

In order to get more illuminating version of the course-of-value iteration operator, assume that the function s_0 is in form $s_0 = [\, z_1 \circ !, s_1 \,] \circ \mathrm{distr} \circ \mathrm{out}$ for some constant[1] $z_1 : 1 \to C$ and function $s_1 : C \times \mathsf{N}^\nu(C) \to C$. Then the corresponding histomorphism $f = \{\!|\, [\, z_0, [\, z_1 \circ !, s_1 \,] \circ \mathrm{distr} \circ \mathrm{out} \,] \,|\!\}$ is characterized by the equations:

$$\begin{array}{rcl} f(zero()) &=& z_0() \\ f(succ(zero())) &=& z_1() \\ f(succ(x)) &=& s_1(f(x), [\![\, \langle f, pred \rangle \,]\!](x)). \end{array}$$

Now, the use of the value on the previous argument is explicit. Particularly, by taking $z_0 = one$, $z_1 = one$ and $s_1(x, y) = add(x, cur(y))$, we get the definition of the Fibonacci function; i.e.

$$fibo \;=\; \{\!|\, [\, one, [\, one \circ !, add \circ (\mathrm{id} \times cur) \,] \circ \mathrm{distr} \circ \mathrm{out} \,] \,|\!\}.$$

The general form of course-of-value iteration operator involves $n+1$ constants $z_0, \ldots, z_n : 1 \to C$ and a function $s_n : C \times (\ldots (C \times \mathsf{N}^\nu(C)) \ldots) \to C$ (here the product has $n + 1$ components). The corresponding histomorphism is

$$f = \{\!|\, [\, z_0, [\, z_1 \circ !, [\, z_2 \circ !, \ldots [\, z_n \circ !, s_n \,] \circ dout \ldots] \circ dout \,] \circ dout \,] \,|\!\},$$

where $dout = \mathrm{distr} \circ \mathrm{out}$. It is characterized by the system of $n + 2$ equations:

$$\begin{array}{rcl} f(zero()) &=& z_0() \\ f(succ(zero())) &=& z_1() \\ & \ldots & \\ f(succ^n(zero())) &=& z_n() \\ f(succ^n(x)) &=& s_n(f^n(x), \ldots, f(x), [\![\, \langle f, pred \rangle \,]\!](x)). \end{array}$$

[1] Instead of the constant, the left branch of the join could be a function $h : C \to C$ which makes use of the value on the previous argument. However, the previous argument is known to be $zero$ and the value on it is z_0, thus result on the argument $succ \circ zero$ is already known to be $z_1 = h(z_0)$.

Note that in such way we can characterize functions which make use of arbitrary but fixed number of preceding values. Of course, we can imagine functions which make use of all preceding values (such recursion scheme is called *course-of-value recursion*). The classical example of such function is $f(n) = 2^n$, which can be characterized as:

$$f(n) \ = \ 1 + f(n-1) + \cdots + f(1) + f(0).$$

If we rewrite the equation in a more explicit form

$$f(n) \ = \ 1 + \sum_{i=0}^{n-1} f(i),$$

we see that the use of all preceding values can be achieved by using primitive recursion and course-of-value iteration at the same time. □

4.3 Futumorphisms

We now introduce a construction dual to cv-algebras and histomorphisms.

Definition 4.4 (cv-coalgebra)
Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor for which there exists a terminal coalgebra $(\nu\mathsf{F}, \mathrm{out})$. Define a bifunctor $\mathsf{F}^{+} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ as follows:

$$\mathsf{F}^{+}(A, X) = A + \mathsf{F}(X).$$

Assume that for any object A there exists an initial F^{+}_{A}-algebra $(\mu\mathsf{F}^{+}_{A}, \mathrm{in})$, i.e. F^{+} induces a data functor $\mathsf{F}^{\mu}(X) = \mu\mathsf{F}^{+}_{X}$. A F-*cv-coalgebra* is a pair (C, φ), where C is an object and $\varphi : C \to \mathsf{F}(\mathsf{F}^{+}(C))$ is an arrow. □

Definition 4.5 (cv-coalgebra homomorphism)
Let (C, φ) and (D, ψ) be two F-cv-algebras. A *homomorphism* from (C, φ) to (D, ψ) is an arrow $f : C \to D$ in the category $\mathcal{C}$, such that

$$\psi \circ f = \mathsf{F}\,(\!|\,\mathrm{in} \circ (f + \mathrm{id})\,|\!) \circ \varphi$$

i.e. makes the following diagram to commute:

$$\begin{tikzcd}
C \arrow[r, "\varphi"] \arrow[d, "f"'] & \mathsf{F}(\mathsf{F}^{\mu}(C)) \arrow[d, "\mathsf{F}\,(\!|\,\mathrm{in} \circ (f + \mathrm{id})\,|\!)"] \\
D \arrow[r, "\psi"] & \mathsf{F}(\mathsf{F}^{\mu}(D))
\end{tikzcd}$$

□

Note that any F-cv-coalgebra is an ordinary coalgebra for a functor $\mathsf{G}(X) = \mathsf{F}(\mathsf{F}^{\mu}(X))$, and homomorphisms between F-cv-coalgebras are ordinary homomorphisms between G-coalgebras.

Definition 4.6 (futumorphism)
Let $(\nu\mathsf{F}, \mathrm{out})$ be a terminal F-coalgebra. For any F-cv-coalgebra $\varphi : C \to \mathsf{F}(\mathsf{F}^{\mu}(C))$, the arrow $\llbracket\, \varphi \,\rrbracket : C \to \nu\mathsf{F}$ is defined by

$$\llbracket\, \varphi \,\rrbracket \;=\; [\!(\, [\, \varphi, \mathrm{id} \,] \circ \mathrm{in}^{-1} \,)\!] \circ \mathrm{in} \circ \mathrm{inl} \qquad \text{futu-DEF}$$

The arrows in form $\llbracket\, \varphi \,\rrbracket$ are called *futumorphisms*. □

Corollary 4.5 *For any* F*-cv-coalgebra* $\varphi : C \to \mathsf{F}(\mathsf{F}^{+}(C))$*, the futumorphism* $f = \llbracket\, \varphi \,\rrbracket : C \to \nu\mathsf{F}$ *is the unique arrow making the following diagram commute:*

$$\begin{tikzcd}
C \arrow[r, "\varphi"] \arrow[d, "f"'] & \mathsf{F}(\mathsf{F}^{\mu}(C)) \arrow[d, "\mathsf{F}\,(\!|\, [\, f, \mathrm{out}^{-1} \,] \,|\!)"] \\
\nu\mathsf{F} \arrow[r, "\mathrm{out}"] & \mathsf{F}(\nu\mathsf{F})
\end{tikzcd}$$

i.e. satisfying the universal property:

$$\mathrm{out} \circ f = \mathsf{F}\,(\!|\, [\, f, \mathrm{out}^{-1} \,] \,|\!) \circ \varphi \quad \equiv \quad f = \llbracket\, \varphi \,\rrbracket \qquad \text{futu-CHARN}$$

□

By a straightforward dualization of the laws for histomorphisms, we get the corresponding laws for futumorphisms.

Corollary 4.6 *Let* $(\mu\mathsf{F}, \mathrm{in})$ *be an initial* F*-algebra.*

- ***Cancellation:*** *For any* F*-cv-coalgebra* $\varphi : C \to \mathsf{F}(\mathsf{F}^{+}(C))$

$$\mathrm{out} \circ \llbracket\, \varphi \,\rrbracket = \mathsf{F}\,(\!|\, [\, \llbracket\, \varphi \,\rrbracket, \mathrm{out}^{-1} \,] \,|\!) \circ \varphi \qquad \text{futu-SELF}$$

- ***Reflection:***

$$\mathrm{id} = \llbracket\, \mathsf{F}\,(\mathrm{in} \circ \mathrm{inl}) \circ \mathrm{out} \,\rrbracket \qquad \text{futu-REFL}$$

- ***Fusion:*** *For any* F*-cv-coalgebra* $\varphi : C \to \mathsf{F}(\mathsf{F}^{+}(C))$, $\psi : D \to \mathsf{F}(\mathsf{F}^{+}(D))$ *and an arrow* $f : C \to D$

$$\psi \circ f = \mathsf{F}\,(\!|\, \mathrm{in} \circ (f + \mathrm{id}) \,|\!) \circ \varphi \quad \Rightarrow \quad \llbracket\, \psi \,\rrbracket \circ f = \llbracket\, \varphi \,\rrbracket \qquad \text{futu-FUSION}$$

- ***Ana from futu:*** *For any* F*-coalgebra* $\varphi : C \to \mathsf{F}(C)$

$$[\!(\, \varphi \,)\!] = [\![\, \mathsf{F}\,(\mathrm{in} \circ \mathrm{inl}) \circ \varphi \,]\!] \qquad \text{futu-ANA}$$

□

Example 4.2 (course-of-value coiteration for streams)
Recall, that the terminal coalgebra for a bifunctor $\mathsf{S}(A, X) = A \times X$ was given by $\mathit{Stream}(A)$, the codata type of streams over A, with the coalgebra structure $\langle\, \mathit{head}, \mathit{tail} \,\rangle : \mathit{Stream}(A) \to A \times \mathit{Stream}(A)$. The inductive data type manifesting in stream futumorphisms is given by the induced data bifunctor $\mathsf{S}^{\mu}(C, A) = \mu\mathsf{S}^{+}_{C,A}$, where $\mathsf{S}^{+}(C, A, X) = C + A \times X$; i.e. data type of nonempty lists where all elements except the last one are from type A, and the last element is of type C. The initial algebra structure is given by $[\, l, c \,] : C + A \times \mathsf{S}^{\mu}(C, A) \to \mathsf{S}^{\mu}(C, A)$, where $l : C \to \mathsf{S}^{\mu}(C, A)$ constructs the singleton list and $c : A \times \mathsf{S}^{\mu}(C, A) \to \mathsf{S}^{\mu}(C, A)$ inserts the new element of type A into first position.

Every S_A-cv-coalgebra $\varphi : C \to A \times \mathsf{S}^{\mu}(C, A)$ can be decomposed using fork $\varphi = \langle\, h_0, t \,\rangle$, where $h_0 : C \to A$ and $t : C \to \mathsf{S}^{\mu}(C, A)$. The futumorphism $f = [\![\, \langle\, h_0, t \,\rangle \,]\!] : C \to \mathit{Stream}(A)$ is the unique solution of the equation system:

$$\begin{array}{rcl} \mathit{head}(f(x)) & = & h_0(x) \\ \mathit{tail}(f(x)) & = & (\!|\, [\, f, \mathit{cons} \,] \,|\!)(t(x)), \end{array}$$

where $\mathit{cons} = \mathrm{out}^{-1} : A \times \mathit{Stream}(A) \to \mathit{Stream}(A)$. Intuitively, the function t in produces a list of stream elements going to follow just next after the current head, and also a new seed as the last element of the list. Then the catamorphism replaces the list constructors c with the "stream constructor" cons, thus forming an initial prefix of the tail stream. Finally, the last constructor l, which contains the new seed, is replaced by f, which continues recursively to produce the rest of the stream.

Assume that the function t explicitly constructs the list of $n + 1$ elements; i.e. it is in form

$$t(x) \;=\; c(h_1(x), (c(h_2(x), \ldots c(h_n(x), l(s(x))) \ldots))),$$

where $h_1, \ldots, h_n : C \to A$ and $s : C \to C$. Then the futumorphism $f = [\![\, \langle\, h_0, c \circ \langle\, h_1, \ldots c \circ \langle\, h_n, l \circ s \,\rangle \ldots \rangle\,\rangle \,]\!] : C \to \mathit{Stream}(A)$ is characterized by a system of $n + 2$ equations:

$$\begin{array}{rcl} \mathit{head}(f(x)) & = & h_0(x) \\ \mathit{head}(\mathit{tail}(f(x))) & = & h_1(x) \\ & \cdots & \\ \mathit{head}(\mathit{tail}^n(f(x))) & = & h_n(x) \\ \mathit{tail}(\mathit{tail}^n(f(x))) & = & f(s(x)). \end{array}$$

For instance, the function $exch : Stream(A) \to Stream(A)$, which pairwise exchanges the elements of any given argument, is characterized by the equation system

$$\begin{array}{rcl} head(exch(x)) & = & head(tail(x)) \\ head(tail(exch(x))) & = & head(x) \\ tail(tail(exch(x))) & = & exch(tail(tail(x))). \end{array}$$

Thus it is definable as a futumorphism:

$$exch = [\![\langle\, head \circ tail, c \circ \langle\, head, l \circ tail \circ tail \,\rangle \,\rangle]\!].$$

$\square$

4.4 Histo- and futumorphisms in Haskell

Histomorphisms map arrows $\mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$ to arrows $\mu\mathsf{F} \to C$. Hence, in order to implement histomorphisms in Haskell, we first have to define the base functor for the "course-of-values" codata structure:

```
> newtype ProdF f a x = ProdF (a, f x)

> instance Functor f => Functor (ProdF f a) where
>    fmap f (ProdF (a, fx)) = ProdF (a, fmap f fx)
```

We also define the pairing function for `ProdF`:

```
> forkF :: (a -> b) -> (a -> f c) -> a
>                                   -> ProdF f b c
> forkF f g = ProdF . fork f g
```

In order to ease the navigation on the "course-of-values" codata structure, we define destructor functions out of it:

```
> hdCV :: Nu (ProdF f c) -> c
> hdCV xs = case out xs of
>               ProdF (c, _)  -> c

> tlCV :: Nu (ProdF f c) -> f (Nu (ProdF f c))
> tlCV xs = case out xs of
>               ProdF (_, fx) -> fx
```

Now, the type of histomorphism combinator can be expressed in Haskell as follows:

```
> histo :: Functor f => (f (Nu (ProdF f c)) -> c)
>                      ->  Mu f              -> c
```

Like in the case of paramorphisms, we have two possibilities for the defining equation of histo combinator. First, we can define it in terms of catamorphism:

```
> histo phi = hdCV . cata (Wrap . forkF phi id)
```

The second possibility is to use the directly recursive definition:

```
histo phi = phi
          . fmap (ana (forkF (histo phi) unIn))
          . unIn
```

This time, however, the first definition is more efficient. In the case of directly recursive definition, "course-of-value" codata structure is recomputed in every step of iteration. On the other hand, the catamorphic definition computes the "course-of-value" codata structure incrementally in a bottom-up fashion, thus effectively implementing the memoization of the values on previous arguments.

Example 4.3 (Fibonacci)
The Fibonacci function can be implemented as histomorphism (for greater clarity we use Haskell integers `Int` as the result type):

```
> fibo :: Nat -> Int
> fibo = histo phi
>     where phi Z     = 1
>           phi (S x) = case tlCV x of
>                         Z   -> 1
>                         S y -> hdCV x + hdCV y
```

□

Example 4.4 (evens)
The function `evens` takes from the given list every second element. It can be defined as histomorphism:

```
> evens :: List a -> List a
> evens = histo phi
>     where phi  N      = nilL
>           phi (C _ x) = case tlCV x of
>                           N     -> nilL
>                           C a y -> consL a (hdCV y)
```

□

Futumorphisms map arrows $C \to \mathsf{F}(\mathsf{F}^{\mu}(C))$ to arrows $C \to \nu\mathsf{F}$. Hence, in order to implement futumorphisms in Haskell, we first have to define the base functor for the inductive data structure $\mathsf{F}^{\mu}(C)$:

```
> newtype SumF f a x = SumF (Sum a (f x))

> instance Functor f => Functor (SumF f a) where
>     fmap f (SumF (InL a)) = SumF (InL a)
>     fmap f (SumF (InR x)) = SumF (InR (fmap f x))

> joinF :: (a -> c) -> (f b -> c) -> SumF f a b -> c
> joinF f g (SumF s) = join f g s
```

We also define constructor functions for the inductive data structure:

```
> lastF :: c -> Mu (SumF f c)
> lastF x = In (SumF (InL x))

> consF :: f (Mu (SumF f c)) -> Mu (SumF f c)
> consF x = In (SumF (InR x))
```

Now, the type of futumorphism combinator can be expressed in Haskell as follows:

```
> futu :: Functor f => (c -> f (Mu (SumF f c)))
>                    ->  c -> Nu f
```

Like in the case of histomorphisms, we have two possibilities for the defining equation of futumorphisms combinator. First, we can define it in terms of anamorphism:

```
futu phi = ana (joinF phi id . unIn) . lastF
```

The second possibility is to use the directly recursive definition obtained from the cancellation law:

```
> futu phi = Wrap
>          . fmap (cata (joinF (futu phi) Wrap))
>          . phi
```

There is no difference between two definitions in terms of efficiency except some small constant factor.

Example 4.5 (exchange)
The function *exch* from the example 4.2 can be implemented as follows:

```
> exch :: Stream a -> Stream a
> exch = futu phi
>    where phi xs = St (headS (tailS xs))
>                      (consF (St (headS xs)
>                                 (lastF (tailS xs))))
```

□

4.5 Related work

We do not know any other directly comparable work on course-of-value iteration or coiteration (except our own work in a type-theoretic setting [UV97, UV00b, Uus98]). The closest is work by Hu, Iwasaki and others [HITT96] about the tupling transformation. They develop calculational rules to eliminate multiple data traversals on functions defined by course-of-value iteration (and also by mutual recursion). Instead of using coinductive data structure to represent the course-of-values, they are using finite products which essentially are the unfolded finite prefixes of the course-of-values the function actually uses. This makes the rules quite hard to follow, but their aim is to use these rules in some automatic program transformation system, and not in programming itself.

CHAPTER 5

MENDLER-STYLE INDUCTIVE TYPES

This chapter is based on [UV99a] and here we consider a novel alternative approach to inductive types in the categorical setting, inspired from the work by N. P. Mendler [Men91] in type theory. The basic motivation for this another formalization lies in the difficulties of extending the traditional approach to inductive types (and coinductive types) for non-covariant base functors. Freyd's dialgebras [Fre90, Fre91] solve the problem for mixed-variant functors, but at the cost that the distinction between inductive and coinductive types vanishes.

One reason for the difficulties in the conventional approach is that the definition of homomorphism between F-algebras explicitly mentions the arrow mapping part of the functor. As a result, if F is not a covariant functor, the definition of homomorphisms has to be changed accordingly, otherwise the distributivity equation the homomorphism must satisfy is incorrectly typed.

The basic idea of so called Mendler-style inductive types is to modify the definition of algebra and their homomorphisms in such a way that the arrow mapping part of the functor does not manifest itself in the distributivity equation. Instead, there is an additional condition that the algebra itself has to satisfy, and functor appears only in the typing. Then the concept can be extended to apply to non-covariant bases by modifying the condition in the definition of algebra, but leaving the definition of algebra homomorphism (and so also the calculational laws) intact.

5.1 Mendler-style inductive types: covariant case

Recall, that for a given object C of the category $\mathcal{C}$, we can form a contravariant homfunctor $\mathcal{C}(-, C) : \mathcal{C}^{\mathrm{op}} \to \mathcal{S}et$ which takes an object A to the hom-set $\mathcal{C}(A, C)$, and an arrow $g : A \to B$ to the function $\mathcal{C}(g, C) : \mathcal{C}(B, C) \to \mathcal{C}(A, C)$

defined by $\lambda\beta : B \to C. \beta \circ g$. Similarly, if $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ is an endofunctor, we can define a contravariant functor $\mathcal{C}(\mathsf{F}(-), C) : \mathcal{C}^{\mathrm{op}} \to \mathcal{S}et$ which takes any object A to the hom-set $\mathcal{C}(\mathsf{F}(A), C)$, and any arrow $g : A \to B$ to the function $\mathcal{C}(\mathsf{F}(g), C) : \mathcal{C}(\mathsf{F}(B), C) \to \mathcal{C}(\mathsf{F}(A), C)$ defined by $\lambda\beta : B \to C. \beta \circ \mathsf{F}(g)$. In the following we denote the functor $\mathcal{C}(\mathsf{F}(-), C)$ by F/C.

Definition 5.1 (Mendler-style algebra for a functor)
Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor. A *Mendler-style* F-*algebra* or F-*malgebra* is a pair (C, Φ), where C is an object of $\mathcal{C}$ and $\Phi : \mathsf{Id}/C \dot{\to} \mathsf{F}/C$ is a natural transformation; i.e. for any arrow $g : A \to B$ the following diagram commutes:

$$\begin{tikzcd}
\mathcal{C}(B, C) \arrow[r, "\Phi_B"] \arrow[d, "\mathcal{C}(g, C)"'] & \mathcal{C}(\mathsf{F}(B), C) \arrow[d, "\mathcal{C}(\mathsf{F}(g), C)"] \\
\mathcal{C}(A, C) \arrow[r, "\Phi_A"'] & \mathcal{C}(\mathsf{F}(A), C)
\end{tikzcd}$$

$\square$

In other words, Φ is an operation on arrows with target C which “lifts” the source of the arrow under the functor F, by taking any arrow $\alpha : A \to C$ to the arrow $\Phi_A(\alpha) : \mathsf{F}(A) \to C$. If the lifted arrow $\alpha : C \to C$ is an automorphism (i.e. an arrow with the same source and target object), then $\Phi_C(\alpha) : \mathsf{F}(C) \to C$ is a conventional F-algebra. In particular, Φ takes the identity arrow on C to a conventional F-algebra $\Phi_C(\mathrm{id})$.

The naturality condition says, that the lifting preserves compositions in the following sense: if $\alpha = \beta \circ g$ for some object B and arrows $\beta : B \to C$, $g : A \to B$, then

$$\Phi_A(\beta \circ g) \;=\; \Phi_B(\beta) \circ \mathsf{F}(g) \tag{5.1}$$

or diagrammatically

$$\begin{tikzcd}
A \arrow[rr, "g"] \arrow[dr, "\alpha"'] & & B \arrow[dl, "\beta"] \\
& C &
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
\mathsf{F}(A) \arrow[rr, "\mathsf{F}(g)"] \arrow[dr, "\Phi_A(\alpha)"'] & & \mathsf{F}(B) \arrow[dl, "\Phi_B(\beta)"] \\
& C &
\end{tikzcd}$$

In particular, by taking $B = C$ and $\beta = \mathrm{id}_C$, we have that

$$\Phi_A(\alpha) \;=\; \Phi_C(\mathrm{id}) \circ \mathsf{F}(\alpha) \tag{5.2}$$

So, the lifting on the arrows is determined by the composition the functor with some F-algebra.

Definition 5.2 (malgebra homomorphism)
Let (C, Φ) and (D, Ψ) be F-malgebras. A homomorphism from (C, Φ) to (D, Ψ) is an arrow $h : C \to D$ such that for any object A the following diagram commutes in $\mathcal{S}et$:

$$\begin{tikzcd}
\mathcal{C}(A, C) \arrow[r, "\mathcal{C}(A,h)"] \arrow[d, "\Phi_A"'] & \mathcal{C}(A, D) \arrow[d, "\Psi_A"] \\
\mathcal{C}(\mathsf{F}(A), C) \arrow[r, "\mathcal{C}(\mathsf{F}(A),h)"'] & \mathcal{C}(\mathsf{F}(A), D)
\end{tikzcd}$$

□

In terms of base category $\mathcal{C}$, the square above tells that for any object A and arrow $\gamma : A \to C$, the following equation holds:

$$h \circ \Phi_A(\gamma) \;=\; \Psi_A(h \circ \gamma) \tag{5.3}$$

or diagrammatically:

$$\begin{tikzcd}
 & A \arrow[dl, "\gamma"'] \arrow[dr, "\delta"] & \\
C \arrow[rr, "h"'] & & D
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
 & \mathsf{F}(A) \arrow[dl, "\Phi_A(\gamma)"'] \arrow[dr, "\Psi_A(\delta)"] & \\
C \arrow[rr, "h"'] & & D
\end{tikzcd}$$

In particular, if we take $A = C$ and $\alpha = \mathrm{id}_C$, then

$$h \circ \Phi_C(\mathrm{id}) \;=\; \Psi_C(h). \tag{5.4}$$

Now, using the equality 5.2 about malgebras we get

$$h \circ \Phi_C(\mathrm{id}) \;=\; \Psi_C(\mathrm{id}) \circ \mathsf{F}(h). \tag{5.5}$$

Thus, homomorphism h is also homomorphism between conventional F-algebras $\Phi_C(\mathrm{id})$ and $\Psi_C(\mathrm{id})$.

Obviously, homomorphisms between malgebras compose, and the identity arrow on the carrier object gives the identity homomorphism. So, we can form a category $\mathcal{A}lg(\mathsf{F})^{\mathrm{m}}$ of Mendler-style F-algebras and their homomorphisms.

Definition 5.3 (initial malgebra for a functor)
A Mendler-style F-algebra $(\mu^{\mathrm{m}}\mathsf{F}, \mathrm{in}^{\mathrm{m}})$ is an *initial* F-*malgebra* if for any Mendler-style F-algebra (C, Φ) there exists an arrow $(\!|\, \Phi \,|\!)^{\mathrm{m}} : \mu^{\mathrm{m}}\mathsf{F} \to C$ satisfying the universal property:

$$f \circ \mathrm{in}^{\mathrm{m}}_{\mu^{\mathrm{m}}\mathsf{F}}(\mathrm{id}) = \Phi_{\mu^{\mathrm{m}}\mathsf{F}}(f) \quad \equiv \quad f = (\!|\, \Phi \,|\!)^{\mathrm{m}} \qquad \text{cataM-CHARN}$$

□

In other words, the initial malgebra $(\mu^{\mathrm{m}}\mathsf{F}, \mathrm{in}^{\mathrm{m}})$ is the initial object in the category $\mathcal{A}lg(\mathsf{F})^{\mathrm{m}}$. The cancellation, reflection and fusion laws for Mendler-style catamorphisms specialize as follows:

Corollary 5.1 *Let $(\mu^{\mathrm{m}}\mathsf{F}, \mathrm{in}^{\mathrm{m}})$ be an initial F-malgebra.*

- ***Cancellation:*** *For any F-malgebra (C, Φ)*

$$(\!|\, \Phi \,|\!)^{\mathrm{m}} \circ \mathrm{in}^{\mathrm{m}}_{\mu^{\mathrm{m}}\mathsf{F}}(\mathrm{id}) = \Phi_{\mu^{\mathrm{m}}\mathsf{F}}((\!|\, \Phi \,|\!)^{\mathrm{m}}) \qquad \text{cataM-SELF}$$

- ***Reflection:***

$$\mathrm{id} = (\!|\, \mathrm{in}^{\mathrm{m}} \,|\!)^{\mathrm{m}} \qquad \text{cataM-REFL}$$

- ***Fusion:*** *For any F-malgebras (C, Φ) and (D, Ψ) and an arrow $f : C \to D$*

$$f \circ \Phi_C(\mathrm{id}) = \Psi_C(f) \quad \Rightarrow \quad f \circ (\!|\, \Phi \,|\!)^{\mathrm{m}} = (\!|\, \Psi \,|\!)^{\mathrm{m}} \qquad \text{cataM-FUSION}$$

□

Note that neither the universal property nor the laws derived from it contain any direct references to the functor F. The functor only appears implicitly on the typing, and as the naturality condition for Mendler-style algebras involved.

Example 5.1 (naturals)
Consider the data type of natural numbers Nat. Recall, that it forms the initial algebra $(Nat, [\,zero, succ\,])$ for the functor $\mathsf{N}(X) = 1 + X$. For any object A and arrow $\gamma : A \to Nat$, define a mapping $\mathrm{in}^{\mathrm{m}}_A(\gamma) = [\,zero, succ \circ \gamma\,]$. Then $(Nat, \mathrm{in}^{\mathrm{m}})$ forms an initial N-malgebra.

For instance, the sum of two naturals can be defined in terms of Mendler-style catamorphism as follows:

$$add(n, m) \ = \ (\!|\, \lambda A, \gamma : A \to Nat.\, [\,\lambda x.m, succ \circ \gamma\,] \,|\!)^{\mathrm{m}}(n).$$

□

5.2 Conventional inductive types reduced to Mendler-style inductive types

The project of this section is to show that conventional inductive types reduce to Mendler-style inductive types. To this end, we prove that, for any endofunctor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$, the categories $\mathcal{A}lg(\mathsf{F})^{\mathrm{m}}$ and $\mathcal{A}lg(\mathsf{F})$ are isomorphic. The proof we present is a proof from scratch. For a reader versed in category theory, the result is a consequence from the Yoneda lemma.

Definition 5.4 (malgebra to algebra mapping)
For any F-malgebra (C, Φ), define

$$\llcorner \Phi \lrcorner \;=\; \Phi_C(\mathrm{id})$$

$\square$

Definition 5.5 (algebra to malgebra mapping)
For any conventional F-algebra (C, φ), define

$$\ulcorner \varphi \urcorner \;=\; \lambda A, \gamma : A \to C.\, \varphi \circ \mathsf{F}(\gamma)$$

$\square$

Proposition 5.2 *If (C, Φ) is a F-malgebra, then $(C, \llcorner \Phi \lrcorner)$ is a F-algebra.*

Proof. Trivial. $\square$

Proposition 5.3 *If (C, φ) is a F-algebra, then $(C, \ulcorner \varphi \urcorner)$ is a F-malgebra.*

Proof. We have to check that $\ulcorner \varphi \urcorner$ is a natural transformation.

$$\left\lceil\begin{array}{ll}
\rhd & \text{pick } A, B, g : A \to B, \beta : B \to C \\
\hline
 & \ulcorner \varphi \urcorner_A(\beta \circ g) \\
= & \qquad - \ulcorner - \urcorner\text{-def} - \\
 & \varphi \circ \mathsf{F}(\beta \circ g) \\
= & \qquad - F \text{ functor } - \\
 & \varphi \circ \mathsf{F}(\beta) \circ \mathsf{F}(g) \\
= & \qquad - \ulcorner - \urcorner\text{-def} - \\
 & \ulcorner \varphi \urcorner_B(\beta) \circ \mathsf{F}(g)
\end{array}\right.$$

$\square$

Proposition 5.4 *If (C, Φ) is a Mendler-style F-algebra, then*

$$\ulcorner \llcorner \Phi \lrcorner \urcorner = \Phi.$$

Proof.

$$\left\lceil\begin{array}{ll}
\rhd & \text{pick } A, \gamma : A \to C \\
\hline
 & \ulcorner \llcorner \Phi \lrcorner \urcorner_A(\gamma) \\
= & \qquad - \ulcorner - \urcorner\text{-def} - \\
 & \llcorner \Phi \lrcorner \circ F(\gamma) \\
= & \qquad - \llcorner - \lrcorner\text{-def} - \\
 & \Phi_C(\mathrm{id}) \circ F(\gamma) \\
= & \qquad - \Phi \text{ natural} - \\
 & \Phi_A(\gamma)
\end{array}\right.$$

$\square$

Proposition 5.5 *If (C, φ) is a conventional F-algebra, then*

$$\llcorner \ulcorner \varphi \urcorner \lrcorner = \varphi.$$

Proof.

$$\left[\begin{array}{ll} & \llcorner \ulcorner \varphi \urcorner \lrcorner \\ = & \quad - \llcorner - \lrcorner\text{-def} - \\ & \ulcorner \varphi \urcorner_C(\mathrm{id}) \\ = & \quad - \ulcorner - \urcorner\text{-def} - \\ & \varphi \circ F(\mathrm{id}) \\ = & \quad - F \text{ functorial} - \\ & \varphi \end{array}\right.$$

$\square$

Proposition 5.6 *If h is a Mendler-style F-algebra homomorphism between (C, Φ) and (D, Ψ), then h is also a conventional F-algebra homomorphism between $(C, \llcorner \Phi \lrcorner)$ and $(D, \llcorner \Psi \lrcorner)$.*

Proof. Already shown, see the equation 5.5 and the discussion before it. $\square$

Proposition 5.7 *If h is a conventional F-algebra homomorphism between (C, φ) and (D, ψ), then h is also a Mendler-style F-algebra homomorphism between $(C, \ulcorner \varphi \urcorner)$ and $(D, \ulcorner \psi \urcorner)$.*

Proof.

$$\left[\begin{array}{ll} \rhd & h \circ \varphi = \psi \circ \mathsf{F}(h) \\ \rhd & \text{pick } A, \gamma : A \to C \\ \hline & h \circ \ulcorner \varphi \urcorner_A(\gamma) \\ = & \quad - \ulcorner - \urcorner\text{-def} - \\ & h \circ \varphi \circ \mathsf{F}(\gamma) \\ = & \quad - \lhd - \\ & \psi \circ \mathsf{F}(h) \circ \mathsf{F}(\gamma) \\ = & \quad - \mathsf{F} \text{ functorial} - \\ & \psi \circ \mathsf{F}(h \circ \gamma) \\ = & \quad - \ulcorner - \urcorner\text{-def} - \\ & \ulcorner \psi \urcorner_A(h \circ \gamma) \end{array}\right.$$

$\square$

These propositions tell us that there exists a functor between the categories $\mathcal{A}lg(\mathsf{F})^{\mathrm{m}}$ and $\mathcal{A}lg(\mathsf{F})$ and a left-and-right inverse for it.

Theorem 5.8 *The categories $\mathcal{A}lg(\mathsf{F})^{\mathrm{m}}$ and $\mathcal{A}lg(\mathsf{F})$ are isomorphic.* □

The following is now immediate:

Corollary 5.9 *If $(\mu^{\mathrm{m}}\mathsf{F}, \mathrm{in}^{\mathrm{m}})$ is an initial Mendler-style F-algebra, then $(\mu^{\mathrm{m}}\mathsf{F}, \llcorner \mathrm{in}^{\mathrm{m}} \lrcorner)$ is an initial (conventional) F-algebra. For any F-algebra $\varphi : \mathsf{F}(C) \to C$, the unique homomorphism into it (i.e. catamorphism) is given by $\llparenthesis \ulcorner \varphi \urcorner \rrparenthesis^{\mathrm{m}} : \mu^{\mathrm{m}}\mathsf{F} \to C$.* □

Corollary 5.10 *If $(\mu\mathsf{F}, \mathrm{in})$ is an initial (conventional) F-algebra, then $(\mu\mathsf{F}, \ulcorner \mathrm{in} \urcorner)$ is an initial Mendler-style F-algebra. For any Mendler-style F-algebra (C, Φ), the unique homomorphism into it (i.e. Mendler-style catamorphism) is given by $\llparenthesis \llcorner \Phi \lrcorner \rrparenthesis : \mu\mathsf{F} \to C$.* □

5.3 Mendler-style inductive types: mixed variant case

The idea of Mendler-style inductive types makes sense not only for covariant base functors $\mathsf{F} : \mathcal{C} \to \mathcal{C}$, but also for mixed variant functors $\mathsf{G} : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$. The mixed variant case is, in fact, more general, as covariant functors are a degenerate case of mixed variant functors: for any $\mathsf{F} : \mathcal{C} \to \mathcal{C}$, one can trivially define $\mathsf{F}^{\wr} : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$, a padding of $\mathsf{F}^{\wr}$ with a "dummy" contravariant argument, by letting $\mathsf{F}^{\wr}(Y, X) = \mathsf{F}(X)$.

Definition 5.6 (Mendler-style algebra for a difunctor)
Let $\mathsf{G} : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$ be an endodifunctor. A *Mendler-style* G-*algebra* or G-*malgebra* is a pair (C, Φ), where C is an object of $\mathcal{C}$ and $\Phi : \mathsf{Id}^{\wr}/C \stackrel{\cdot}{\to} \mathsf{G}/C$ is a dinatural transformation; i.e. for any arrow $g : A \to B$ the following diagram commutes:

$$\begin{tikzcd}
 & \mathcal{C}(B,C) \arrow[dl, "\mathcal{C}(g,C)"'] \arrow[dr, equal] & \\
\mathcal{C}(A,C) \arrow[d, "\Phi_A"'] & & \mathcal{C}(B,C) \arrow[d, "\Phi_B"] \\
\mathcal{C}(\mathsf{G}(A,A),C) \arrow[dr, "\mathcal{C}(\mathsf{G}(g,\mathrm{id}_A),C)"'] & & \mathcal{C}(\mathsf{G}(B,B),C) \arrow[dl, "\mathcal{C}(\mathsf{G}(\mathrm{id}_B,g),C)"] \\
 & \mathcal{C}(\mathsf{G}(B,A),C) &
\end{tikzcd}$$

□

In terms of the base category, Φ is a mapping that takes any arrow $\gamma : A \to C$ to the arrow $\Phi_A(\alpha) : \mathsf{G}(A,A) \to C$ in such a way that if $\alpha = \beta \circ g$ for some object B and arrows $\beta : B \to C$, $g : A \to B$, then the following equation holds:

$$\Phi_A(\beta \circ g) \circ \mathsf{G}(g, \mathrm{id}_A) = \Phi_B(\beta) \circ \mathsf{G}(\mathrm{id}_B, g) \tag{5.6}$$

or diagrammatically:

$$\begin{tikzcd} & B \arrow[dd, "\beta"] \\ A \arrow[ur, "g"] \arrow[dr, "\alpha"'] & \\ & C \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} & \mathsf{G}(B,A) \arrow[dl, "\mathsf{G}(g,\mathrm{id}_A)"'] \arrow[dr, "\mathsf{G}(\mathrm{id}_B,g)"] & \\ \mathsf{G}(A,A) \arrow[dr, "\Phi_A(\alpha)"'] & & \mathsf{G}(B,B) \arrow[dl, "\Phi_B(\beta)"] \\ & C & \end{tikzcd}$$

If the contravariant argument of the difunctor G is “dummy” (i.e. $\mathsf{G}(X,Y) = \mathsf{F}(Y)$ for some covariant functor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$) then the dinaturality condition boils down to the naturality condition in definition 5.1, and equation 5.6 simplifies to equation 5.1. So in this case the definitions 5.1 and 5.6 coincide.

Definition 5.7 (malgebra homomorphism)
Let (C,Φ) and (D,Ψ) be G-malgebras for a difunctor $\mathsf{G} : \mathcal{C}^{\mathsf{op}} \times \mathcal{C} \to \mathcal{C}$. A homomorphism from (C,Φ) to (D,Ψ) is an arrow $h : C \to D$ such that for any object A the following diagram commutes in $\mathcal{S}et$:

$$\begin{tikzcd} \mathcal{C}(A,C) \arrow[r, "\mathcal{C}(A,h)"] \arrow[d, "\Phi_A"'] & \mathcal{C}(A,D) \arrow[d, "\Psi_A"] \\ \mathcal{C}(\mathsf{G}(A,A),C) \arrow[r, "\mathcal{C}(\mathsf{G}(A,A),h)"'] & \mathcal{C}(\mathsf{G}(A,A),D) \end{tikzcd}$$

□

In terms of base category $\mathcal{C}$, the square above tells that for any object A and arrow $\gamma : A \to C$, the following equation holds:

$$h \circ \Phi_A(\gamma) \;=\; \Psi_A(h \circ \gamma) \tag{5.7}$$

or diagrammatically:

$$\begin{tikzcd} & A \arrow[dl, "\gamma"'] \arrow[dr, "\delta"] & \\ C \arrow[rr, "h"'] & & D \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} & \mathsf{G}(A,A) \arrow[dl, "\Phi_A(\gamma)"'] \arrow[dr, "\Psi_A(\delta)"] & \\ C \arrow[rr, "h"'] & & D \end{tikzcd}$$

Note that the equation 5.7 looks exactly the same as the equation 5.3. The only difference between two is on the typing of Φ and Ψ.

Like in the covariant case, G-malgebras and their homomorphisms (for a difunctor G) form a category $\mathcal{A}lg(\mathsf{G})^{\mathrm{m}}$.

Definition 5.8 (initial malgebra for a difunctor)
A Mendler-style G-algebra $(\mu^{\mathrm{m}}\mathsf{G}, \mathrm{in}^{\mathrm{m}})$ is the *initial* G-*malgebra* if for any Mendler-style G-algebra (C, Φ) there exists a unique arrow $(\!|\, \Phi \,|\!)^{\mathrm{m}} : \mu^{\mathrm{m}}\mathsf{G} \to C$ satisfying the universal property:

$$f \circ \mathrm{in}^{\mathrm{m}}_{\mu^{\mathrm{m}}\mathsf{G}}(\mathrm{id}) = \Phi(f) \quad \equiv \quad f = (\!|\, \Phi \,|\!)^{\mathrm{m}} \qquad \text{cataM-C}{\small\text{HARN}}$$

In other words, the initial malgebra $(\mu^{\mathrm{m}}\mathsf{G}, \mathrm{in}^{\mathrm{m}})$ is an initial object in the category $\mathcal{A}lg(\mathsf{G})^{\mathrm{m}}$. The cancellation, reflection and fusion laws for Mendler-style catamorphisms specialize as follows:

Corollary 5.11 *Let* $(\mu^{\mathrm{m}}\mathsf{G}, \mathrm{in}^{\mathrm{m}})$ *be an initial* G-*malgebra.*

- ***Cancellation:*** *For any* G-*malgebra* (C, Φ)

$$(\!|\, \Phi \,|\!)^{\mathrm{m}} \circ \mathrm{in}^{\mathrm{m}}_{\mu^{\mathrm{m}}\mathsf{G}}(\mathrm{id}) = \Phi_{\mu^{\mathrm{m}}\mathsf{G}}((\!|\, \Phi \,|\!)^{\mathrm{m}}) \qquad \text{cataM-S}{\small\text{ELF}}$$

- ***Reflection:***

$$\mathrm{id} = (\!|\, \mathrm{in}^{\mathrm{m}} \,|\!)^{\mathrm{m}} \qquad \text{cataM-R}{\small\text{EFL}}$$

- ***Fusion:*** *For any* G-*malgebras* (C, Φ) *and* (D, Ψ) *and an arrow* $f : C \to D$

$$f \circ \Phi_C(\mathrm{id}) = \Psi_C(f) \quad \Rightarrow \quad f \circ (\!|\, \Phi \,|\!)^{\mathrm{m}} = (\!|\, \Psi \,|\!)^{\mathrm{m}} \qquad \text{cataM-F}{\small\text{USION}}$$

□

Note the fact that the arrow mapping part of the signature difunctor is not mentioned manifestly in the calculational laws for an initial Mendler-style algebra, it only appears in the dinaturality condition and this would in normal practice always be a "theorem for free" à la Wadler [Wad89].

Example 5.2 (course-of-value naturals)
Let $\mathsf{G}(Y, X) = [Y \to \mathsf{N}(X)] \times \mathsf{N}(X)$ and write Nat' for the carrier of the initial Mendler-style G-algebra $(\mu^{\mathrm{m}}\mathsf{G}, \mathrm{in}^{\mathrm{m}})$. Assume that there exists a predecessor function $pred' : Nat' \to 1 + Nat'$ which satisfies the following specification: for any object A and morphism $\gamma : A \to Nat'$

$$pred' \circ \mathrm{in}^{\mathrm{m}}(\gamma) = \mathsf{N}(\gamma) \circ \mathrm{snd}\,.$$

Then the functions $\mathit{zero}' : 1 \to \mathit{Nat}'$ and $\mathit{succ}' : \mathit{Nat}' \to \mathit{Nat}'$ can be defined as

$$
\begin{array}{rcl}
\mathit{zero}' & = & \mathrm{in}^{\mathrm{m}}_{\mathit{Nat}'}(\mathrm{id}) \circ \langle\, \lambda x.\, \mathit{pred}', \mathrm{inl} \,\rangle \\
\mathit{succ}' & = & \mathrm{in}^{\mathrm{m}}_{\mathit{Nat}'}(\mathrm{id}) \circ \langle\, \lambda x.\, \mathit{pred}', \mathrm{inr} \,\rangle.
\end{array}
$$

Now, the Fibonacci function can be defined as a Mendler-style catamorphism $\mathit{fibo} = (\!|\, \Phi \,|\!)^{\mathrm{m}} : \mathit{Nat}' \to \mathit{Nat}$, where

$$
\begin{array}{rcl}
\Phi_A(\gamma : A \to \mathit{Nat})(p, \mathrm{inl}()) & = & \mathit{one}() \\
\Phi_A(\gamma : A \to \mathit{Nat})(p, \mathrm{inr}(n)) & = & [\, \mathit{one}, \lambda n'.\, \mathit{add}(\gamma(n), \gamma(n')) \,](p(n))
\end{array}
$$

$\square$

5.4 Restricted existential types

The project opposite to that of the Section 5.2 — reducing Mendler-style inductive types to conventional inductive types — is unperformable in general. But, as we will see in Section 5.5, it can be carried out, if certain restricted existential types are available. Let us explain what these are.

Definition 5.9 (restricted cowedge)
Let $\mathsf{G} : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{C}$ be an endodifunctor and $\mathsf{H} : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{S}et$ a difunctor to $\mathcal{S}et$. An H-*restricted* G-*cowedge* (*cowedge from* G) is a pair (C, Φ) formed of an object C of $\mathcal{C}$ and dinatural transformation Φ between the difunctors H and G/C, i.e., a family of functions $\{\Phi_A\}_{A \in \mathcal{C}}$ between the sets $\mathsf{H}(A, A)$ and $\mathcal{C}(\mathsf{G}(A, A), C)$ indexed over objects of $\mathcal{C}$ such that, for any arrow $g : A \to B$ the following diagram commutes:

$$
\begin{tikzcd}
& \mathsf{H}(B,A) \arrow[dl, "\mathsf{H}(g,\mathrm{id}_A)"'] \arrow[dr, "\mathsf{H}(\mathrm{id}_B,g)"] & \\
\mathsf{H}(A,A) \arrow[d, "\Phi_A"'] & & \mathsf{H}(B,B) \arrow[d, "\Phi_B"] \\
\mathcal{C}(\mathsf{G}(A,A),C) \arrow[dr, "\mathcal{C}(\mathsf{G}(g,\mathrm{id}_A),C)"'] & & \mathcal{C}(\mathsf{G}(B,B),C) \arrow[dl, "\mathcal{C}(\mathsf{G}(\mathrm{id}_B,g),C)"] \\
& \mathcal{C}(\mathsf{G}(B,A),C) &
\end{tikzcd}
$$

$\square$

In other words, Φ is a function that takes objects A of $\mathcal{C}$ to functions Φ_A sending elements a of $\mathsf{H}(A,A)$ to morphisms $\Phi_A(a) : \mathsf{G}(A,A) \to C$ so that the following condition is met: for any objects A, B and morphism $g : A \to B$ of $\mathcal{C}$ and any element c of $\mathsf{H}(B,A)$, it holds in $\mathcal{C}$ that

$$\Phi_A(\mathsf{H}(g, \mathrm{id}_A)c) \circ \mathsf{G}(g, \mathrm{id}_A) = \Phi_B(\mathsf{H}(\mathrm{id}_B, g)c) \circ \mathsf{G}(\mathrm{id}_B, g)$$

or diagrammatically

$$\begin{tikzcd}
\mathsf{G}(B,A) \arrow[r, "\mathsf{G}(\mathrm{id}_B, g)"] \arrow[d, "\mathsf{G}(g, \mathrm{id}_A)"'] & \mathsf{G}(B,B) \arrow[d, "\Phi_B\,(\mathsf{H}(\mathrm{id}_B, g)c)"] \\
\mathsf{G}(A,A) \arrow[r, "\Phi_A\,(\mathsf{H}(g, \mathrm{id}_A)c)"'] & C
\end{tikzcd}$$

Definition 5.10 (restricted cowedge homomorphism)
An H-*restricted* G-*cowedge homomorphism* between H-restricted G-cowedges (C, Φ) and (D, Ψ) is an arrow $h : C \to D$ of $\mathcal{C}$ with the property that, for any object A of $\mathcal{C}$, it holds in $\mathcal{S}et$ that

$$\mathcal{C}(\mathsf{G}(A,A), h) \circ \Phi_A = \Psi_A$$

or diagrammatically

$$\begin{tikzcd}
& \mathsf{H}(A,A) \arrow[dl, "\Phi_A"'] \arrow[dr, "\Psi_A"] & \\
\mathcal{C}(\mathsf{G}(A,A), C) \arrow[rr, "\mathcal{C}(\mathsf{G}(A,A), h)"'] & & \mathcal{C}(\mathsf{G}(A,A), D)
\end{tikzcd}$$

$\square$

This condition is equivalent to the following one: for any object A of $\mathcal{C}$ and any element a of $\mathsf{H}(A,A)$, it is the case in $\mathcal{C}$ that $h \circ \Phi_A(a) = \Psi_A(a)$.

$$\begin{tikzcd}
& \mathsf{G}(A,A) \arrow[dl, "\Phi_A(a)"'] \arrow[dr, "\Psi_A(a)"] & \\
C \arrow[rr, "h"'] & & D
\end{tikzcd}$$

The H-restricted G-cowedges and homomorphisms between them form a category, $\mathcal{C}ow^{\mathsf{H}}_{\mathsf{G}}$.

Definition 5.11 (restricted coend)
An H-restricted G-cowedge $(\Sigma(\mathsf{H},\mathsf{G}), \mathrm{inj}^{\mathsf{H}}_{\mathsf{G}})$ is a H-*restricted* G-*coend* if it is an initial object of $\mathcal{C}ow^{\mathsf{H}}_{\mathsf{G}}$; i.e. for any H-restricted G-cowedge (C, Φ) there exists an unique arrow $f = [\,\Phi\,]^{\mathsf{H}}_{\mathsf{G}} : \Sigma(\mathsf{H},\mathsf{G}) \to C$ satisfying the universal property:

$$(\forall A, a \in \mathsf{H}(A,A).\, f \circ (\mathrm{inj}^{\mathsf{H}}_{\mathsf{G}})_A(a) = \Phi_A(a)) \quad \equiv \quad f = [\,\Phi\,]^{\mathsf{H}}_{\mathsf{G}} \quad \text{case-CHARN}$$

□

The cancellation, reflection and fusion laws for restricted coends specialize as follows:

Corollary 5.12 *Let* $(\Sigma(\mathsf{H},\mathsf{G}), \mathrm{inj}^{\mathsf{H}}_{\mathsf{G}})$ *be a* H-*restricted* G-*coend.*

- ***Cancellation:*** *For any* H-*restricted* G-*cowedge* (C, Φ)

$$\forall A, a \in \mathsf{H}(A,A).\, [\,\Phi\,]^{\mathsf{H}}_{\mathsf{G}} \circ (\mathrm{inj}^{\mathsf{H}}_{\mathsf{G}})_A(a) = \Phi_A(a) \qquad \text{case-SELF}$$

- ***Reflection:***

$$\mathrm{id}_{\Sigma(\mathsf{H},\mathsf{G})} = [\,\mathrm{inj}^{\mathsf{H}}_{\mathsf{G}}\,]^{\mathsf{H}}_{\mathsf{G}} \qquad \text{case-REFL}$$

- ***Fusion:*** *For any* H-*restricted* G-*cowedges* (C, Φ) *and* (D, Ψ) *and arrow* $h : C \to D$

$$(\forall A, a \in \mathsf{H}(A,A).\, h \circ \Phi_A(a) = \Psi_A(a)) \quad \Rightarrow \quad h \circ [\,\Phi\,]^{H}_{G} = [\,\Psi\,]^{H}_{G} \qquad \text{case-FUSION}$$

□

Example 5.3 (coends)
Consider the constant functor $\underline{1} : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{S}et$, which sends everything into a singleton set (i.e. an terminal object of $\mathcal{S}et$). Given an endodifunctor $\mathsf{G} : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{C}$, a pair (C, Φ) is a $\underline{1}$-restricted G-cowedge if Φ is a family of arrows $\Phi_A = \mathsf{G}(A,A) \to C$ which make the diagram

$$\begin{tikzcd}
\mathsf{G}(B,A) \arrow[r, "\mathsf{G}(\mathrm{id}_B, g)"] \arrow[d, "\mathsf{G}(g, \mathrm{id}_A)"'] & \mathsf{G}(B,B) \arrow[d, "\Phi_B"] \\
\mathsf{G}(A,A) \arrow[r, "\Phi_A"'] & C
\end{tikzcd}$$

commute for every $g : A \to B$. In other words, the pair (C, Φ) is a *cowedge of* G in ordinary sense[1] (i.e. Φ is a dinatural transformation from G to a constant functor $\underline{\mathsf{C}}$). Given two cowedges (C, Φ) and (D, Ψ), an arrow $h : C \to D$ is a homomorphism between them iff $h \circ \Phi_A = \Psi_A$ for any object A. Finally, a $\underline{1}$-restricted G-coend is the *coend of* G (see e.g. [Mac97], where the notation $\int^c \mathsf{G}(c,c)$ is used for $\Sigma(\underline{1}, \mathsf{G})$). □

5.5 Mendler-styles inductive types reduced to conventional inductive types

The necessary preparations made in the previous section, we are now in a position to construct a reduction of Mendler-style inductive types to conventional inductive types. We will obtain it in the same fashion as we obtained the reduction of conventional inductive types to Mendler-style inductive types in Section 5.2.

Let G be an endodifunctor on $\mathcal{C}$ such that, for any object C of $\mathcal{C}$, there exists a $\mathsf{Id}^\wr/C$-restricted G-coend $((\Sigma(\mathsf{Id}^\wr/C, \mathsf{G}), \mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^\wr/C}), [\,\cdot\,]_{\mathsf{G}}^{\mathsf{Id}^\wr/C})$. Then, we can define the following endofunction G^{e} on $\mathcal{C}$:

$$\begin{array}{rcl} \mathsf{G}^{\mathrm{e}} C &=& \Sigma(\mathsf{Id}^\wr/C, G) \\ \mathsf{G}^{\mathrm{e}}(h : C \to D) &=& [\,\lambda A, \gamma : A \to C.\, (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^\wr/D})_A(h \circ \gamma)\,]_{\mathsf{G}}^{\mathsf{Id}^\wr/C}. \end{array}$$

The function G^{e} turns out to be functorial (as one might expect), so G^{e} is an endofunctor on $\mathcal{C}$.

Definition 5.12
Given a Mendler-style G-algebra (C, Φ). Define

$$\llcorner \Phi \lrcorner = [\,\Phi\,]_{\mathsf{G}}^{\mathsf{Id}^\wr/C}$$

Definition 5.13
Given a conventional G^{e}-algebra (C, φ). Define

$$\ulcorner \varphi \urcorner \;=\; \lambda A, \gamma : A \to C.\, \varphi \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^\wr/C})_A(\gamma)$$

Proposition 5.13 *If* (C, φ) *is a conventional* G^{e}*-algebra, then* $(C, \ulcorner \varphi \urcorner)$ *is a Mendler-style* G*-algebra.*

[1] Mac Lane [Mac97] uses the term *wedge* for both, the dinatural transformations from and to constant functor. However, universal wedges are called ends and coends respectively, hence our use of the term cowedge

Proof. It has to be checked that $\ulcorner C, \varphi \urcorner$ is dinatural.

$$
\left[\begin{array}{ll}
\triangleright & \text{pick } A, B, g : A \to B, \beta : B \to C \\
\hline
& \ulcorner \varphi \urcorner_A(\beta \circ g) \circ \mathsf{G}(g, \mathrm{id}_A) \\
= & \quad \text{– } \ulcorner - \urcorner\text{-def –} \\
& \varphi \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^l/C})_A(\beta \circ g) \circ \mathsf{G}(g, \mathrm{id}_A) \\
= & \quad \text{– } \mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^l/C} \text{ dinatural –} \\
& \varphi \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^l/C})_B \beta \circ \mathsf{G}(\mathrm{id}_B, g) \\
= & \quad \text{– } \ulcorner - \urcorner\text{-def –} \\
& \ulcorner \varphi \urcorner_B(\beta) \circ \mathsf{G}(\mathrm{id}_B, g)
\end{array}\right.
$$

□

Proposition 5.14 *If (C, Φ) is a Mendler-style G-algebra, then $(C, \llcorner \Phi \lrcorner)$ is a conventional G^{e}-algebra.*

Proof. Trivial. □

Proposition 5.15 *If (C, φ) is a conventional G^{e}-algebra, then*

$$\llcorner \ulcorner \varphi \urcorner \lrcorner = \varphi.$$

Proof.

$$
\left[\begin{array}{ll}
& \llcorner \ulcorner \varphi \urcorner \lrcorner \\
= & \quad \text{– } \llcorner - \lrcorner\text{-def –} \\
& [\, \ulcorner \varphi \urcorner \,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad \text{– } \ulcorner - \urcorner\text{-def –} \\
& [\, \lambda A, \gamma : A \to C. \varphi \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^l/C})_A(\gamma) \,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad \text{– case fusion –} \\
& \varphi \circ [\, \mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^l/C} \,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad \text{– case reflection –} \\
& \varphi
\end{array}\right.
$$

□

Proposition 5.16 *If (C, Φ) is a Mendler-style G-algebra, then*

$$\ulcorner \llcorner \Phi \lrcorner \urcorner = \Phi.$$

Proof.

$$
\left[\begin{array}{ll}
\rhd & \text{pick } A, \gamma : A \to C \\
\hline
 & \ulcorner \llcorner \Phi \lrcorner \urcorner_A(\gamma) \\
= & \quad - \ulcorner - \urcorner\text{-def} - \\
 & \llcorner \Phi \lrcorner \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^{\wr}/C})_A(\gamma) \\
= & \quad - \llcorner - \lrcorner\text{-def} - \\
 & [\, \Phi \,]_{\mathsf{G}}^{\mathsf{Id}^{\wr}/C} \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^{\wr}/C})_A(\gamma) \\
= & \quad \text{– case cancellation –} \\
 & \Phi_A(\gamma)
\end{array}\right.
$$

□

Proposition 5.17 *If h is a conventional G^{e}-algebra homomorphism between (C, φ) and (D, ψ), then h is also a Mendler-style G-algebra homomorphism between $(C, \ulcorner \varphi \urcorner)$ and $(D, \ulcorner \psi \urcorner)$.*

Proof.

$$
\left[\begin{array}{ll}
\rhd & h \circ \varphi = \psi \circ \mathsf{G}^{\mathrm{e}} h \\
\rhd & \text{pick } A, \gamma : A \to C \\
\hline
 & h \circ \ulcorner \varphi \urcorner_A(\gamma) \\
= & \quad - \ulcorner - \urcorner\text{-def} - \\
 & h \circ \varphi \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^{\wr}/C})_A(\gamma) \\
= & \quad - \lhd - \\
 & \psi \circ \mathsf{G}^{\mathrm{e}} h \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^{\wr}/C})_A(\gamma) \\
= & \quad - \mathsf{G}^{\mathrm{e}}\text{-def} - \\
 & \psi \circ [\, \lambda A, \gamma : A \to C.\, (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^{\wr}/D})_A(h \circ \gamma) \,]_{\mathsf{G}}^{\mathsf{Id}^{\wr}/C} \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^{\wr}/C})_A(\gamma) \\
= & \quad \text{– case cancellation –} \\
 & \psi \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^{\wr}/D})_A(h \circ \gamma) \\
= & \quad - \ulcorner - \urcorner\text{-def} - \\
 & \ulcorner \psi \urcorner_A(h \circ \gamma)
\end{array}\right.
$$

□

Proposition 5.18 *If h is a Mendler-style G-algebra homomorphism between (C, Φ) and (D, Ψ), then h is also a conventional G^{e}-algebra homomorphism between $(C, \llcorner \Phi \lrcorner)$ and $(D, \llcorner \Psi \lrcorner)$.*

Proof.

$$
\left[\begin{array}{ll}
\rhd & \forall A, \gamma : A \to C.\, h \circ \Phi_A(\gamma) = \Psi_A(h \circ \gamma) \\
\hline
 & h \circ \llcorner \Phi \lrcorner \\
= & \quad - \llcorner - \lrcorner\text{-def} - \\
 & h \circ [\, \Phi\,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad - \text{case fusion} - \\
 & [\, \lambda A, \gamma : A \to C.\, h \circ \Phi_A(\gamma)\,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad - \lhd, \text{with } A := A, \gamma := \gamma - \\
 & [\, \lambda A, \gamma : A \to C.\, \Psi_A(h \circ \gamma)\,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad - \text{case cancellation} - \\
 & [\, \lambda A, \gamma : A \to C.\, [\, \Psi\,]_{\mathsf{G}}^{\mathsf{Id}^l/D} \circ (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^l/D})_A(h \circ \gamma)\,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad - \text{case fusion} - \\
 & [\, \Psi\,]_{\mathsf{G}}^{\mathsf{Id}^l/D} \circ [\, \lambda A, \gamma : A \to C.\, (\mathrm{inj}_{\mathsf{G}}^{\mathsf{Id}^l/D})_A(h \circ \gamma)\,]_{\mathsf{G}}^{\mathsf{Id}^l/C} \\
= & \quad - \mathsf{G}^{\mathrm{e}}\text{-def} - \\
 & [\, \Psi\,]_{\mathsf{G}}^{\mathsf{Id}^l/D} \circ \mathsf{G}^{\mathrm{e}}(h) \\
= & \quad - \llcorner - \lrcorner\text{-def} - \\
 & \llcorner \Psi \lrcorner \circ \mathsf{G}^{\mathrm{e}}(h)
\end{array}\right.
$$

□

These propositions tell us that there exists a functor between the categories $\mathcal{A}lg(\mathsf{G}^{\mathrm{e}})$ and $\mathcal{A}lg(\mathsf{G})^{\mathrm{m}}$ and a left-and-right inverse for it.

Theorem 5.19 *The categories $\mathcal{A}lg(\mathsf{G}^{\mathrm{e}})$ and $\mathcal{A}lg(\mathsf{G})^{\mathrm{m}}$ are isomorphic.* □

From here, the following is obvious already.

Corollary 5.20 *If $(\mu\mathsf{G}^{\mathrm{e}}, \mathrm{in})$ is an initial G^{e}-algebra, then $(\mu\mathsf{G}^{\mathrm{e}}, \ulcorner \mathrm{in} \urcorner)$ is an initial G-malgebra. For any G-malgebra (C, Φ), the catamorphism $(\!|\, \llcorner \Phi \lrcorner \,|\!) : \mu\mathsf{G}^{\mathrm{e}} \to C$ is the unique homomorphism from $(\mu\mathsf{G}^{\mathrm{e}}, \mathrm{in})$ to (C, Φ).* □

Corollary 5.21 *If $(\mu^{\mathrm{m}}\mathsf{G}, \mathrm{in}^{\mathrm{m}})$ is an initial Mendler-style G-algebra, then $(\mu^{\mathrm{m}}\mathsf{G}, \llcorner \mathrm{in}^{\mathrm{m}} \lrcorner)$ is an initial conventional G^{e}-algebra. For any conventional G^{e}-algebra (C, φ), the Mendler-style catamorphism $(\!|\, \ulcorner \varphi \urcorner \,|\!)^{\mathrm{m}} : \mu^{\mathrm{m}}\mathsf{G} \to C$ is the unique conventional homomorphism from $(\mu^{\mathrm{m}}\mathsf{G}, \mathrm{in}^{\mathrm{m}})$ to (C, φ).* □

5.6 Mendler-style inductive types in Haskell

Mendler-style inductive types can be modeled in Haskell by using existential types and rank-2 type signatures. While not part of the official Haskell98 language definition, several Haskell implementations (e.g. Hugs, ghc, hbc) support them as language extensions.

According to Corollary 5.20, an initial Mendler-style algebra for a difunctor G is an initial (conventional) G^{e}-algebra, where functor G^{e} is constructed from G by terms of certain restricted coends. Hence, in order to model Mendler-style inductive types, we first have to implement restricted coends.

The Haskell correspondent for a H-restricted G-cowedge (C, Φ) is a polymorphic function `phi :: H a -> G a -> C` (together with the type `C`), where `H` and `G` are type constructors. Thus, H-restricted G-coends can be implemented as follows:

```
> data RCoEnd h g = forall a . InjRCE (h a) (g a)
```

Given type constructors `h` and `g`, this defines the type `RCoEnd h g` as a pair of values of type `h a` and `g a` respectively. The type variable `a` is existentially quantified[2] and does not appear in `RCoEnd h g`. It also defines the data constructor `InjRCE :: h a -> g a -> RCoEnd g c` which corresponds to the restricted coend. The universal cowedge homomorphism out of `InjRCE` can be defined as follows:

```
> caseRCE :: (forall a . h a -> g a -> c)
                             -> RCoEnd h g -> c
> caseRCE phi (InjRCE ha ga) = phi ha ga
```

Note the use of rank 2 type signature to ensure that the first argument is a polymorphic function (i.e. is a restricted cowedge).

The type constructor corresponding to G^{e} can be defined by instantiating the first parameter of `RCoEnd` with a type constructor represented by `(->c)`. Unfortunately, Haskell does not allow sectioning of infix type constructors (as it does for "ordinary" infix operators). Hence, we have to define the corresponding type constructor explicitly.

```
> newtype Fun c a = Fun (a -> c)

> newtype Ext g c = Ext (RCoEnd (Fun c) g)
```

We also "lift" the definitions of restricted coends and universal cowedge homomorphisms for `Ext g c`.

```
> injExt :: (a -> c) -> g a -> Ext g c
> injExt h x = Ext (InjRCE (Fun h) x)
```

[2]The apparently counterintuitive use of `forall` to capture existentially quantified variables is justified by the logical equivalence $\forall A.P \Rightarrow Q \;\equiv\; (\exists A.P) \Rightarrow Q$, if A is not free in Q.

```
> caseExt :: (forall a . (a -> c) -> g a -> d)
>                              -> Ext g c -> d
> caseExt phi (Ext (InjRCE (Fun h) x)) = phi h x
```

The arrow mapping part of the functor G^e can be defined as follows:

```
> instance Functor (Ext g) where
>      fmap f = caseExt (\ h -> injExt (f . h))
```

Now, using the Corollary 5.20, we can define Mendler-style inductive types, initial malgebras and Mendler-style catamorphisms as follows:

```
> type MuM g = Mu (Ext g)

> inM :: (a -> MuM g) -> g a -> MuM g
> inM h x = In (injExt h x)

> cataM :: (forall a . (a -> c) -> g a -> c)
>                                -> MuM g -> c
> cataM phi = cata (caseExt phi)
```

Instead of going through conventional inductive types, we could also implement Mendler-style inductive types directly as fixed points of certain existential types.

```
data MuM g = forall a. InM (a -> MuM g) (g a)

cataM :: (forall a. (a -> c) -> g a -> c)
                             -> MuM g -> c
cataM phi (InM h x) = phi (cataM phi . h) x
```

In this case, according to Corollary 5.9, we could define conventional inductive types in terms of Mendler-style inductive types (only for type constructors which are functors).

```
type Mu f = MuM f

inMu :: f (Mu f) -> Mu f
inMu = InM id

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

It may be helpful to think about the existentially quantified type variable `a` as some (abstract) type of internal representations for the data type. Then, values of type `MuM g` are constructed from a function which converts internal representations to the data type together with the actual value itself, where the "outer" structure (given by type constructor `g`) is explicit but substructures are in the internal form. In particular, the definition of conventional inductive types is obtained by using the data type itself also for the internal representation.

Example 5.4 (naturals)
The Mendler-style definition of natural numbers involves the same type constructor `N` as the conventional definiton (see example 2.10).

```
> type NatM = MuM N
```

The constructor functions for naturals can be defined as follows:

```
> zeroNM :: NatM
> zeroNM  = inM id Z

> succNM  :: NatM -> NatM
> succNM n = inM id (S n)
```

The sum of two naturals can be defined in terms of Mendler-style catamorphism as follows:

```
> addNM :: NatM -> NatM -> NatM
> addNM x y = cataM phi x
>     where phi add_y Z     = y
>           phi add_y (S n) = succNM (add_y n)
```

□

Example 5.5 (course-of-value naturals)
Course-of-value naturals from example 5.2 can be implemented as follows:

```
> data N' x = N' (x -> N x) (N x)
> type NatCM = MuM N'
```

In order to define "standard" constructor functions, we first have to define the predecessor function for course-of-value naturals:

```
> predC :: NatCM -> N NatCM
> predC = caseExt phi . unIn
>    where phi h (N' _  Z)     = Z
>          phi h (N' _ (S n)) = S (h n)
```

Now, constructor functions can be defined as follows:

```
> zeroC  :: NatCM
> zeroC   = inM id (N' predC Z)

> succC  :: NatCM -> NatCM
> succC n = inM id (N' predC (S n))
```

The Fibonacci function from course-of-value naturals to integers can be defined as follows:

```
> fibC :: NatCM -> Int
> fibC  = cataM phi
>   where phi fib (N' p Z) = 1
>         phi fib (N' p (S n))
>                           = case p n of
>                               Z   -> fib n
>                               S m -> fib n + fib m
```

□

5.7 Related work

The concept of Mendler-style inductive type is an abstraction from N. P. Mendler's work [Men91] on an extension of system F (2nd-order simply-typed lambda-calculus) with inductive and coinductive types. This system supported iteration and coiteration through unusual operators whose beta-reduction rules did not mention the arrow mapping component of the base functor of the (co)inductive type. In [UV97, UV00b, Uus98, Mat98, Mat00], an observation was emphasized that the system does not loose any of its desirable meta-theoretic properties, if the base functor is permitted to be non-covariant. It was also shown how to interpret the liberalized system in lattice theory ($\mu\mathsf{F}$ is not necessarily of (pre-)fixed point of F, if F is non-monotonic). The same lattice theory explanations reappeared in [SU99]. The category-theoretic account given here is a "glorification" of the lattice-theoretic semantics.

CHAPTER 6

MENDLER-STYLE RECURSION SCHEMES

In this chapter we present an alternative formalization of recursion operators (for conventional inductive types) which is based on Mendler-style algebras. In particular, we develop Mendler-style operators for basic iteration, primitive recursion and course-of-value iteration. The new operators are equivalent to the corresponding conventional ones, but are somewhat more intuitive (at least in our opinion) against the background of "ordinary" (general-)recursive programming. This chapter is based on [UV00a].

In order to explain the difference between conventional and Mendler-style approach, consider the function $f : \mu\mathsf{F} \to C$ defined by simple iteration. The recursive defining equation for it is in the form

$$f \circ \mathrm{in} \;=\; \Phi(f),$$

where Φ is some definable function from arrows $\mu\mathsf{F} \to C$ to arrows $\mathsf{F}\mu\mathsf{F} \to C$. Just in this form, the equation does not necessarily define f iteratively. Indeed, the characterizing equations for primitive recursion and course-of-value iteration are exactly in the same form. In fact, the equation may have no solution in which case it does not define f at all.

The conventional method to ensure that the equation defines f by a simple iteration consists in insisting that $\Phi(f) = \varphi \circ \mathsf{F}(f)$, where $\varphi : \mathsf{F}(C) \to C$ is some F-algebra. This means imposing a relatively *syntactic* condition on the right-hand side of the equation: no expression other than '$\varphi \circ \mathsf{F}(f)$' is acceptable unless we are eager and able to prove that it equals $\varphi \circ \mathsf{F}(f)$ (which may require quite a bit of equational reasoning).

The Mendler-style method to ensure that the equation defines f by a simple iteration is leave the form of its right-hand side as it is (i.e. '$\Phi(f)$') but to require Φ not to use any specifics about the type $\mu\mathsf{F}$. This is achievable by insisting that Φ

is an instance of a function parametric in A from arrows of type $A \to C$ to arrows of type $\mathsf{F}(A) \to C$ (which is verifiable by type-checking). This means adopting a considerably more *semantic* approach to controlling the right-hand side of the equation.

6.1 Simple iteration

Mendler-style coding of the simple iteration follows directly from the properties of initial Mendler-style algebras for a (covariant) functor presented in Section 5.2. According to Theorem 5.8 and its corollaries any initial algebra determines an initial Mendler-style algebra and vice versa. Hence, we can take an initial algebra $(\mu F, \mathrm{in})$ and characterize Mendler-style homomorphisms out of the initial malgebra $(\mu F, \ulcorner \mathrm{in} \urcorner)$ directly in terms of it.

Definition 6.1 (m-catamorphism)
Let $(\mu\mathsf{F}, \mathrm{in})$ be an initial F-algebra. For any F-malgebra (C, Φ), a *m-catamorphism* $f = (\!| \Phi |\!)^{\mathrm{m}} : \mu\mathsf{F} \to C$ is a unique arrow satisfying the universal property

$$f \circ \mathrm{in} = \Phi_{\mu\mathsf{F}}(f) \quad \equiv \quad f = (\!| \Phi |\!)^{\mathrm{m}} \qquad \text{mcata-CHARN}$$

□

From this, the cancellation, reflection, and fusion laws for m-catamorphism follow straightforwardly.

Corollary 6.1 *Let* $(\mu\mathsf{F}, \mathrm{in})$ *be an initial* F*-algebra.*

- ***Cancellation:*** *For any* F*-malgebra* (C, Φ)

$$(\!| \Phi |\!)^{\mathrm{m}} \circ \mathrm{in} \;=\; \Phi_{\mu\mathsf{F}}((\!| \Phi |\!)^{\mathrm{m}}) \qquad \text{mcata-SELF}$$

- ***Reflection:***

$$\mathrm{id} \;=\; (\!| \lambda A, \gamma : A \to \mu F.\ \mathrm{in} \circ \mathsf{F}(\gamma) |\!)^{\mathrm{m}} \qquad \text{mcata-REFL}$$

- ***Fusion:*** *For any* F*-malgebras* (C, Φ) *and* (D, Ψ) *and an arrow* $f : C \to D$

$$(\forall A, \gamma : A \to C.\ f \circ \Phi_A(\gamma) = \Psi_A(f \circ \gamma)) \quad \Rightarrow \quad f \circ (\!| \Phi |\!)^{\mathrm{m}} = (\!| \Psi |\!)^{\mathrm{m}} \qquad \text{mcata-FUSION}$$

□

Note that the left-hand side of the fusion law is equivalent to the simpler equation $f \circ \Phi_{\mu\mathsf{F}}(\mathrm{id}) = \Psi_{\mu\mathsf{F}}(f)$. However, for calculational purposes the one in mcata-FUSION is preferable, as it can be directly instantiated in different contexts.

From the Corollary 5.10, we get the definition of m-catamorphism as conventional catamorphism. Similarly, the Corollary 5.9 gives to us the definition of conventional catamorphism as m-catamorphism.

Corollary 6.2 *Let (C, Φ) be a F-malgebra, then*

$$(\!|\, \Phi \,|\!)^{\mathrm{m}} = (\!|\, \Phi_{\mu\mathsf{F}}(\mathrm{id}) \,|\!) \qquad \text{mcata-DEF}$$

□

Corollary 6.3 *Let (C, φ) be a F-algebra, then*

$$(\!|\, \varphi \,|\!) = (\!|\, \lambda A, \gamma : A \to C.\, \varphi \circ \mathsf{F}(\gamma) \,|\!)^{\mathrm{m}} \qquad \text{mcata-CATA}$$

□

6.2 Primitive recursion

In this section we formalize the primitive recursion operator in the Mendler-style setting. For this, we first introduce the notions of rec-malgebra and their homomorphisms.

Definition 6.2 (rec-malgebra)
Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor for which there exists an initial algebra $(\mu\mathsf{F}, \mathrm{in})$. A F-*rec-malgebra* is a pair (C, Φ), where C is an object and $\Phi : \mathcal{C}(-, C \times \mu\mathsf{F}) \dot{\to} \mathcal{C}(\mathsf{F}(-), C)$ is a natural transformation; i.e. for any arrow $g : A \to B$ the following diagram commutes:

$$\begin{tikzcd}
\mathcal{C}(B, C \times \mu\mathsf{F}) \arrow[r, "\Phi_B"] \arrow[d, "\mathcal{C}(g, C \times \mathrm{id}_{\mu\mathsf{F}})"'] & \mathcal{C}(\mathsf{F}(B), C) \arrow[d, "\mathcal{C}(\mathsf{F}(g), C)"] \\
\mathcal{C}(A, C \times \mu\mathsf{F}) \arrow[r, "\Phi_A"'] & \mathcal{C}(\mathsf{F}(A), C)
\end{tikzcd}$$

□

In other words, Φ is a family of functions $\{\Phi_A\}_{A \in \mathcal{C}}$ which take arrows $\alpha : A \to C \times \mu\mathsf{F}$ to the arrows $\Phi_A(\alpha) : \mathsf{F}(A) \to C$. The naturality condition says, that Φ preserves compositions in the following sense: if $\alpha = \beta \circ g$ for some object B and arrows $\beta : B \to C \times \mu\mathsf{F}$, $g : A \to B$, then

$$\Phi_A(\beta \circ g) \;=\; \Phi_B(\beta) \circ \mathsf{F}(g) \tag{6.1}$$

or diagrammatically

$$\begin{tikzcd} A \arrow[rr, "g"] \arrow[dr, "\alpha"'] & & B \arrow[dl, "\beta"] \\ & C \times \mu\mathsf{F} & \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} \mathsf{F}(A) \arrow[rr, "\mathsf{F}(g)"] \arrow[dr, "\Phi_A(\alpha)"'] & & \mathsf{F}(B) \arrow[dl, "\Phi_B(\beta)"] \\ & C & \end{tikzcd}$$

In particular, by taking $B = C \times \mu\mathsf{F}$ and $\beta = \mathrm{id}_{C\times\mu\mathsf{F}}$, we get an equivalent condition:

$$\Phi_A(\alpha) \;=\; \Phi_{C\times\mu\mathsf{F}}(\mathrm{id}) \circ \mathsf{F}(\alpha) \tag{6.2}$$

Definition 6.3 (rec-malgebra homomorphism)
Let (C, Φ) and (D, Ψ) be two F-rec-malgebras. A *homomorphism* from (C, Φ) to (D, Ψ) is an arrow $h : C \to D$ in the category $\mathcal{C}$, such that for any object A the following diagram commutes in $\mathcal{S}et$:

$$\begin{tikzcd} \mathcal{C}(A, C \times \mu\mathsf{F}) \arrow[r, "\mathcal{C}(A, h\times\mathrm{id}_{\mu\mathsf{F}})"] \arrow[d, "\Phi_A"'] & \mathcal{C}(A, D \times \mu\mathsf{F}) \arrow[d, "\Psi_A"] \\ \mathcal{C}(\mathsf{F}(A), C) \arrow[r, "\mathcal{C}(\mathsf{F}(A), h)"'] & \mathcal{C}(\mathsf{F}(A), D) \end{tikzcd}$$

□

In terms of the base category, the square above tells that for any object A and arrow $\gamma : A \to C \times \mu\mathsf{F}$, the following equation holds:

$$h \circ \Phi_A(\gamma) \;=\; \Psi_A((h \times \mathrm{id}) \circ \gamma) \tag{6.3}$$

or diagrammatically:

$$\begin{tikzcd} & A \arrow[dl, "\gamma"'] \arrow[dr, "\delta"] & \\ C \times \mu\mathsf{F} \arrow[rr, "h\times\mathrm{id}_{\mu\mathsf{F}}"'] & & D \times \mu\mathsf{F} \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} & \mathsf{F}(A) \arrow[dl, "\Phi_A(\gamma)"'] \arrow[dr, "\Psi_A(\delta)"] & \\ C \arrow[rr, "h"'] & & D \end{tikzcd}$$

In particular, if we take $A = C \times \mu\mathsf{F}$ and $\gamma = \mathrm{id}_{C\times\mu\mathsf{F}}$, then

$$h \circ \Phi_C(\mathrm{id}) \;=\; \Psi_C(h \times \mathrm{id}). \tag{6.4}$$

Note that, in a cartesian closed base category, F-rec-malgebras and their homomorphisms are equivalent to ordinary Mendler-style algebras and homomorphisms for a difunctor $\mathsf{G}(Y, X) = [Y \to \mu\mathsf{F}] \times \mathsf{F}(X)$.

Definition 6.4 (m-paramorphism)
Let $(\mu\mathsf{F}, \mathrm{in})$ be an initial F-algebra. For any F-rec-malgebra (C, Φ), a *m-paramorphism* $f = (\!| \Phi |\!)^{\mathrm{m}} : \mu\mathsf{F} \to C$ is a unique arrow satisfying the universal property

$$f \circ \mathrm{in} = \Phi_{\mu\mathsf{F}}\langle f, \mathrm{id} \rangle \quad \equiv \quad f = (\!| \Phi |\!)^{\mathrm{m}} \qquad \text{mpara-CHARN}$$

□

Proposition 6.4 *Let* $(\mu\mathsf{F}, \mathrm{in})$ *be an initial* F*-algebra.*

- ***Cancellation:*** *For any* F*-rec-malgebra* (C, Φ)

$$(\!| \Phi |\!)^{\mathrm{m}} \circ \mathrm{in} = \Phi_{\mu\mathsf{F}}\langle (\!| \Phi |\!)^{\mathrm{m}}, \mathrm{id} \rangle \qquad \text{mpara-SELF}$$

- ***Reflection:***

$$\mathrm{id} = (\!| \lambda A, \gamma : A \to \mu\mathsf{F} \times \mu\mathsf{F}.\ \mathrm{in} \circ \mathsf{F}(\mathrm{fst} \circ \alpha) |\!)^{\mathrm{m}} \qquad \text{mpara-REFL}$$

- ***Fusion:*** *For any* F*-rec-malgebras* (C, Φ) *and* (D, Ψ) *and an arrow* $f : C \to D$

$$(\forall A, \gamma : A \to C \times \mu\mathsf{F}.\ f \circ \Phi_A(\gamma) = \Psi_A((f \times \mathrm{id}) \circ \gamma)) \\ \Rightarrow \quad f \circ (\!| \Phi |\!)^{\mathrm{m}} = (\!| \Psi |\!)^{\mathrm{m}} \\ \text{mpara-FUSION}$$

Proof. The cancellation law is directly obtained form the universal property of paramorphisms by substituting $f := (\!| \Phi |\!)^{\mathrm{m}}$ thus making the right-hand equation in mpara-CHARN trivially true. For the reflection law we argue:

$$
\left[\begin{array}{ll}
 & \mathrm{id} \\
= & \quad \text{– mpara-CHARN –} \\
 & \left[\begin{array}{ll} & \mathrm{id} \circ \mathrm{in} \\ = & \quad \text{– identity, F functor –} \\ & \mathrm{in} \circ \mathsf{F}(\mathrm{id}) \\ = & \quad \text{– pairing –} \\ & \mathrm{in} \circ \mathsf{F}(\mathrm{fst} \circ \langle \mathrm{id}, \mathrm{id} \rangle) \end{array}\right. \\
 & (\!| \lambda A, \gamma : A \to \mu\mathsf{F} \times \mu\mathsf{F}.\ \mathrm{in} \circ \mathsf{F}(\mathrm{fst} \circ \gamma) |\!)^{\mathrm{m}}
\end{array}\right.
$$

Finally, the fusion law is proved as follows:

$$
\begin{array}{ll}
\rhd & \forall A, \gamma : A \to C \times \mu\mathsf{F}.\ f \circ \Phi_A(\gamma) = \Psi_A((f \times \mathrm{id}) \circ \gamma) \\
\hline
& f \circ (\!|\, \Phi \,|\!)^{\mathrm{m}} \\
= & \quad \text{– mpara-CHARN –} \\
& \left[\begin{array}{ll}
& f \circ (\!|\, \Phi \,|\!)^{\mathrm{m}} \circ \mathrm{in} \\
= & \quad \text{– mpara-SELF –} \\
& f \circ \Phi_{\mu\mathsf{F}} \langle\, (\!|\, \Phi \,|\!)^{\mathrm{m}}, \mathrm{id} \,\rangle \\
= & \quad - \lhd - \\
& \Psi_{\mu\mathsf{F}}((f \times \mathrm{id}) \circ \langle\, (\!|\, \Phi \,|\!)^{\mathrm{m}}, \mathrm{id} \,\rangle) \\
= & \quad \text{– pairing –} \\
& \Psi_{\mu\mathsf{F}} \langle\, f \circ (\!|\, \Phi \,|\!)^{\mathrm{m}}, \mathrm{id} \,\rangle
\end{array}\right. \\
& (\!|\, \Psi \,|\!)^{\mathrm{m}}
\end{array}
$$

□

Proposition 6.5 *For any* F*-rec-malgebras* (C, Φ)

$$(\!|\, \Phi \,|\!)^{\mathrm{m}} = (\!|\, \Phi_{\mu\mathsf{F} \times \mu\mathsf{F}}(\mathrm{id}) \,|\!) \qquad \text{mpara-DEF}$$

Proof.

$$
\begin{array}{ll}
& (\!|\, \Phi_{\mu\mathsf{F} \times \mu\mathsf{F}}(\mathrm{id}) \,|\!) \\
= & \quad \text{– mpara-CHARN –} \\
& \left[\begin{array}{ll}
& (\!|\, \Phi_{\mu\mathsf{F} \times \mu\mathsf{F}}(\mathrm{id}) \,|\!) \circ \mathrm{in} \\
= & \quad \text{– para-SELF –} \\
& \Phi_{\mu\mathsf{F} \times \mu\mathsf{F}}(\mathrm{id}) \circ \mathsf{F} \,\langle\, (\!|\, \Phi_{\mu\mathsf{F} \times \mu\mathsf{F}}(\mathrm{id}) \,|\!), \mathrm{id} \,\rangle \\
= & \quad \text{– 6.2 –} \\
& \Phi_{\mu\mathsf{F}} \langle\, (\!|\, \Phi_{\mu\mathsf{F} \times \mu\mathsf{F}}(\mathrm{id}) \,|\!), \mathrm{id} \,\rangle
\end{array}\right. \\
& (\!|\, \Phi \,|\!)^{\mathrm{m}}
\end{array}
$$

□

Proposition 6.6 *For any arrow* $\varphi : \mathsf{F}(C \times \mu\mathsf{F}) \to C$

$$(\!|\, \varphi \,|\!) = (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\, \varphi \circ \mathsf{F}(\gamma) \,|\!)^{\mathrm{m}} \qquad \text{mpara-PARA}$$

Proof.

$$
\begin{array}{ll}
& (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\, \varphi \circ \mathsf{F}(\gamma) \,|\!)^{\mathrm{m}} \\
= & \quad \text{– para-CHARN –} \\
& \left[\begin{array}{ll}
& (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\, \varphi \circ \mathsf{F}(\gamma) \,|\!)^{\mathrm{m}} \circ \mathrm{in} \\
= & \quad \text{– mpara-SELF –} \\
& \varphi \circ \mathsf{F} \langle\, (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\, \varphi \circ \mathsf{F}(\gamma) \,|\!)^{\mathrm{m}}, \mathrm{id} \,\rangle
\end{array}\right. \\
& (\!|\, \varphi \,|\!)
\end{array}
$$

□

Proposition 6.7 *For any* F*-malgebra* (C, Φ)

$$(\!|\, \Phi \,|\!)^{\mathrm{m}} = (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\ \Phi_A(\mathrm{fst} \circ \gamma) \,|\!)^{\mathrm{m}} \qquad \text{mpara-MCATA}$$

Proof.

$$\begin{array}{ll}
 & (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\ \Phi_A(\mathrm{fst} \circ \gamma) \,|\!)^{\mathrm{m}} \\
= & \quad \text{– mcata-CHARN –} \\
 & \left[\begin{array}{ll}
 & (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\ \Phi_A(\mathrm{fst} \circ \gamma) \,|\!)^{\mathrm{m}} \circ \mathrm{in} \\
= & \quad \text{– mpara-SELF –} \\
 & \Phi_{\mu\mathsf{F}}(\mathrm{fst} \circ \langle\, (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\ \Phi_A(\mathrm{fst} \circ \gamma) \,|\!)^{\mathrm{m}}, \mathrm{id} \,\rangle) \\
= & \quad \text{– pairing –} \\
 & \Phi_{\mu\mathsf{F}} (\!|\, \lambda A, \gamma : A \to C \times \mu\mathsf{F}.\ \Phi_A(\mathrm{fst} \circ \gamma) \,|\!)^{\mathrm{m}}
\end{array}\right. \\
 & (\!|\, \Phi \,|\!)^{\mathrm{m}}
\end{array}$$

□

6.3 Course-of-value iteration

In this section we formalize the course-of-value iteration operator in the Mendler-style setting. We do it in the analogous way to the primitive recursion by introducing the notions of cv-malgebra and their homomorphisms. For this, we need Mulry's notion of strong dinaturality [Mul91].

Definition 6.5 (strong dinaturality)
Let $\mathsf{H}, \mathsf{G} : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{A}$ be difunctors. A *strong dinatural transformation* $\Phi : \mathsf{H} \dot{\to} \mathsf{G}$ is a family of maps Φ_A for all $A \in \mathcal{C}$, such that for every arrow $g : A \to B$ the following diagram commutes:

$$\begin{tikzcd}
 & W \arrow[dl] \arrow[dr] & \\
\mathsf{H}(A,A) \arrow[dr, "\mathsf{H}(\mathrm{id}_A, g)"] \arrow[dd, "\Phi_A"'] & & \mathsf{H}(B,B) \arrow[dl, "\mathsf{H}(g, \mathrm{id}_B)"'] \arrow[dd, "\Phi_B"] \\
 & \mathsf{H}(A,B) & \\
\mathsf{G}(A,A) \arrow[dr, "\mathsf{G}(\mathrm{id}_A, g)"] & & \mathsf{G}(B,B) \arrow[dl, "\mathsf{G}(g, \mathrm{id}_B)"'] \\
 & \mathsf{G}(A,B) &
\end{tikzcd}$$

where W is the pullback of $\mathsf{H}(\mathrm{id}_A, g)$ and $\mathsf{H}(g, \mathrm{id}_B)$. □

Proposition 6.8 ([Mul91]) *Every strong dinatural transformation $\Phi : \mathsf{H} \dot{\to} \mathsf{G}$ is a dinatural transformation.*

Proof. Since $\mathsf{H}(\mathrm{id}_A, g) \circ \mathsf{H}(g, \mathrm{id}_A) = \mathsf{H}(g, \mathrm{id}_B) \circ \mathsf{H}(\mathrm{id}_B, g)$, the pair of arrows $\mathsf{H}(g, \mathrm{id}_A)$ and $\mathsf{H}(\mathrm{id}_B, g)$ factors through W and thus $\mathsf{G}(\mathrm{id}_A, g) \circ \Phi_A \circ \mathsf{H}(g, \mathrm{id}_A) = \mathsf{G}(g, \mathrm{id}_B) \circ \Phi_B \circ \mathsf{H}(\mathrm{id}_B, g)$. ☐

Note that malgebras and rec-malgebras, which by definition are dinatural transformations, are also strong dinatural transformations, as the pullback squares for them are trivial.

Definition 6.6 (cv-malgebra)
Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be an endofunctor. A F-*cv-malgebra* is a pair (C, Φ), where C is an object and $\Phi : \mathcal{C}(-, C \times \mathsf{F}(=)) \dot{\to} \mathcal{C}(\mathsf{F}(-), C)$ is a strong dinatural transformation; i.e. for any arrow $g : A \to B$ the following diagram commutes:

$$\begin{tikzcd}
& & W \arrow[dll] \arrow[drr] & & \\
\mathcal{C}(A, C \times \mathsf{F}(A)) \arrow[drr, "\mathcal{C}(\mathrm{id}_A, \mathrm{id}_C \times \mathsf{F}(g))"] \arrow[dd, "\Phi_A"'] & & & & \mathcal{C}(B, C \times \mathsf{F}(B)) \arrow[dll, "\mathcal{C}(g, \mathrm{id}_{C \times \mathsf{F}(B)})"'] \arrow[dd, "\Phi_B"] \\
& & \mathcal{C}(A, C \times \mathsf{F}(B)) & & \\
\mathcal{C}(\mathsf{F}(A), C) \arrow[drr, equal] & & & & \mathcal{C}(\mathsf{F}(B), C) \arrow[dll, "\mathcal{C}(\mathsf{F}(g), C)"] \\
& & \mathcal{C}(\mathsf{F}(A), C) & &
\end{tikzcd}$$

where $W = \{(\alpha : A \to C \times \mathsf{F}(A), \beta : A \to C \times \mathsf{F}(A)) \mid (\mathrm{id}_C \times \mathsf{F}(g)) \circ \alpha = \beta \circ g\}$ is a pullback of $\mathcal{C}(\mathrm{id}_A, \mathrm{id}_C \times \mathsf{F}(g))$ and $\mathcal{C}(g, \mathrm{id}_{C \times \mathsf{F}(B)})$. ☐

In terms of the base category $\mathcal{C}$, Φ is a family of functions $\{\Phi_A\}_{A \in \mathcal{C}}$ which take $\mathsf{F}^{\times}_{C}$-coalgebras (see Definition 4.1) $\alpha : A \to C \times \mathsf{F}(A)$ to the arrows $\Phi_A(\alpha) : \mathsf{F}(A) \to C$. The strong dinaturality condition means, that for arbitrary $\mathsf{F}^{\times}_{C}$-coalgebras $\alpha : A \to C \times \mathsf{F}(A)$ and $\beta : B \to C \times \mathsf{F}(B)$, and an arrow $g : A \to B$, the following holds:

$$(\mathrm{id}_C \times \mathsf{F}(g)) \circ \alpha = \beta \circ g \quad \Rightarrow \quad \Phi_A(\alpha) = \Phi_B(\beta) \circ \mathsf{F}(g) \qquad (6.5)$$

or diagrammatically

$$
\begin{tikzcd}
A \arrow[r, "\alpha"] \arrow[d, "g"'] & C \times \mathsf{F}(A) \arrow[d, "\mathrm{id}_C \times \mathsf{F}(g)"] \\
B \arrow[r, "\beta"'] & C \times \mathsf{F}(B)
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
\mathsf{F}(A) \arrow[rr, "\mathsf{F}(g)"] \arrow[dr, "\Phi_A(\alpha)"'] & & \mathsf{F}(B) \arrow[dl, "\Phi_B(\beta)"] \\
& C &
\end{tikzcd}
$$

Note that the left-hand side of the implication says, that g is a homomorphism between $\mathsf{F}_C^{\times}$-coalgebras α and β.

Assume that there exists a terminal $\mathsf{F}_C^{\times}$-coalgebra $(\nu\mathsf{F}_C^{\times}, \mathrm{out})$. Then, by taking $B = \nu\mathsf{F}_C^{\times}$ and $\beta = \mathrm{out}$, the condition 6.5 simplifies to the following equation:

$$\Phi_A(\alpha) \;=\; \Phi_{\nu\mathsf{F}_C^{\times}}(\mathrm{out}) \circ \mathsf{F}(\llbracket \alpha \rrbracket) \tag{6.6}$$

As the calculation below shows, the equation is equivalent to the previous implication. Indeed, if equation 6.6 holds, then

$$
\begin{array}{ll}
\rhd & \text{pick } A, B, g : A \to B, \alpha : A \to C \times \mathsf{F}(A), \beta : B \to C \times \mathsf{F}(B) \\
\rhd & (\mathrm{id}_C \times \mathsf{F}(g)) \circ \alpha = \beta \circ g \\
\hline
 & \Phi_A(\alpha) \\
= & \quad - \text{6.6} - \\
 & \Phi_{\nu\mathsf{F}_C^{\times}}(\mathrm{out}) \circ \mathsf{F}(\llbracket \alpha \rrbracket) \\
= & \quad - \lhd, \text{ana-F}\textsc{usion} - \\
 & \Phi_{\nu\mathsf{F}_C^{\times}}(\mathrm{out}) \circ \mathsf{F}(\llbracket \beta \rrbracket \circ g) \\
= & \quad - \mathsf{F} \text{ functor} - \\
 & \Phi_{\nu\mathsf{F}_C^{\times}}(\mathrm{out}) \circ \mathsf{F}(\llbracket \beta \rrbracket) \circ \mathsf{F}(g) \\
= & \quad - \text{6.6} - \\
 & \Phi_B(\beta) \circ \mathsf{F}(g)
\end{array}
$$

Definition 6.7 (cv-malgebra homomorphism)
Let (C, Φ) and (D, Ψ) be two F-cv-malgebras. A *homomorphism* from (C, Φ) to (D, Ψ) is an arrow $h : C \to D$ in the category $\mathcal{C}$, such that for any object A the following diagram commutes in $\mathcal{S}et$:

$$
\begin{tikzcd}
\mathcal{C}(A, C \times \mathsf{F}(A)) \arrow[r, "\mathcal{C}(\mathrm{id}_A, h \times \mathrm{id}_{\mathsf{F}(A)})"] \arrow[d, "\Phi_A"'] & \mathcal{C}(A, D \times \mathsf{F}(A)) \arrow[d, "\Psi_A"] \\
\mathcal{C}(\mathsf{F}(A), C) \arrow[r, "\mathcal{C}(\mathrm{id}_{\mathsf{F}(A)}, h)"'] & \mathcal{C}(\mathsf{F}(A), D)
\end{tikzcd}
$$

□

In terms of the base category, the square above tells that for any object A and $\mathsf{F}^{\times}_{C}$-coalgebra $\gamma : A \to C \times \mathsf{F}(A)$, the following equation holds:

$$h \circ \Phi_A(\gamma) \;=\; \Psi_A((h \times \mathrm{id}) \circ \gamma) \tag{6.7}$$

or diagrammatically:

$$\begin{tikzcd}[column sep=small] & A \arrow[dl, "\gamma"'] \arrow[dr, "\delta"] & \\ C \times \mathsf{F}(A) \arrow[rr, "h \times \mathrm{id}_{\mathsf{F}(A)}"'] & & D \times \mathsf{F}(A) \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} & \mathsf{F}(A) \arrow[dl, "\Phi_A(\gamma)"'] \arrow[dr, "\Psi_A(\delta)"] & \\ C \arrow[rr, "h"'] & & D \end{tikzcd}$$

Assuming that there exists a terminal $\mathsf{F}^{\times}_{D}$-coalgebra $(\nu\mathsf{F}^{\times}_{D}, \mathrm{out})$, the condition above simplifies to the equivalent equation:

$$h \circ \Phi_A(\mathrm{out}) \;=\; \Psi_A((h \times \mathrm{id}) \circ \mathrm{out}) \tag{6.8}$$

Note that, in a cartesian closed base category, F-cv-malgebras and their homomorphisms are equivalent to ordinary Mendler-style algebras and homomorphisms for a difunctor $\mathsf{G}(Y, X) = [Y \to \mathsf{F}(X)] \times \mathsf{F}(X)$ (the instance of which we used in the example 5.2 for course-of-value naturals).

Definition 6.8 (m-histomorphism)
Let $(\mu\mathsf{F}, \mathrm{in})$ be an initial F-algebra. For any F-cv-malgebra (C, Φ), a *m-histomorphism* $f = \{\!|\, \Phi \,|\!\}^{\mathrm{m}} : \mu\mathsf{F} \to C$ is a unique arrow satisfying the universal property

$$f \circ \mathrm{in} = \Phi_{\mu\mathsf{F}}\langle\, f, \mathrm{in}^{-1} \,\rangle \quad \equiv \quad f = \{\!|\, \Phi \,|\!\}^{\mathrm{m}} \qquad \text{mhisto-CHARN}$$

□

Proposition 6.9 *Let $(\mu\mathsf{F}, \mathrm{in})$ be an initial F-algebra.*

- ***Cancellation:*** *For any F-cv-malgebra (C, Φ)*

$$\{\!|\, \Phi \,|\!\}^{\mathrm{m}} \circ \mathrm{in} = \Phi_{\mu\mathsf{F}}\langle\, \{\!|\, \Phi \,|\!\}^{\mathrm{m}}, \mathrm{in}^{-1} \,\rangle \qquad \text{mhisto-SELF}$$

- ***Reflection:***

$$\mathrm{id} = \{\!|\, \lambda A, \gamma : A \to \mu\mathsf{F} \times \mathsf{F}(A).\ \mathrm{in} \circ \mathsf{F}(\mathrm{fst} \circ \gamma) \,|\!\}^{\mathrm{m}} \qquad \text{mhisto-REFL}$$

- ***Fusion:*** *For any F-cv-malgebras (C, Φ) and (D, Ψ) and an arrow $f : C \to D$*

$$(\forall A, \gamma : A \to C \times \mathsf{F}(A).\ f \circ \Phi_A(\gamma) = \Psi_A((f \times \mathrm{id}) \circ \gamma)) \\ \Rightarrow \quad f \circ \{\!|\, \Phi \,|\!\}^{\mathrm{m}} = \{\!|\, \Psi \,|\!\}^{\mathrm{m}}$$

mhisto-FUSION

Proof. The cancellation law is directly obtained form the universal property. For the reflection law we argue:

$$
\left[\begin{array}{ll}
 & \mathrm{id} \\
= & \quad \text{– mhisto-}\textsc{Charn}\text{ –} \\
 & \left[\begin{array}{ll}
 & \mathrm{id} \circ \mathrm{in} \\
 = & \quad \text{– identity, } \mathsf{F} \text{ functor –} \\
 & \mathrm{in} \circ \mathsf{F}(\mathrm{id}) \\
 = & \quad \text{– pairing –} \\
 & \mathrm{in} \circ \mathsf{F}\,(\mathrm{fst} \circ \langle\, \mathrm{id}, \mathrm{in}^{-1} \,\rangle)
 \end{array}\right. \\
 & \llparenthesis\, \lambda A, \gamma : A \to \mu\mathsf{F} \times \mathsf{F}(A).\ \mathrm{in} \circ \mathsf{F}(\mathrm{fst} \circ \gamma) \,\rrparenthesis^{\mathrm{m}}
\end{array}\right.
$$

Finally, the fusion law is proved as follows:

$$
\left[\begin{array}{ll}
\rhd & \forall A, \gamma : A \to C \times \mathsf{F}(A).\ \ f \circ \Phi_A(\gamma) = \Psi_A((f \times \mathrm{id}) \circ \gamma) \\ \hline
 & f \circ \llparenthesis\, \Phi \,\rrparenthesis^{\mathrm{m}} \\
= & \quad \text{– mhisto-}\textsc{Charn}\text{ –} \\
 & \left[\begin{array}{ll}
 & f \circ \llparenthesis\, \Phi \,\rrparenthesis^{\mathrm{m}} \circ \mathrm{in} \\
 = & \quad \text{– mhisto-}\textsc{Self}\text{ –} \\
 & f \circ \Phi_{\mu\mathsf{F}}\langle\, \llparenthesis\, \Phi \,\rrparenthesis^{\mathrm{m}}, \mathrm{in}^{-1} \,\rangle \\
 = & \quad - \lhd - \\
 & \Psi_{\mu\mathsf{F}}\langle\, f \circ \llparenthesis\, \Phi \,\rrparenthesis^{\mathrm{m}}, \mathrm{in}^{-1} \,\rangle
 \end{array}\right. \\
 & \llparenthesis\, \Psi \,\rrparenthesis^{\mathrm{m}}
\end{array}\right.
$$

$\square$

If there exists a terminal $\mathsf{F}^{\times}_{D}$-coalgebra $(\nu\mathsf{F}^{\times}_{D}, \mathrm{out})$, then any m-histomorphism can be defined in terms of a (conventional) histomorphism, and vice versa.

Proposition 6.10 *Let* $(\nu\mathsf{F}^{\times}_{C}, \mathrm{out})$ *be a terminal* $\mathsf{F}^{\times}_{C}$*-coalgebra, then for any* F*-cv-malgebra* (C, Φ)

$$
\llparenthesis\, \Phi \,\rrparenthesis^{\mathrm{m}} = \llparenthesis\, \Phi_{\nu\mathsf{F}^{\times}_{C}}(\mathrm{out}) \,\rrparenthesis \qquad \text{mhisto-}\textsc{Def}
$$

Proof.

$$
\left[\begin{array}{ll}
 & \llparenthesis\, \Phi_{\nu\mathsf{F}^{\times}_{C}}(\mathrm{out}) \,\rrparenthesis \\
= & \quad \text{– mhisto-}\textsc{Charn}\text{ –} \\
 & \left[\begin{array}{ll}
 & \llparenthesis\, \Phi_{\nu\mathsf{F}^{\times}_{C}}(\mathrm{out}) \,\rrparenthesis \circ \mathrm{in} \\
 = & \quad \text{– histo-}\textsc{Self}\text{ –} \\
 & \Phi_{\nu\mathsf{F}^{\times}_{C}}(\mathrm{out}) \circ \mathsf{F}\,[\!(\, \langle\, \llparenthesis\, \Phi_{\nu\mathsf{F}^{\times}_{C}}(\mathrm{out}) \,\rrparenthesis, \mathrm{in}^{-1} \,\rangle \,)\!] \\
 = & \quad \text{– 6.6 –} \\
 & \Phi_{\mu\mathsf{F}}\langle\, \llparenthesis\, \Phi_{\nu\mathsf{F}^{\times}_{C}}(\mathrm{out}) \,\rrparenthesis, \mathrm{in}^{-1} \,\rangle
 \end{array}\right. \\
 & \llparenthesis\, \Phi \,\rrparenthesis^{\mathrm{m}}
\end{array}\right.
$$

$\square$

Proposition 6.11 *For any* F*-cv-algebra* $\varphi : \mathsf{F}(\mathsf{F}^{\nu}(C)) \to C$

$$\{\!|\,\varphi\,|\!\} = \{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \varphi \circ \mathsf{F}\llbracket\,\gamma\,\rrbracket\,|\!\}^{\mathrm{m}} \qquad \text{mhisto-H}\textsc{isto}$$

Proof.

$$\begin{array}{ll}
& \{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \varphi \circ \mathsf{F}\llbracket\,\gamma\,\rrbracket\,|\!\}^{\mathrm{m}} \\
= & \quad \text{– histo-C}\textsc{harn}\text{ –} \\
& \left[\begin{array}{ll}
& \{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \varphi \circ \mathsf{F}\llbracket\,\gamma\,\rrbracket\,|\!\}^{\mathrm{m}} \circ \mathrm{in} \\
= & \quad \text{– mhisto-S}\textsc{elf}\text{ –} \\
& \varphi \circ \mathsf{F}\llbracket\,\langle\,\{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \varphi \circ \mathsf{F}\llbracket\,\gamma\,\rrbracket\,|\!\}^{\mathrm{m}}, \mathrm{in}^{-1}\,\rangle\,\rrbracket
\end{array}\right. \\
& \{\!|\,\varphi\,|\!\}
\end{array}$$

$\square$

Every m-catamorphism can be defined as m-histomorphism, which uses only the value on the “predecessor” of the current argument.

Proposition 6.12 *For any* F*-malgebra* (C, Φ)

$$(\!|\,\Phi\,|\!)^{\mathrm{m}} = \{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \Phi(\mathrm{fst} \circ \gamma)\,|\!\}^{\mathrm{m}} \qquad \text{mhisto-MC}\textsc{ata}$$

Proof.

$$\begin{array}{ll}
& \{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \Phi(\mathrm{fst} \circ \gamma)\,|\!\}^{\mathrm{m}} \\
= & \quad \text{– mcata-C}\textsc{harn}\text{ –} \\
& \left[\begin{array}{ll}
& \{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \Phi(\mathrm{fst} \circ \gamma)\,|\!\}^{\mathrm{m}} \circ \mathrm{in} \\
= & \quad \text{– mhisto-S}\textsc{elf}\text{ –} \\
& \Phi(\mathrm{fst} \circ \langle\,\{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \Phi(\mathrm{fst} \circ \gamma)\,|\!\}^{\mathrm{m}}, \mathrm{in}^{-1}\,\rangle) \\
= & \quad \text{– pairing –} \\
& \Phi\{\!|\,\lambda A, \gamma : A \to C \times \mathsf{F}(A).\ \Phi(\mathrm{fst} \circ \gamma)\,|\!\}^{\mathrm{m}}
\end{array}\right. \\
& (\!|\,\Phi\,|\!)^{\mathrm{m}}
\end{array}$$

$\square$

6.4 Mendler-style recursion operators in Haskell

In Haskell, we can implement m-catamorphisms as follows:

```
> mcata :: (forall a. (a -> c) -> f a -> c)
>                             -> Mu f -> c
> mcata phi (In x) = phi (mcata phi) x
```

The constraint for `phi`, that it is Mendler-style algebra, is expressed by its typing, which requires `phi` to be polymorphic on `a`. Differently from conventional catamorphisms, the type for `mcata` does not contain the restriction for type constructor `f` to be an instance of class `Functor`. This is not required, as the defining equation (which expresses the cancellation law), does not use `fmap`. Note that there was no such requirement in the definition of type `Mu f` either. Hence, if we use `mcata` combinator instead of `cata`, we can define inductive types only by defining the corresponding type constructor, and no instance declaration for class `Functor` is required.

Example 6.1 (naturals to integers)
The function $nat2int$, which converts naturals to corresponding integers, can be defined as m-catamorphism:

```
> nat2int :: Nat -> Int
> nat2int = mcata phi
>    where phi n2i Z     = 0
>          phi n2i (S n) = 1 + n2i n
```

Note that `n` in the second equation of `phi` corresponds to the original predecessor, and not to the value of the function on it (as it had been case if we had used `cata`). The value on the predecessor is computed by applying to it the function provided as the first argument of `phi`. Using the suitable naming of this argument, the definition of `phi` becomes very similar to the directly recursive definition for the function $nat2int$. □

Example 6.2 (length)
The function, which computes the length of a given list, can be defined as follows:

```
> lengthM :: List a -> Nat
> lengthM = mcata phi
>     where phi len N        = zeroN
>           phi len (C _ xs) = succN (len xs)
```

□

The Haskell correspondent for a F-rec-malgebra is a polymorphic function of type `(a -> (c, Mu f)) -> (f a -> c)` for some fixed type constructor `f` and type `c`. However, for convenience, we use an equivalent version of it; namely, `(a -> c) -> (a -> Mu f) -> (f a -> c)`. Now, we can implement m-paramorphisms, by using the accordingly modified cancellation law, as follows:

```
> mpara :: (forall a. (a -> c) -> (a -> Mu f)
>                             ->  f a -> c)
>                             ->  Mu f -> c
> mpara phi (In x) = phi (mpara phi) id x
```

Example 6.3 (factorial)
The factorial function can be implemented as Mendler-style paramorphism:

```
> factM :: Nat -> Nat
> factM = mpara phi
>     where phi fac i Z = oneN
>           phi fac i (S x)
>                        = mulN (succN (i x)) (fac x)
```

The first functional argument of `phi` is used for computing the value on the previous argument `x` (like in the case of `mcata` combinator). However, now `phi` has also the second functional argument `i`, which is applied to the previous argument in places where the argument itself is needed. □

Example 6.4 (dropwhile)
The function *dropWhile* can be implemented as follows:

```
> dropWhileM :: (a -> Bool) -> List a -> List a
> dropWhileM p = mpara phi
>     where phi dropW i N          = nilL
>           phi dropW i (C x xs)
>                     | p x        = i xs
>                     | otherwise = consL x (dropW xs)
```

□

The Haskell correspondent for a F-cv-malgebra is a polymorphic function of type `(a -> (c, f a)) -> (f a -> c)` for some fixed type constructor `f` and type `c`. Again, for convenience, we use a slightly modified, but equivalent, version of it; namely, `(a -> c) -> (a -> f a) -> (f a -> c)`. Now, we can implement m-histomorphisms, by using the accordingly modified cancellation law, as follows:

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

Note, that the definition does not use any intermediate data or codata structure. Hence, it does not memoize values on previous arguments. (However, it is possible to arrive to the memoizing version by exploiting mhisto-DEF.)

Example 6.5 (Fibonacci)
The Fibonacci function can be implemented as Mendler-style histomorphism:

```
> fiboM :: Nat -> Int
> fiboM = mhisto phi
>     where phi fib pre Z      =  1
>           phi fib pre (S x)
>                      = case pre x of
>                          Z   -> 1
>                          S y -> fib x + fib y
```

The first functional argument of `phi` is used for computing the value on the previous argument `x` (like in the case of `mcata` or `mpara` combinator). However, now `phi` has also the second functional argument `pre`, which is applied to the previous argument `x` in places where its predecessor is needed. □

Example 6.6 (evens)
The function `evens`, which takes from the given list every second element, can be defined as follows:

```
> evensM :: List a -> List a
> evensM = mhisto phi
>     where phi eve pre N          =  nilL
>           phi eve pre (C _ x)
>                     = case pre x of
>                           N     -> nilL
>                           C a y -> consL a (eve y)
```

□

6.5 Related work

Mendler-style recursion combinators were invented in type theory by N. P. Mendler. In [Men87] (a conference paper), he studied an extension of system F with (co)inductive types and primitive (co)recursion; [Men91] (its journal version) treats a simplified calculus that only supported (co)iteration. Some important works commenting on [Men87]/[Men91] and, in particular, on the embeddings between simply typed lambda calculi with conventional- and Mendler-style iterators and primitive-recursors and system F are [Lei90, Geu92, Spł93]. Mendler-style course-of-value iteration was studied by us in a type-theoretic setting in [UV97, UV00b, Uus98]. In [UV00a] we also studied a Mendler-style combinator for simultaneous iteration.

CHAPTER 7

CONCLUSIONS

In this last chapter we summarize the contribution of this thesis and outline some possible directions for future work.

7.1 Summary

We have studied the theory of inductive and coinductive types in a categorical framework. The goal of this thesis was to develop new recursion combinators that capture more complex recursion patterns than simple (co)iteration but still possess nice reasoning properties. In particular, we considered combinators for primitive (co)recursion and course-of-value (co)iteration using two different approaches.

The first approach was based on the treatment of inductive and coinductive types as initial algebras and terminal coalgebras. In this setting, it is well known that the primitive recursion can be simulated by a simple iteration which computes a value paired together with the argument, and that this construction leads to the notion of paramorphism which captures the primitive recursion directly. We showed (in Chapter 3), that the obvious dualization of this construction leads to notion of apomorphism which captures the recursion pattern known as primitive corecursion. More importantly, we also showed (in Chapter 4) that a more involved generic simulation of memoization by iteration leads to the notion of histomorphism, a direct formalization of course-of-value iteration, and also described the dual notion of futumorphism, a formalization of course-of-value coiteration.

The second approach, inspired by type-theoretic work by N. P. Mendler, was here pursued for inductive types only. To recast Mendler's work in category-theoretic terms, we invented the concepts of malgebra and malgebra homomorphism and treated inductive types as initial malgebras (chapter 5). From that basis, we then introduced Mendler-style analogs for the cata, para and histo combinators (chapter 6). From the theory developed, it appears that Mender-style recursion combinators are just as well-suited for program calculation as the conventional ones, but support a programming style more close to customary (general-) recursive programming.

7.2 Future work

Semantics of Mendler-style inductive and coinductive types. While the basic theory of Mendler-style inductive and coinductive types has been settled, many questions remain still unresolved. First, the precise conditions of the existence of initial (terminal) Mendler-style (co)algebras for mixed-variant base functors and their relationship to Freyd's dialgebras [Fre90, Fre91] need further study. Also, recently, Bird and others [BM98, BP99] have proposed a new approach for nested data types. How this work relates to ours is currently unclear and is a very interesting topic to investigate.

Modeling of interactive processes. Coalgebras and coinductive types have received much attention recently. They facilitate elegant modeling of interactive processes and several very important notions of object-oriented programming like objects, classes and inheritance. Our preliminary investigations on Mendler-style coinductive types show that at least modeling of simple processes is easily achievable by them. As the next step, we plan to use Mendler-style coinductive types to model more complex process calculi (like CSP or CCS), and, if we succeed in this, we start to develop the specification methodology of processes based on these models. We also plan to provide several case studies for specifying processes using the methodology.

Computations with side-effects and (co)inductive types. The use of monads to represent side-effecting computations is nowadays considered standard, and for instance in lazy functional language Haskell they are the main structuring language construction for side-effects including input/output. The popularity of using monads is caused by the fact that they provide a simple and effective way to handle computations that interact purely functionally but internally use side-effects. At the same time, the monadic approach is not without shortcomings, as the model it provides for input/output assumes that the environment is closed (i.e. the program is the only one which interacts with environment). Recently Kieburtz [Kie99] proposed a conjecture that comonads (duals of monads) together with coinductive types yield a more appropriate formalism for modeling the interaction with outer environment. We plan to verify this conjecture, and more generally to investigate the possibilities for integration of monads and comonads with Mendler-style (co)inductive types.

Generic programming. Genericity and reusability are two important issues for simplifying the design and maintenance of programs. The purpose of generic programming [BJJM99] is to develop new methods to parameterize algorithms and programs. For instance, while traditional polymorphism allows parameterization with respect to types, the so-called polytypism [JJ96] allows also parameterization also with respect to type constructors. Most of the approaches for generic programming are using inductive and coinductive types, as they come equipped with universal combinators representing different generic recursion schemes. Mendler-style inductive and coinductive types have the same potential, but their real utility in generic programming needs further investigation.

Program transformation. The genericity and reusability of programs have a side-effect that resulting programs can be very resource-consuming. The problem can be solved by using different program transformation techniques, like partial evaluation or deforestation. In the context of inductive and coinductive types, the last is especially interesting, as it allows to eliminate data structures constructed during intermediate computations, and can be made fully automatic. Traditional deforestation is based on the unfold-fold method by Burstall and Darlington [BD77], and is quite inefficient as it requires keeping the full computation history to guarantee the termination. Takano and Meijer [TM95] proposed an alternative approach based on (co)inductive types, called "acid rain", where intermediate data structures are removed using pure calculation, and keeping the computation history is not required. We hope that this method can be generalized for Mendler-style (co)inductive types. Also, we plan to investigate other program transformation methods in this setting.

REFERENCES

[BBA00] L. S. Barbosa, J. B. Barros, and J. J. Almeida. Polytypic recursion patterns. To appear in Proc. SBLP'00, vol. of ENTCS, May 2000.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BdM97] Richard Bird and Oege de Moor. *Algebra of Programming*, volume 100 of *Prentice Hall Int. Series in Computer Science*. Prentice Hall, London, 1997.

[Bir87] Richard S. 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 3–42. Springer-Verlag, Berlin, 1987.

[Bir98] Richard S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Int. Series in Computer Science. Prentice Hall, London, 2nd edition, 1998.

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming –an introduction–. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Revised Lectures 3rd Int. School on Advanced Functional Programming, AFP'98, Braga, Portugal, 12–19 Sept. 1998*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, Berlin, 1999.

[BM98] Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, Berlin, 1998.

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

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report 92/480/18, Dept. of Computer Science, Univ. of Calgary, June 1992.

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

[Fre90] Peter J. Freyd. Recursive types reduced to inductive types. In *Proceedings 5th IEEE Annual Symp. on Logic in Computer Science, LICS'90, Philadelphia, PA, USA, 4–7 June 1990*, pages 498–507. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Fre91] Peter J. Freyd. Algebraically complete categories. In A. Carboni, M. C. Pedicchio, and G. Rosolini, editors, *Proceedings Int. Conf. Category Theory '90, CT'90, Como, Italy, 22–28 July 1990*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, Berlin, 1991.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings 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. ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.ps.Z.

[GH99] Jeremy Gibbons and Graham Hutton. Proof methods for structured corecursive programs. In *Proceedings 1st Scottish Functional Programming Workshop, Stirling, Scotland, Aug/Sept 1999*, page ??? 1999.

[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 for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh, September 1987.

[HITT96] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals.

In *Proceedings 2nd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'97, Amsterdam, The Netherlands, 9–11 June 1997*, pages 164–175. ACM Press, New York, 1996.

[Hoa72] C. A. R. Hoare. Notes on data structuring. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[How96] Brian T. Howard. Inductive, coinductive, and pointed types. In *Proceedings 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'96, Philadelphia, PA, USA, 24–26 May 1996*, SIGPLAN Notices 31(6), pages 102–109. ACM Press, New York, 1996.

[JJ96] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, Berlin, 1996.

[Kie99] Richard Kieburtz. Codata and comonads in Haskell. Unpublished manuscript, July 1999.

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

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

[Mac97] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 2nd edition, 1997. (1st ed., 1971).

[Mal90a] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, 1990.

[Mal90b] Grant R. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Dept. of Computer Science, Univ. of Groningen, 1990.

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

[Mat00] Ralph Matthes. Tarski's fixed-point theorem and lambda calculi with monotone inductive types. In Benedikt Löwe and Florian Rudolph, editors, *Refereed Papers of Research Coll. on Foundations of the Formal Sciences, Berlin, Germany, 7–9 May 1999*, pages 91–112. Kluwer Academic Publishers, Dordrecht, 2000.

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

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

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

[Mic68] Donald Michie. Memo functions and machine learning. *Nature*, (218):19–22, April 1968.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Mul91] Philip S. Mulry. Strong monads, algebras and fixed points. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science: Proceedings LMS Symp., Durham, UK, 20–30 July 1991*, volume 177 of *London Math. Society Lecture Note Series*, pages 202–216. Cambridge University Press, Cambridge, 1991.

[PJH99] Simon Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell98, A Non-strict Purely Functional Language*, February 1999. Available from `http://www.haskell.org/definition/`.

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

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

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 306–316. ACM Press, New York, 1995.

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

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

[UV99a] Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, 1999.

[UV99b] Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *INFORMATICA*, 10(1):5–26, 1999.

[UV00a] Tarmo Uustalu and Varmo Vene. Coding recursion a la Mendler (extended abstract). In Johan Jeuring, editor, *Proceedings 2nd Workshop on Generic Programming, WGP'2000, Ponte de Lima, Portugal, 6 July 2000*, Tech. Report UU-CS-2000-19, Dept. of Computer Science, Utrecht Univ., pages 69–85. June 2000.

[UV00b] Tarmo Uustalu and Varmo Vene. Least and greatest fixedpoints in intuitionistic natural deduction. To appear in *Theoretical Computer Science*, March 2000.

[Ves97] Peter Vesely. Typechecking the Charity term logic. Unpublished notes, April 1997.

[Vos95] Tanja Vos. Program construction and generation based on recursive types. MSc thesis INF/SCR-95-12, Dept. of Computer Science, Univ. of Utrecht, March 1995.

[VU98] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998.

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

KATEGOORNE PROGRAMMEERIMINE INDUKTIIVSETE JA KOINDUKTIIVSETE TÜÜPIDEGA

Kokkuvõte

Algoritmika ehk programmide konstrueerimise matemaatika on teoreetilise informaatika haru, mille eesmärgiks on uute matemaatiliselt põhjendatud tarkvaratehnika meetodide väljatöötamine. Seejuures kasutatav matemaatiline aparatuur baseerub põhiliselt universaalalgebral ja loogikal, ning eriti just viimasel ajal kategooriate teoorial. Algoritmika üks olulisemaid tunnuseid on, et tulemprogrammi korrektsus spetsifikatsiooni suhtes garanteeritakse konstruktsiooni käigus ning selle eraldi verifitseerimist ei ole vaja. Eelistatakse deklaratiivseid programmeerimisparadigmasid, iseäranis tüübitud funktsionaalseid keeli, kuna nende semantiline baas on väga lähedane kasutatava matemaatilise aparatuuriga. Muuhulgas võimaldab see nii spetsifitseerimis- kui ka realiseerimisfaasis jääda ühe paradigma piiresse.

Käesolevas doktoritöös on kategooriate teooria abil uuritud induktiivseid ja koinduktiivseid andmetüüpe ja nendega seotud rekursiooniskeeme. Töö käigus jõuti järgmiste uute tulemusteni:

- Uuriti korekursiivsete funktsioonide defineerimise skeemi, mille formalisatsiooniks terminaalsete koalgebratega distributiivsetes kategooriates on nn. apomorfismid (primitiivne korekursioon); sõnastati ja tõestati apomorfismide iseloomulikud omadused, võrreldi neid anamorfismidega (lihtsa koiteratsiooniga); esitati lihtsaid näiteid koandmetüüpidega funktsionaalprogrammeerimisest, kus apomorfismid on tululikud.

- Uuriti rekursiivsete ja korekursiivsete funktsioonide defineerimise erinevaid skeeme. Näidati, et *course-of-value*-iteratiivsed funktsioonid on formaliseeritavad initsiaalsete algebratega distributiivsetes kategooriates nn. histomorfismidena ning *course-of-value*-koiteratiivsed funktsioonid on duaalselt formaliseeritavad terminaalsete koalgebratega distributiivsetes kategooriates nn. futumorfismidena.

- Formaliseeriti nn. Mendleri-laadi induktiivsete tüüpide kategoorne semantika, tuues selleks sisse Mendleri-laadi algebrate ning nende vaheliste homomorfismide mõisteid. Näidati, et kovariantse baasfunktori korral on indutseeritud Mendleri-laadi algebrate kategooria ekvivalentne sama funktori (tavaliste) algebrate kategooriaga. Segavariantse baasfunktori jaoks näidati, et kui baaskategoorias leiduvad teatud suured summad (täpsemalt teatud

tensorite kolõpud), siis saab konstrueerida uue kovariantse funktori, mille algebrate kategooria on esialgse funktori Mendleri-laadi algebrate kategooriaga ekvivalentne. Lisaks uuriti Mendleri-laadi induktiivsete tüüpidega seotud rekursioonioperaatorite omadusi ning nende kasutatavust programmide konstrueerimisel.

ACKNOWLEDGEMENTS

First and foremost, my greatest thanks go to Tarmo Uustalu who has been not only a direct partner of the research reported here, but also a good friend. Our collaboration has been great pleasure and I can only hope that it will continue. The discussions we had in Spring 1996 during my visit to Stockholm were the starting point of the work and is still one of the greatest experience I have had in my academic life.

Special thanks to my supervisor Merik Meriste who is responsible for igniting my interest in programming language theory and functional languages in particular. He has always been very supportive and patient, especially at difficult moments throughout my PhD study.

I want to thank Prof. Enn Tõugu for inviting me to the Royal Institute of Technology in Stockholm for a two two-month periods during 1996. These visits were financed by the Stockholm-Ladugårdslandet Club of District 2350 of Rotary International for which I am very grateful.

I would like to thank all my friends, who often wondered whether this thesis will ever be finished, for their support.

Last but not least, I am grateful to my mother for the support and encouragement all over the years.

The work reported in this thesis was partially supported by the Estonian Science Foundation grant no. 2976.

CURRICULUM VITAE

VARMO VENE

Citizenship: Estonian Republic.

Born: July 2, 1968, Tartu, Estonia.

Marital status: single.

Address: Anne 90-58, Tartu, EE-50705 Estonia,
phone.: +372 7 482 460,
e-mail: varmo@cs.ut.ee

Education

1986 – 1992 Applied mathematics, Faculty of Mathematics, University of Tartu.

1992 – 1994 MSc studies in Computer Science, Faculty of Mathematics, University of Tartu.

1996 – 2000 PhD studies in Computer Science, Faculty of Mathematics, University of Tartu.

Professional employment

1994 – 2000 Researcher, Institute of Computer Science, University of Tartu.

1998 – Researcher, Institute of Cybernetics, Tallinn Technical University.

2000 – Lecturer, Institute of Computer Science, University of Tartu.

CURRICULUM VITAE

VARMO VENE

Kodakondsus: Eesti Vabariik.

Sünniaeg ja -koht: 2. juuli, 1968, Tartu, Eesti.

Perekonnaseis: vallaline.

Aadress: Anne 90-58, Tartu, EE-50705 Eesti,
tel.: +372 7 482 460,
e-post: varmo@cs.ut.ee

Haridus

1986 – 1992 Tartu Ülikool, matemaatikateaduskond, rakendusmatemaatika eriala.

1992 – 1994 Tartu Ülikool, matemaatikateaduskond, informaatika magistratuur.

1996 – 2000 Tartu Ülikool, matemaatikateaduskond, informaatika doktorantuur.

Erialane teenistuskäik

1994 – 2000 Tartu Ülikool, Arvutiteaduse Instituut, teadur.

1998 – Tallinna Tehnikaülikool, Küberneetika Instituut, teadur (0.3 kohta).

2000 – Tartu Ülikool, Arvutiteaduse Instituut, lektor.

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

1. Mati Heinloo. The design of nonhomogeneous spherical vessels, cylindrical tubes and circular discs. Tartu, 1991. 23 p.
2. Boris Komrakov. Primitive actions and the Sophus Lie problem. Tartu, 1991. 14 p.
3. Jaak Heinloo. Phenomenological (continuum) theory of turbulence. Tartu, 1992. 47 p.
4. Ants Tauts. Infinite formulae in intuitionistic logic of higher order. Tartu, 1992. 15 p.
5. Tarmo Soomere. Kinetic theory of Rossby waves. Tartu, 1992. 32 p.
6. Jüri Majak. Optimization of plastic axisymmetric plates and shells in the case of Von Mises yield condition. Tartu, 1992. 32 p.
7. Ants Aasma. Matrix transformations of summability and absolute summability fields of matrix methods. Tartu, 1993. 32 p.
8. Helle Hein. Optimization of plastic axisymmetric plates and shells with piece-wise constant thickness. Tartu, 1993. 28 p.
9. Toomas Kiho. Study of optimality of iterated Lavrentiev method and its generalizations. Tartu, 1994. 23 p.
10. Arne Kokk. Joint spectral theory and extension of non-trivial multiplicative linear functionals. Tartu, 1995. 165 p.
11. Toomas Lepikult. Automated calculation of dynamically loaded rigidplastic structures. Tartu, 1995. 93 p. (in russian)
12. Sander Hannus. Parametrical optimization of the plastic cylindrical shells by taking into account geometrical and physical nonlinearities. Tartu, 1995. 74 p. (in russian)
13. Sergrei Tupailo. Hilbert's epsilon-symbol in predicative subsystems of analysis. Tartu, 1996. 134 p.
14. Enno Saks. Analysis and optimization of elastic-plastic shafts in torsion. Tartu, 1996. 96 p.
15. Valdis Laan. Pullbacks and flatness properties of acts. Tartu, 1999. 90 p.
16. Märt Põldvere. Subspaces of Banach spaces having Phelps' uniqueness property. Tartu, 1999. 74 p.
17. Jelena Ausekle. Compactness of operators in Lorentz and Orlicz sequence spaces. Tartu, 1999. 72 p.

18. Krista Fischer. Structural mean models for analyzing the effects of compliance in clinical trials. Tartu, 1999. 125 p.
19. Helger Lipmaa. Secure and efficent time-stamping systems. Tartu, 1999. 56 p.
20. Jüri Lember. Consistency of empirical k-centres. Tartu, 1999. 148 p.
21. Ella Puman. Optimization of plastic conical shells. Tartu, 2000. 102 p.
22. Kaili Müürisep. Eesti keele arvutigrammatika: süntaks. Tartu, 2000. 109 lk.