Bachelor thesis
Computing Science

Radboud University

Creating Canned Recursion in Functional Programming Languages from Category Theory

Author:
Luko van der Maas
`l.vandermaas@student.ru.nl`
s1010320

Supervisor:
prof. dr. Herman Geuvers
`H.Geuvers@cs.ru.nl`

Assessor:
dr. Sjaak Smetsers
`S.Smetsers@cs.ru.nl`

January 18, 2020

Abstract

We study schemes for creating canned recursion using category theory. We present an overview of different types of recursion including, (co)iteration, primitive recursion and course-of-value recursion using catamorphisms, anamorphisms, paramorphisms, hylomorphisms and histomorphisms. For all of these schemes we discuss their origin, prove correctness and then demonstrate their utility by some examples.

Contents

Chapter 1

Introduction

In functional programming, recursion is one of the most important concepts to grasp. It is the building block to create more powerful programs. However, when programming with recursion, we are often prone to make small mistakes. Furthermore, when performing recursion on a data type, we often create the exact same function structure for a lot of different functions on this data type. Hence, researchers have made abstractions for these functions. Quite early on, these researchers thought of using a branch of mathematics called "category theory", using it to abstractly talk about recursively defined functions. Category theory is centred around objects (types) and arrow (programs), which thus lends itself very well to functional programming.

There have been many attempts to create this so called canned recursion using category theory. In 1991 Meijer et al. [8] were one of the first to come up with methods of using category theory to describe recursive functions on lists. Since then, creating recursion schemes with category theory has been a hot topic in theoretical computing science. Many schemes have been introduced, with most of them aimed at different types of recursion. To categorize these types and explain them consistently, we provide an overview of existing methods of creating canned recursion using category theory. We present this overview in a notation partially derived from Meijer et al. [8], Uustalu and Vene [11], the PhD thesis of Vene [13] and with the categorical notation from Pierce [10]. A new notation for proofs has been added to improve readability and understanding. Also some proofs that have been skipped in the original papers will be added.

In Part I (Preliminaries) we introduce category theory and its notation. We start from scratch and work our way through examples and universal constructions until we have a definition of a Cartesian Closed Category. We also define the category **FPL** in which we do most of the work when defining canned recursion. We then move on to defining Functors and F-algebras. If all of this is already known, it is advised to read Section 2.1.2 and Section 2.1.4 where **FPL** is defined and the notation of proofs is explained.

In Part II, we describe four forms of canned recursion. In Chapter 3 we first define initial F-algebras and then we will define the catamorphism and anamorphism followed by primitive recursion with catamorphisms. Chapter 4 will introduce the hylomorphism where we discuss it in the context of categories with coinciding initial and terminal objects and in the context of categories that do not have this property. In Chapter 5 we describe course-of-value recursion, explain how to achieve it using histomorphisms and demonstrate some of its limitations.

Part I

Preliminaries

Chapter 2

Category theory

In this chapter we introduce category theory. Our presentation is mostly based on Pierce [10] with some additions from Milewski [9].

2.1 Definition of a category

A category is a collection of objects and arrows. An arrow f points from one object A to another B: $f : A \to B$. If we have second arrow $g : B \to C$, we can compose these arrows: $(g \circ f) : A \to C$. We can very easily present these categories using diagrams.

$$\begin{tikzcd}A \arrow[r, "f"] & B \arrow[r, "g"] & C\end{tikzcd}$$

and because arrows compose, we get a third arrow for free.

$$\begin{tikzcd}A \arrow[r, "f"] \arrow[rr, bend right, "g\circ f"'] & B \arrow[r, "g"] & C\end{tikzcd}$$

We are now ready to define a category rigorously.

Definition 2.1 (*Category*)

A category $\mathbf{C}$ is a tuple $(\mathsf{Ob}_C, \mathsf{Arr}_C)$, where:

- $\mathsf{Ob}_{\mathbf{C}}$ is a collection of objects.
- $\mathsf{Arr}_{\mathbf{C}}$ is a collection of arrows with a domain and codomain, so for every $f \in \mathsf{Arr}_{\mathbf{C}}$, there are $A, B \in \mathsf{Ob}_{\mathbf{C}}$ with $f : A \to B$.

satisfies the following laws:

- Composition of arrows as denoted by $\circ$ behaves such that if $f \in \mathsf{Arr}_{\mathbf{C}} : A \to B$ and $g \in \mathsf{Arr}_{\mathbf{C}} : B \to C$ then $g \circ f : A \to C \in \mathsf{Arr}_{\mathbf{C}}$.

- Composition of arrows follows associativity: for any arrows $f \in \mathsf{Arr}_{\mathbf{C}} : A \to B$, $g \in \mathsf{Arr}_{\mathbf{C}} : B \to C$ and $h \in \mathsf{Arr}_{\mathbf{C}} : C \to D$

$$h \circ (g \circ f) = (h \circ g) \circ f \tag{Assoc}$$

- Every object has an identity arrow: for any object $A \in \mathsf{Ob}_{\mathbf{C}}$, an arrow id_A[1] $: A \to A$ exists that satisfies the identity law:

  For any $f \in \mathsf{Arr}_{\mathbf{C}} : A \to B$

$$\mathsf{id}_B \circ f = f \tag{Identity-Left}$$
$$f \circ \mathsf{id}_A = f \tag{Identity-Right}$$

2.1.1 Example categories

Many fields in mathematics and computing science make use of structures that fit very well in the definition of a category. To provide some intuition about what a category is, we will give some examples.

Example 2.2 (*The cateogory of sets*)

The most obvious category that exists is **Set**. In the category **Set**, the sets are the objects and the total functions are the arrows. Composition is just composition of functions. Associativity holds because composition of functions is associative. The identity arrow also exists because that is the identity function, which is defined for every set.

We can also create categories that look very similar to **Set** but where the functions have to preserve more properties on the relation of the elements in the sets.

[1]Note that the subscript of arrows like id_A will often be dropped such that we just have id. It will be clear from the context which identity function is used.

Example 2.3 (*The category of posets*)

One of these types of sets where the function has to preserve a property is a partial ordering, also called a poset. A poset consists of a set P and and a binary relation between elements of the set: $\leq_P$, this relation has to satisfy reflexivity, transitivity and antisymmetry. The category **Poset** is defined as follows:

- $\mathsf{Ob}_{\mathbf{Poset}}$ contain the pairs $(P, \leq_P)$ where $\leq_P$ is a partial ordering of P.
- $\mathsf{Arr}_{\mathbf{Poset}}$ contains the order preserving functions between the sets: $f : (P, \leq_P) \to (Q, \leq_Q)$, if $f : P \to Q$ and $p \leq_P p'$ then $f(p) \leq_Q f(p')$.

Now we need to show that this definition satisfies the defined laws:

- Composition is again composition of functions, but we do have to show that it preserves the partial orderings in the sets.

  Proof. We assume we have two order preserving functions: $f : (P, \leq_P) \to (Q, \leq_Q)$ and $g : (Q, \leq_Q) \to (R, \leq_R)$. If we now know $p \leq_P p'$ then from the order preserving nature of f we know that $f(p) \leq_Q f(p')$. But we now also know from the order preserving nature of g that $g(f(p)) \leq_R g(f(p'))$. Thus, $g \circ f$ preserved the partial order and thus composition is order preserving. □

- The identity arrow of an object is just the identity function, it preserves order trivially.
- Associativity of the composition operator follows from the associativity of the composition of functions.

We can also turn a Poset itself into a category:

Example 2.4 (*The category of one poset*)

We now define a category based on one poset P.

- $\mathsf{Ob}_{\mathbf{P}}$ contains the elements $p \in P$.
- $\mathsf{Arr}_{\mathbf{P}}$ contains an arrow between two objects if there is a relation between the two elements represented by the object. For any two elements $p, q \in P$, there is an arrow from p to q iff $p \leq q$. Arrows are thus unique.

Then for the laws

- Composition of arrows holds because $\leq$ is transitive.

- The identity arrows exist because $\leq$ is reflexive.
- Associativity holds because there is at most one arrow between every pair of elements.

Remark. From this result it also follows that the **Poset** category is a category of categories. We will see more of those later on.

2.1.2 Functional programming as a category

Functional programming combines many ideas and concepts originating from category theory. Translating a basic functional programming language to a category can be done in a fairly simple way. We will call this category **FPL**, which is defined in the following manner

Definition 2.5 (*FPL*)

The category **FPL** of a functional programming language is defined as

- Objects are types.
- Arrows are programs.

With the laws proven as follows

- Composition follows from composition of programs.
- The identity arrow is the identity program that just returns its argument.
- Associativity of composition follows from the associativity of composition of programs.[2]

In this category, we will also have all universal constructions as will be described in Section 2.2. With these additions, we can use universal constructions in a functional programming language.[3]

2.1.3 Dualisation

In category theory, any category $\mathbf{C}$ also has a dual category $\mathbf{C}^{\mathrm{op}}$, called $\mathbf{C}$ opposite.

[3]Defining the category as we did above does have a few problems. First of all some functional programming languages still have some side effects like exceptions. We will ignore the existence of these language features. But most prominently we will often talk about two arrows being equal or an arrow being the unique solution. In almost all languages programs are only equal if they are exactly the same. In these languages we can reason about program equality, but this is not part of the language and thus not really part of the category. This will mostly be ignored as programs being unique does not really matter for implementing them.

Definition 2.6 (*Dualisation*)

For any category $\mathbf{C}$, there exists a dual category $\mathbf{C}^{\text{op}}$. If $\mathbf{C}$ has a name "x", then the dual category is often called "co-x". $\mathbf{C}^{\text{op}}$ has the same objects as $\mathbf{C}$, but it has every arrow reversed.

Any construction we build around category theory also comes in pairs, where one of the pair is just the other with the arrows reversed. Not only categories and structures are reversed, we can also dualise our proofs. This often means that, if we have proven something for a category $\mathbf{C}$, it will also hold for any $\mathbf{C}^{\text{op}}$ if we swap the domains and codomains. This so-called principle of dualisation results in many free theorems.

2.1.4 Notation of proofs

There are two main ways to prove statements in category theory. First, there are the usual equality proofs in which we start of at one side of the equals sign and work our way with known laws towards the other side of the equals sign. However, in category theory researchers also very commonly depict proofs using diagrams. In these diagrams all needed objects and the arrows between them are written down. These diagrams commute if nothing is said otherwise. Below is a definition, first in textual style and then in the form of diagrams.

Definition 2.7 (*Commutativity*)

If four arrows $f : X \to Z, f' : W \to Y, g : X \to W, g' : Z \to Y$ are said to commute, then the following holds

$$f' \circ g = g' \circ f \qquad \textsc{Commu}$$

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "g"] & Z \arrow[d, "g'"] \\
W \arrow[r, "f'"] & Y
\end{tikzcd}$$

If a diagram is said to commute, then all compositions of arrows that have the same start and end are equal.

When proving a theorem using diagrams, we will describe the building of the diagram such that in the end a certain property follows from the diagram. This is often seen as building the inner squares of a diagram, such that the outer square commutes. This outer square is then what we want to prove.

When proving a theorem using equational reasoning, we will use a very specific notation. It starts with zero or more assumptions denoted by symbols like $\triangleright$ and $\blacktriangleright$. Following these assumptions, the proof begins with the starting equation. Then, a rule is given between every two statements that shows why the two statements are equal.

We will be combining these proof methods to give more clarity. A symbol in a rectangle in a proof will denote that the commutativity of this rectangle is equal to the equation named with that symbol. Certain rows in a proof will also be coloured, these colours correspond to composition paths in the diagram. The entire path will not always be shown but the important parts will be. The colors are used to make sure that a line in the proof can easily be found in the diagram.

We will now prove a simple theorem with the combination of a written proof and a diagram.

Theorem 2.8 (*Combining commutativity*)

If the two inner squares of the following diagram commute, then so does the outer one.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "a"] & B \arrow[r, "f'"] \arrow[d, "b"] & C \arrow[d, "c"] \\
A' \arrow[r, "g"] & B' \arrow[r, "g'"] & C'
\end{tikzcd}$$

Thus, if $b \circ f = g \circ a$ and $c \circ f' = g' \circ b$, then $c \circ f' \circ f = g' \circ g \circ a$.

Proof.

$\triangleright \; b \circ f = g \circ a$

$\blacktriangleright \; c \circ f' = g' \circ b$

$(c \circ f') \circ f$

$= \quad - \blacktriangleright -$

$(g' \circ b) \circ f$

$= \quad -$ Assoc $-$

$g' \circ (b \circ f)$

$= \quad - \triangleright -$

$g' \circ (g \circ a)$

$= \quad -$ Assoc $-$

$(g' \circ g) \circ a$

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "a"] \arrow[dr, phantom, "\triangleright"] & B \arrow[r, "f'"] \arrow[d, "b"] \arrow[dr, phantom, "\blacktriangleright"] & C \arrow[d, "c"] \\
A' \arrow[r, "g"] & B' \arrow[r, "g'"] & C'
\end{tikzcd}$$

$\square$

2.1.5 Isomorphisms

Isomorphisms are about arrows that have an inverse.

Definition 2.9 (*Isomorphisms*)

An arrow $f : A \to B$ is an isomorphism if there exists an $f^{-1} : B \to A$ such that $f \circ f^{-1} = \mathsf{id}$ and $f^{-1} \circ f = \mathsf{id}$. Thus, the following diagram commutes:

$$\begin{tikzcd} A \arrow[loop left, "\mathsf{id}"] \arrow[rr, bend left, "f"] & & B \arrow[ll, bend left, "f^{-1}"] \arrow[loop right, "\mathsf{id}"] \end{tikzcd}$$

It is often useful to talk about whether or not two objects are equal. When they are the exact same object this is quite easy, but there is a more interesting way to define equality in categories, namely equality up to isomorphism.

Definition 2.10 (*Equality up to isomorphism*)

Two objects are equal up to isomorphism when there exists an isomorphism between them.

Example 2.11 (*Identity as isomorphism*)

An object is equal up to isomorphism with itself because the identity arrow is an isomorphism with itself. In other words the following diagram trivially commutes

$$\begin{tikzcd} A \arrow[loop left, "\mathsf{id}"] \arrow[rr, bend left, "\mathsf{id}"] & & A \arrow[ll, bend left, "\mathsf{id}"] \arrow[loop right, "\mathsf{id}"] \end{tikzcd}$$

Remark. When talking about equality, the qualification "up to isomorphism" is often dropped.

2.2 Universal Constructions

Now that we know what a category is, we analyse the properties that these categories have. In category theory there exist constructions that occur in many different categories. We can use them to create universal structures occurring in many different categories.

2.2.1 Initial and Terminal Objects

Not only arrows have special properties like isomorphisms, there are also special types of objects. We analyse two of these, the initial object and its dualisation: the terminal object.

The notion of initial and terminal objects originates in initial and terminal algebras. It often refers to the smallest thing you can have (for initial objects) or the largest thing (for terminal objects). In category theory we call the initial object 0 and the terminal object 1.

Definition 2.12 (*Initial objects*)

An object 0 is initial iff for every object A there exists exactly one arrow from 0 to A.

Definition 2.13 (*Terminal objects*)

An object 1 is terminal iff for every object A there exists exactly one arrow from A to 1.

A lot of categories have initial and terminal objects even when this is not entirely obvious.

Example 2.14 (*Initial and terminal objects in Set*)

In the category **Set**, an initial object is $\{\}$, the empty set. This is the case because one can create only one function from the empty set to any other set, namely the empty function. A terminal object is $\{x\}$, a singleton set. This is the case because there exists only one total function from any set to a singleton set: $f : A \to \{x\}$ with: for all $a \in A$, $f(a) = x$.

Example 2.15 (*Initial and terminal objects in FPL*)

In the category **FPL**, () is a terminal object, as we can only define programs that return () for every input. The category **FPL** has no empty types, thus there are no initial objects.

Terminal objects are often used to define constants. This works because a terminal object has only one possible element. Thus, we can create exactly one arrow from a terminal object to any other element of any object. These arrows are thus used to reason about the contents of an object.

2.2.2 Products and Sums

Products

In both set theory and in functional programming languages, we have the notion of products. A product is a way to combine two sets or types into one. We can also express this in category theory. In category theory we reason about objects and arrows, categorical products should thus consist of these two things. A product in both sets and FPLs consists of a new set or type, often written as $A \times B$, and a pair of functions called the projections: $\mathsf{fst} : A \times B \to A$ and $\mathsf{snd} : A \times B \to B$. A product thus consists of an object and two arrows: $(A \times B, \mathsf{fst}, \mathsf{snd})$.

We can now combine arbitrary functions to create product functions. We can combine two functions, $f : C \to A$ and $g : C \to B$ using $\langle f, g \rangle : C \to A \times B$. This notation is defined as $\langle f, g \rangle(x) = \langle f(x), g(x) \rangle$. We can also get the original functions from $\langle f, g \rangle$, $\mathsf{fst} \circ \langle f, g \rangle = f$ and $\mathsf{snd} \circ \langle f, g \rangle = g$.

Definition 2.16 (*Products*)

A product of two objects A and B is the object $A \times B$ with two arrows $\mathsf{fst} : A \times B \to A$ and $\mathsf{snd} : A \times B \to B$, such that for any two arrows $f : C \to A$ and $g : C \to B$, there is exactly one arrow $\langle f, g \rangle$, called a projection arrow, fpr which that the following holds:

$$\mathsf{fst} \circ \langle f, g \rangle = f \qquad \text{(Product-}\textsc{Left}\text{)}$$
$$\mathsf{snd} \circ \langle f, g \rangle = g \qquad \text{(Product-}\textsc{Right}\text{)}$$

or in other words:

$$\begin{tikzcd} & C \arrow[dl, "f"'] \arrow[d, dashed, "\langle f,g \rangle"] \arrow[dr, "g"] & \\ A & A \times B \arrow[l, "\mathsf{fst}"] \arrow[r, "\mathsf{snd}"'] & B \end{tikzcd}$$

Using this definition, any object X that has two arrows $f : X \to A$ and $g : X \to B$ such that Product-Left and Product-Right hold, creates the product (X, f, g).

Lemma 2.17 (*Equality of products*)

Any two products of arbitrary objects A and B are equal up to isomorphism.

Proof. We take two arbitrary objects A and B and two arbitrary products on these objects (X, f_1, f_2) and (Y, g_1, g_2). Because X is a product and Y has arrows from Y to A and B, we have an arrow $\langle g_1, g_2 \rangle : Y \to X$, and the other way around, creating the arrow $\langle f_1, f_2 \rangle : X \to Y$.

Now to prove that $\langle g_1, g_2 \rangle$ and $\langle f_1, f_2 \rangle$ are an isomorphism, we want to prove that

$$\langle g_1, g_2 \rangle \circ \langle f_1, f_2 \rangle = \mathsf{id}_X$$

and that

$$\langle f_1, f_2 \rangle \circ \langle g_1, g_2 \rangle = \mathsf{id}_Y$$

We will prove the first statement by proving that, for $i \in \{1, 2\}$

$$f_i = f_i \circ \langle g_1, g_2\rangle \circ \langle f_1, f_2\rangle$$

Then following from the uniqueness of the projection arrow and the fact that id is also a valid projection arrow it follows that

$$\langle g_1, g_2\rangle \circ \langle f_1, f_2\rangle = \mathsf{id}$$

f_i
$=$ – Product-Left –
$g_i \circ \langle f_1, f_2\rangle$
$=$ – Product-Left –
$f_i \circ \langle g_1, g_2\rangle \circ \langle f_1, f_2\rangle$

$$\begin{tikzcd}
 & X \arrow[ddl, bend right, "f_1"'] \arrow[ddr, bend left, "f_2"] \arrow[d, dashed, "\langle f_1, f_2\rangle"] & \\
 & Y \arrow[dl, "g_1"'] \arrow[dr, "g_2"] \arrow[d, dashed, "\langle g_1, g_2\rangle"] & \\
A & X \arrow[l, "f_1"] \arrow[r, "f_2"] & B
\end{tikzcd}$$

The second statement is proven analogous by swapping X and Y and the associated functions. □

Example 2.18 (*Products in FPL*)

In **FPL** we have product types, often also called tuples. For any two types A and B we thus have the product $((A, B), \mathsf{fst}, \mathsf{snd})$.

There is also an operator that can be very useful.

Definition 2.19 ($\times$)

If there is an arrow $f : A \to B$ and an arrow $g : C \to D$, then there exists the arrow

$$f \times g : A \times C \to B \times D$$

This arrow is defined as

$$f \times g = \langle f \circ \mathsf{fst}, g \circ \mathsf{snd}\rangle \qquad \text{(Product-}\times\text{)}$$

Sums

When defining a new structure in category theory, there always exists a dual version of it. For products, the dual is sums. Instead of two objects combined in one, we have an object that is one of two objects.

A sum consists again of an object and two functions. This time, if we have an object called $A + B$, we need two function $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$.

Definition 2.20 (*Sums*)

A sum of two objects A and B, is an object $A + B$ with two arrows $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$, such that for any object C, if $f : A \to C$ and $g : B \to C$, the unique arrow $[f, g] : A + B \to C$, called the join, exists such that the following holds:

$$[f, g] \circ \mathsf{inl} = f \qquad \text{(Sum-Left)}$$
$$[f, g] \circ \mathsf{inr} = g \qquad \text{(Sum-Right)}$$

or in other words:

$$\begin{tikzcd}
A \arrow[r, "\mathsf{inl}"] \arrow[dr, "f"'] & A + B \arrow[d, dashed, "{[f,g]}"] & B \arrow[l, "\mathsf{inr}"'] \arrow[dl, "g"] \\
& C &
\end{tikzcd}$$

Lemma 2.21 (*Equality of sums*)

Any two sums of the same two objects are equal up to isomorphism.

Proof. This proof is dual to the proof of Lemma 2.17. □

Example 2.22 (*Sums in FPL*)

In **FPL** we also have sum types. Two often occurring ones are Eithers and Maybes. An Either is a sum with the two programs inl and inr: $(\mathsf{Either}\, A\ B, \mathsf{inl}, \mathsf{inr})$.

For the Maybe it is a bit less obvious why it is a sum. With a Maybe we either have a value of type A or we have no value represented by a constant. No value can be seen as a terminal object 1 because it can only have one value. So, a Maybe is actually a $1 + A$ sum. We can take the functions nothing and just, to complete our definitions: $(\mathsf{Maybe}\, A, \mathsf{nothing} : 1 \to \mathsf{Maybe}\, A, \mathsf{just} : A \to \mathsf{Maybe}\, A)$.

Of course there also exists a dual version of the $\times$ operator.

Definition 2.23 (+)

If there is an arrow $f : A \to B$ and an arrow $g : C \to D$, then there exists the arrow

$$f + g : A + C \to B + D$$

This arrow is defined as

$$f + g = [\mathsf{inl} \circ f, \mathsf{inr} \circ g] \qquad \text{(Sum-+)}$$

2.2.3 Exponential objects

We have seen how tuples and eithers are represented as categorical constructs, but there is one very important part of any functional programming language that we have not seen yet, arrow types or programs as objects. In a categorical sense, we get an object that is the collection of all arrows from B to A. The object containing all arrows $f : B \to A$ is called the exponential object A^B.

However, in category theory we do not use elements or contents of objects. We use arrows instead.

We can characterize an exponential object by a new special arrow $\mathsf{eval} : (A^B \times B) \to A$ that is defined as $\mathsf{eval}(f, b) = f(b)$. To create a proper categorical definition, we need one more arrow, for any arrow $g : (C \times B) \to A$, $\mathsf{curry}(g) : C \to A^B$. The arrow curry partially applies the first argument of the product to the arrow resulting in a new arrow, thus currying it. We will also write $\mathsf{curry}(g)(b)$ as $g_b : C \to A$.

Definition 2.24 (*Exponential object*)

For any two objects A and B, an object A^B is an exponential object if there is an arrow $\mathsf{eval} : A^B \times B \to A$ such that for any arrow $f : C \times B \to A$ there is a unique arrow $\mathsf{curry}(f) : C \to A^B$ where the following equality holds:

$$g = \mathsf{eval} \circ (\mathsf{curry}(g) \times \mathsf{id}) \qquad \text{(Exponential-}\textsc{Eval}\text{)}$$

Or the following diagram commutes:

$$\begin{tikzcd}
A^B \times B \arrow[r, "\mathsf{eval}"] & A \\
C \times B \arrow[u, dashed, "\mathsf{curry}(g) \times \mathsf{id}"] \arrow[ur, "g"'] &
\end{tikzcd}$$

Remark. Not all categories have exponential objects but the categories we mostly use, **Set** and **F-Alg** do.

Example 2.25 (const)

We have a function $\mathsf{const} : (A \times B) \to A$ that ignores the B it is given. We thus get the following diagram

$$\begin{tikzcd}
A^B \times B \arrow[rr, "\mathsf{eval}"] & & A \\
A \times B \arrow[u, dashed, "\mathsf{curry(const)}\times\mathsf{id}"] \arrow[urr, "\mathsf{const}"'] & &
\end{tikzcd}$$

We can now use a curried version of const to ignore incoming values

$$\mathsf{const}_1(5) = 1$$

Now that we have defined the most important universal constructs, we can use them in many of our definitions. However, not all categories have these constructs but the ones that do get a special name.

Definition 2.26 (*CCC*)

A Cartesian Closed Category (CCC) is a category that has terminal objects, products, sums and exponential objects.

Both **Set** and **FPL** are cartesian closed.

2.3 Functors

We have seen that several mathematical and programmatic structures behave like categories. However, there is one mathematical structure we have not yet tried to capture into a category, that is a categories of categories, we will call it **Cat**. In **Cat** our objects will be categories, our arrows will be structure preserving functions called functors.

To give more of an intuition about what functors are, we will start with an example:

Example 2.27 (*Maybe functor*)

We will start with the functor $\mathsf{Maybe} : \mathbf{FPL} \to \mathbf{FPL}$. Our functor **FPL** takes every object $A \in \mathbf{FPL}$ to the object $1 + A \in \mathbf{FPL}$. A function $f : A \to B$ will be taken to the function $[\mathsf{inl} \circ \mathsf{id}, \mathsf{inr} \circ f]$. Thus, $\mathsf{Maybe}(f) = \mathsf{id} + f$.

Now that we have a more intuitive understanding, we can define a functor:

Definition 2.28 (*Functor*)

A functor $\mathsf{F} : \mathbf{C} \to \mathbf{D}$ is a map taking an object A in $\mathbf{C}$ to an object B in $\mathbf{D}$: $\mathsf{F}(A) = B$, such that for any $f : A \to B$, $\mathsf{F}(f) : \mathsf{F}(A) \to \mathsf{F}(B)$. The functor of $f : A \to B$, $g : B \to C$ and id arrows should hold to the following laws

- The identity function is preserved through the map: for any $A \in \mathsf{Ob}_C$,
$$\mathsf{F}(\mathsf{id}_A) = \mathsf{id}_{F(A)} \tag{Functor-ID}$$
- Composition is preserved through the map: for any $A, B, C \in \mathsf{Ob}_C$ and $f : A \to B, g : B \to C, h : A \to C \in \mathsf{Arr}_C$,
$$g \circ f = h \to \mathsf{F}(g) \circ \mathsf{F}(f) = \mathsf{F}(h) \tag{Functor-Comp}$$

Thus, if the following diagram commutes

$$\begin{tikzcd} A \arrow[loop left, "\mathsf{id}"] \arrow[r, "f"] \arrow[rr, bend right, "h"'] & B \arrow[r, "g"] & C \end{tikzcd}$$

then so should this diagram

$$\begin{tikzcd} \mathsf{F}(A) \arrow[loop left, "\mathsf{F}(\mathsf{id})"] \arrow[r, "\mathsf{F}(f)"] \arrow[rr, bend right, "\mathsf{F}(h)"'] & \mathsf{F}(B) \arrow[r, "\mathsf{F}(g)"] & \mathsf{F}(C) \end{tikzcd}$$

Remark. We will often leave out the mapping of functions when defining a functor, this mapping of functions can be inferred from the mapping of objects that is done.

2.3.1 F-algebras

The notion of F-algebras originates in attempts to encode algebras in category theory.

For an algebra we need an object and a set of arrows. We will be using monoids as an example. For the object, we use Nat (the object of all natural numbers and arrows like the successor and addition). We also need two arrows for a monoid. A binary operation $\bullet : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}$ and a unit arrow $e : 1 \to \mathsf{Int}$ for the identity element. These are defined as, $\bullet = +$ and $e = 0$. Diagrammatically this is shown as follows

$$\begin{tikzcd} & \mathsf{Int} \times \mathsf{Int} \arrow[d, "\bullet"] \\ 1 \arrow[r, "e"] & \mathsf{Int} \end{tikzcd}$$

However, we can actually combine these functions into one object using sums.

$$\begin{tikzcd}
1 + \mathsf{Int} \times \mathsf{Int} \arrow[rrd, dashed, "{[e,\bullet]}"] & & \mathsf{Int} \times \mathsf{Int} \arrow[ll, "\mathsf{inl}"'] \arrow[d, "\bullet"] \\
1 \arrow[u, "\mathsf{inr}"] \arrow[rr, "e"] & & \mathsf{Int}
\end{tikzcd}$$

We now have an object that holds a monoid expression and an evaluation function $[e, \bullet]$ that evaluates it. The monoid expression object can be represented as a functor $\mathsf{M}(X) = 1 + X \times X$. We have now arrived at the essence of an F-Algebra: we have a functor M, an object Int and an evaluation function $[e, \bullet]$. This is denoted as the M-algebra $(\mathsf{Int}, [e, \bullet])$. An F-algebra is thus a way to represent taking a combination of objects to a single object. In terms of **FPL**, we have a structure containing elements of a specific type and combine that structure into a (new) element of that type.

Definition 2.29 (*F-Algebra*)

Given a functor $\mathsf{F} : \mathbf{C} \to \mathbf{C}$, an F-Algebra is a pair of $(A \in \mathsf{Ob}_C, \varphi : \mathsf{F}(A) \to A)$

$$\begin{tikzcd}
\mathsf{F}(A) \arrow[d, "\varphi"] \\
A
\end{tikzcd}$$

The notion of F-algebras ends up being surprisingly useful for not just evaluation type functions. To see just how useful they are, we will first apply one of the oldest tricks in category theory, try to create a category of it. We thus want to make a category of F-algebras:

Definition 2.30 (*Category of F-algebras*)

The category of F-algebras, **F-Alg** consists of:

- $\mathsf{Ob}_{\textbf{F-Alg}}$ contains F-algebras, (A, φ).
- $\mathsf{Arr}_{\textbf{F-Alg}}$ contains F-algebra homomorphisms. A homomorphism is an arrow that preserves structure. A homomorphism from F-algebra (A, φ) to F-algebra (B, ψ) is an arrow $f : A \to B$, such that

$$\psi \circ \mathsf{F}(f) = f \circ \varphi \qquad \text{(FAlg-Arrow)}$$

In other words the following diagram[4] should commute

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

The laws for a category hold as follows.

- If we have the arrows $f : (A, \varphi) \to (B, \psi), g : (B, \psi) \to (C, \chi) \in \mathsf{Arr}_{\textbf{F-Alg}}$, then we want to show that $g \circ f : (A, \varphi) \to (C, \chi)$ is an arrow in **F-Alg**. Thus, we want to show that

$$\mathsf{F}(f \circ g) \circ \chi = \varphi \circ f \circ g$$

Proof.

$\triangleright \quad \psi \circ \mathsf{F}(f) = f \circ \varphi$

$\blacktriangleright \quad \chi \circ \mathsf{F}(g) = g \circ \psi$

$\mathsf{F}(f \circ g) \circ \chi$

$=$ – Functor-Comp –

$\mathsf{F}(f) \circ \mathsf{F}(g) \circ \chi$

$=$ – Theorem 2.8 –

$\quad \psi \circ \mathsf{F}(f)$

$\quad =$ – $\triangleright$ –

$\quad f \circ \varphi$

$\quad \chi \circ \mathsf{F}(g)$

$\quad =$ – $\blacktriangleright$ –

$\quad g \circ \psi$

$\varphi \circ f \circ g$

$$\begin{tikzcd}
\mathsf{F}(A) \arrow[rr, bend left, "\mathsf{F}(f \circ g)"] \arrow[r, "\mathsf{F}(f)"] \arrow[d, "\varphi"] \arrow[dr, phantom, "\triangleright"] & \mathsf{F}(B) \arrow[r, "\mathsf{F}(g)"] \arrow[d, "\psi"] \arrow[dr, phantom, "\blacktriangleright"] & \mathsf{F}(C) \arrow[d, "\chi"] \\
A \arrow[r, "f"] \arrow[rr, bend right, "f \circ g"'] & B \arrow[r, "g"] & C
\end{tikzcd}$$

$\square$

- The associativity of arrows in **F-Alg** follows directly from the associativity of arrows in the domain and codomain of F.

- The identity arrow always exists because we can construct it as follows for any F-algebra $(A, \varphi) \in \mathsf{Ob}_{\textbf{F-Alg}}$:

$$\begin{tikzcd}
\mathsf{F}(A) \arrow[r, "\mathsf{F}(\mathsf{id}_A)=\mathsf{id}_{\mathsf{F}(A)}"] \arrow[d, "\varphi"] & \mathsf{F}(A) \arrow[d, "\varphi"] \\
A \arrow[r, "\mathsf{id}"] & A
\end{tikzcd}$$

Because $\mathsf{id} \circ \varphi = \varphi \circ \mathsf{id}$, this is an arrow in **F-Alg**.

Thus, we have defined the category of F-algebras.

We can now apply one of the universal constructions on this new category, Initial objects. The consequences of initial objects in F-algebras will kick-start our research into recursion schemes.

[4]We will be using a lot of diagrams in definitions and proofs associated with **F-Alg**, however do note that these diagrams are often in the domain and codomain of these functors and not in the actual category **F-Alg** itself. Also when describing an arrow $f : (A, \varphi) \to (B, \psi)$ this arrow is actually present in the domain of the functor.

Part II

Canned recursion[5]

[5]Meijer et al. [8] came up with the term canned induction. The term canned recursion is based on the same principle.

Chapter 3

Iteration, Co-Iteration and Primitive Recursion

The main ideas for iteration in a categorical context where first coined by Meijer et al. [8]. This paper has been the ground work for lots of other research into canned recursion. However, for this chapter we will be using the the PhD thesis by Vene [13] as the main source, the main results of which also appear in a paper by Uustalu and Vene [11]. Some proofs and examples that are given are our own.

3.1 Catamorphism

In this chapter, we will start of with iteration, e.g. recursion where we only get the result of the arrow one step back. When we apply this to the natural numbers, it results in a recursion scheme as follows:

$$\begin{aligned} f(0) &= c() \\ f(n+1) &= h(f(n)) \end{aligned}$$

Applying this to lists results in:

$$\begin{aligned} f(\mathsf{nil}()) &= c() \\ f(\mathsf{cons}(x, xs)) &= h(x, f(xs)) \end{aligned}$$

This kind of recursion follows very naturally from the category **F-Alg** when we look at initial objects in this category. We will first go through some lemmas and a complicated theorem, but then we will see its usefulness.

3.1.1 Initial F-algebras

An initial object in **F-Alg** should have a unique arrow from it to every object. Thus, an initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$ if it exists[1], should make the following diagram commute for any F-algebra (C, φ):

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, "\mathsf{F}(f)"] \arrow[d, "\mathsf{in}"] & \mathsf{F}(C) \arrow[d, "\varphi"] \\
\mu\mathsf{F} \arrow[r, dashed, "f"] & C
\end{tikzcd}$$

As this arrow is unique, it can only depend upon (C, φ), thus we will call this arrow $(\!|\varphi|\!)$[2]. This is called a catamorphism and the arrows are called banana's (named by Meijer et al. [8]). This arrow f has to be unique, thus the following should also hold

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

If we assume that $(\mu\mathsf{F}, \mathsf{in})$ is an initial F-algebra, then we can show that some very useful lemmas hold for this F-algebra:

Lemma 3.1 (*Cata-Self*)

For any F-algebra $(C, \varphi : \mathsf{F}(C) \to C)$ and initial F-algebra $(\mu\mathsf{F}, \mathsf{in} : \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F})$, we get the the catamorphism $(\!|\varphi|\!) : \mu\mathsf{F} \to C$ for which the following law holds

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

In other words, the following diagram commutes:

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

[1]Not every F-algebra category has initial objects. Any F-algebra where F is polynomial, built up from products, sums, Identity functors and constant functors, does however have an initial object. Any functors that we will use in this thesis will have an initial object.

[2]When applying the functor over a catamorphism we will often drop the brackets of the functor as they already look like brackets themselves.

Proof. This lemma follows directly from the definition of an arrow in **F-Alg**. □

Lemma 3.2 (*Cata-Repl*)

For any initial F*-algebra* $(\mu\mathsf{F}, \mathsf{in} : \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F})$ *we get a catamorphism from itself to itself,* $(\!|\mathsf{in}|\!)$*, for which the following holds*

$$\mathsf{id} = (\!|\mathsf{in}|\!) \qquad \text{(cata-Repl)}$$

In other words, the following diagrams commutes:

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, bend left, "\mathsf{F}(\mathsf{id})"] \arrow[r, bend right, "\mathsf{F}(\!|\mathsf{in}|\!)"'] \arrow[d, "\mathsf{in}"] & \mathsf{F}(\mu\mathsf{F}) \arrow[d, "\mathsf{in}"] \\
\mu\mathsf{F} \arrow[r, dashed, bend left, "\mathsf{id}"] \arrow[r, dashed, bend right, "(\!|\mathsf{in}|\!)"'] & \mu\mathsf{F}
\end{tikzcd}$$

Proof. We know that $\mathsf{id} \circ \mathsf{in} = \mathsf{in} \circ \mathsf{F}(\mathsf{id})$. But then following from cata-Charn with $\varphi = \mathsf{id}$, it follows that $\mathsf{id} = (\!|\mathsf{in}|\!)$. □

Lemma 3.3 (*Cata-Fusion*)

For any F*-algebras* $(C, \varphi : \mathsf{F}(C) \to C)$ *and* $(D, \psi : \mathsf{F}(D) \to D)$ *with arrow* $f : C \to D$ *and initial* F*-algebra* $(\mu\mathsf{F}, \mathsf{in} : \mathsf{F}(\mu\mathsf{F}) \to \mu\mathsf{F})$*, we get the two catamorphisms* $(\!|\varphi|\!) : \mu\mathsf{F} \to C$ *and* $(\!|\psi|\!) : \mu\mathsf{F} \to D$*, where*

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

In other words, the following diagram should commute:

$$\begin{tikzcd}
\mathsf{F}(D) \arrow[d, "\psi"] & \mathsf{F}(C) \arrow[l, "\mathsf{F}(f)"'] \arrow[d, "\varphi"'] & \mathsf{F}(\mu\mathsf{F}) \arrow[l, "\mathsf{F}(\llparenthesis\varphi\rrparenthesis)"'] \arrow[d, "\mathsf{in}"] \arrow[ll, bend right, "\mathsf{F}(\llparenthesis\psi\rrparenthesis)"'] \\
D & C \arrow[l, "f"'] & \mu\mathsf{F} \arrow[l, dashed, "\llparenthesis\varphi\rrparenthesis"'] \arrow[ll, dashed, bend left, "\llparenthesis\psi\rrparenthesis"']
\end{tikzcd}$$

Proof.

▷ $f \circ \varphi = \psi \circ \mathsf{F}(f)$

$f \circ \llparenthesis\varphi\rrparenthesis$

$=$ – cata-Charn –

$f \circ \llparenthesis\varphi\rrparenthesis \circ \mathsf{in}$

$=$ – FAlg-Arrow –

$f \circ \varphi \circ \mathsf{F}(\llparenthesis\varphi\rrparenthesis)$

$=$ – ▷ –

$\psi \circ \mathsf{F}(f) \circ \mathsf{F}(\llparenthesis\varphi\rrparenthesis)$

$=$ – Functor-Comp –

$\psi \circ \mathsf{F}(f \circ \llparenthesis\varphi\rrparenthesis)$

$\llparenthesis\psi\rrparenthesis$

$$\begin{tikzcd}
\mathsf{F}(D) \arrow[d, "\psi"] & \mathsf{F}(C) \arrow[l, "\mathsf{F}(f)"'] \arrow[d, "\varphi"'] & \mathsf{F}(\mu\mathsf{F}) \arrow[l, "\mathsf{F}(\llparenthesis\varphi\rrparenthesis)"'] \arrow[d, "\mathsf{in}"] \arrow[ll, bend right=20, "\mathsf{F}(f \circ \llparenthesis\varphi\rrparenthesis)"'] \arrow[ll, bend right=40, "\mathsf{F}(\llparenthesis\psi\rrparenthesis)"'] \\
D & C \arrow[l, "f"'] & \mu\mathsf{F} \arrow[l, dashed, "\llparenthesis\varphi\rrparenthesis"'] \arrow[ll, dashed, bend left, "\llparenthesis\psi\rrparenthesis"']
\end{tikzcd}$$

□

With these three lemmas we can prove the most important theorem of catamorphisms, Lambeks theorem. Lambeks theorem is about a very import property of any initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$, namely that in is an isomorphism.

Theorem 3.4 (*Lambek [6]*)

For the initial F*-algebra* $(\mu\mathsf{F}, \mathsf{in} : \mu\mathsf{F} \to \mathsf{F}\mu\mathsf{F})$, in *is an isomorphism with* in^{-1} *defined as:*

$$\mathsf{in}^{-1} = \llparenthesis\mathsf{F}(\mathsf{in})\rrparenthesis$$

Proof. We will have to show that the identity laws hold for in^{-1} and $\llparenthesis\mathsf{F}\,\mathsf{in}\rrparenthesis$. We start with the first one:

$$
\begin{aligned}
& \mathsf{in} \circ (\!|\mathsf{F}\ \mathsf{in}|\!) \\
= & \quad \text{– cata-Fusion –} \\
& \quad \left[\begin{aligned}
& \mathsf{in} \circ \mathsf{F}(\mathsf{in}) \\
= & \quad \text{– Equality of arrows –} \\
& \mathsf{in} \circ \mathsf{F}(\mathsf{in})
\end{aligned}\right. \\
& (\!|\mathsf{in}|\!) \\
= & \quad \text{– cata-Repl –} \\
& \mathsf{id}
\end{aligned}
$$

$$\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[d, "\mathsf{in}"] & \mathsf{F}(\mathsf{F}\mu\mathsf{F}) \arrow[l, "\mathsf{F}\ \mathsf{in}"'] \arrow[d, "\mathsf{F}\ \mathsf{in}"'] & \mathsf{F}\mu\mathsf{F} \arrow[l, "\mathsf{F}(\!|\mathsf{F}\ \mathsf{in}|\!)"'] \arrow[d, "\mathsf{in}"] \arrow[ll, bend right, "\mathsf{F}(\!|\mathsf{in}|\!)"'] \\
\mu\mathsf{F} & \mathsf{F}\mu\mathsf{F} \arrow[l, "\mathsf{in}"'] & \mu\mathsf{F} \arrow[l, "(\!|\mathsf{F}\ \mathsf{in}|\!)"'] \arrow[ll, bend left, "(\!|\mathsf{in}|\!)"] \arrow[ll, bend left=60, "\mathsf{id}"]
\end{tikzcd}$$

Thus, $\mathsf{in} \circ (\!|\mathsf{F}\ \mathsf{in}|\!) = \mathsf{id}$
Now we will prove it for the other equation:

$$\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[d, "\mathsf{in}"] & \mathsf{F}(\mathsf{F}\mu\mathsf{F}) \arrow[l, "\mathsf{F}\ \mathsf{in}"'] \arrow[d, "\mathsf{F}\ \mathsf{in}"'] & \mathsf{F}\mu\mathsf{F} \arrow[l, "\mathsf{F}(\!|\mathsf{F}\ \mathsf{in}|\!)"'] \arrow[d, "\mathsf{in}"] \arrow[ll, bend right, "\mathsf{F}(\!|\mathsf{in}|\!)"'] \\
\mu\mathsf{F} & \mathsf{F}\mu\mathsf{F} \arrow[l, "\mathsf{in}"'] & \mu\mathsf{F} \arrow[l, "(\!|\mathsf{F}\ \mathsf{in}|\!)"'] \arrow[ll, bend left, "(\!|\mathsf{in}|\!)"] \arrow[ll, bend left=60, "\mathsf{id}"]
\end{tikzcd}$$

$$
\left[\begin{aligned}
& (\!|\mathsf{F}\ \mathsf{in}|\!) \circ \mathsf{in} \\
= & \quad \text{– cata-Self –} \\
& \mathsf{F}\ \mathsf{in} \circ \mathsf{F}(\!|\mathsf{F}\ \mathsf{in}|\!) \\
= & \quad \text{– F-functor –} \\
& \mathsf{F}(\mathsf{in} \circ (\!|\mathsf{F}\ \mathsf{in}|\!)) \\
= & \quad \text{– see above –} \\
& \mathsf{F}\ \mathsf{id} \\
= & \quad \text{– F-functor –} \\
& \mathsf{id}
\end{aligned}\right.
$$

$$\mathsf{F}\left(\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[d, "\mathsf{in}"] & \mathsf{F}(\mathsf{F}\mu\mathsf{F}) \arrow[l, "\mathsf{F}\ \mathsf{in}"'] \arrow[d, "\mathsf{F}\ \mathsf{in}"'] & \mathsf{F}\mu\mathsf{F} \arrow[l, "\mathsf{F}(\!|\mathsf{F}\ \mathsf{in}|\!)"'] \arrow[d, "\mathsf{in}"] \arrow[ll, bend right, "\mathsf{F}(\!|\mathsf{in}|\!)"'] \\
\mu\mathsf{F} & \mathsf{F}\mu\mathsf{F} \arrow[l, "\mathsf{in}"'] & \mu\mathsf{F} \arrow[l, "(\!|\mathsf{F}\ \mathsf{in}|\!)"'] \arrow[ll, bend left, "(\!|\mathsf{in}|\!)"] \arrow[ll, bend left=60, "\mathsf{id}"]
\end{tikzcd}\right)$$

Thus, for any initial F-algebra in^{-1} always exists and is $(\!|\mathsf{F}\,\mathsf{in}|\!)$. $\square$

From this theorem it follows that for any initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$ in **F-Alg**, $\mu\mathsf{F}$ is a fixed point up to isomorphism under application of in. In other words, we can switch between functor and non-functor representation of an object. Thus, an initial F-algebra is often an object that consists of some combination of itself, for example a binary tree that consists of either a leaf or two more binary trees.

Now that we proved all these lemmas and theorems we can put them to good use. We can use Lambeks theorem to redefine our catamorphism in terms of itself:

$$(\!|\varphi|\!) = \varphi \circ \mathsf{F}(\!|\varphi|\!) \circ \mathsf{in}^{-1}$$

In other words, the following diagram commutes:

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, dashed, "\mathsf{F}(\!|\varphi|\!)"] & \mathsf{F}(C) \arrow[d, "\varphi"] \\
\mu\mathsf{F} \arrow[u, "\mathsf{in}^{-1}"] \arrow[r, dashed, "(\!|\varphi|\!)"] & C
\end{tikzcd}$$

This is now a proper recursive definition of $(\!|\varphi|\!)$ that we can implement and calculate.

3.1.2 Application

Now that we have a proper definition of $(\!|\varphi|\!)$, we can utilise it. We will start by showing how we can apply the theory on numbers:

The Nat object

Let us sy we have an object Nat that repres the natural numbers in any Cartesian Closed Category **C**, where such an object object can exist.

Definition 3.5 (Nat *object*)

An object is a Nat object if it has the following arrows: $\mathsf{zero} : 1 \to \mathsf{Nat}$, $\mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}$ and $\mathsf{prev} : \mathsf{Nat} \to 1 + \mathsf{Nat}$ where:

$$\mathsf{prev} \circ \mathsf{zero} = \mathsf{inl}$$
$$\mathsf{prev} \circ \mathsf{succ} = \mathsf{inr}$$

such that

$$[\mathsf{zero}, \mathsf{succ}] : 1 + \mathsf{Nat} \to \mathsf{Nat}$$

is an isomorphism with the inverse prev, thus

$$[\mathsf{zero}, \mathsf{succ}]^{-1} = \mathsf{prev}$$

Or the following diagram commutes:

$$\begin{tikzcd}
1 \arrow[r, "\mathsf{zero}"] \arrow[dr, bend right, "\mathsf{inl}"'] & \mathsf{Nat} \arrow[d, bend left, "\mathsf{prev}"] & \mathsf{Nat} \arrow[l, "\mathsf{succ}"'] \arrow[dl, bend left, "\mathsf{inl}"] \\
& 1 + \mathsf{Nat} \arrow[u, bend left, "{[\mathsf{zero},\mathsf{succ}]}"] &
\end{tikzcd}$$

Example 3.6 (Nat *object in Set*)

The Nat object in **Set** is quite intuitive, as object we take the set $\mathbf{N}$. zero is a function from $\{x\}$ to $\mathbf{N}$ where

$$\mathsf{zero}(x) = 0$$

and succ is a function from $\mathbf{N}$ to $\mathbf{N}$ where

$$\mathsf{succ}(n) = n + 1$$

prev is a function from $\mathbf{N}$ to $\mathbf{N} \cup \{x\}$ where x is not a number. prev is defined as

$$\mathsf{prev}(n) = \begin{cases} x & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

The initial object of **N-Alg** where $\mathsf{N}(X) = 1 + X$ happens to have these properties.

Lemma 3.7 (*Initial* N*-algebra*)

*The initial object of **N-Alg** is a* Nat *object.*

Proof. The initial object of **N-Alg** is the N-algebra $(\mu\mathsf{N}, \mathsf{in})$. Here, in is of the type $1 + \mu\mathsf{N} \to \mu\mathsf{N}$, thus in is of the form $[c : 1 \to \mu\mathsf{N}, h : \mu\mathsf{N} \to \mu\mathsf{N}]$. We take $c = \mathsf{zero}$ and $h = \mathsf{succ}$.

That $[\mathsf{zero}, \mathsf{succ}]$ is an isomorphism with $[\mathsf{zero}, \mathsf{succ}]^{-1} = (\!|\mathsf{id} + [\mathsf{zero}, \mathsf{succ}]|\!)$ follows directly from Lambeks theorem (Theorem 3.4).

We then have to prove the two laws for Nat objects

$$(\!|\mathsf{id} + [\mathsf{zero}, \mathsf{succ}]|\!) \circ \mathsf{zero} = \mathsf{inl}$$
$$(\!|\mathsf{id} + [\mathsf{zero}, \mathsf{succ}]|\!) \circ \mathsf{succ} = \mathsf{inr}$$

$$
\begin{aligned}
& \llparenthesis \mathsf{id} + [\mathsf{zero}, \mathsf{succ}] \rrparenthesis \circ \mathsf{zero} \\
= & \quad \text{– Sum-Left –} \\
& \llparenthesis \mathsf{id} + [\mathsf{zero}, \mathsf{succ}] \rrparenthesis \circ [\mathsf{zero}, \mathsf{succ}] \circ \mathsf{inl} \\
= & \quad \text{– Isomorphism –} \\
& \mathsf{id} \circ \mathsf{inl} \\
= & \quad \text{– Identity-Left –} \\
& \mathsf{inl}
\end{aligned}
$$

$$
\begin{aligned}
& \llparenthesis \mathsf{id} + [\mathsf{zero}, \mathsf{succ}] \rrparenthesis \circ \mathsf{succ} \\
= & \quad \text{– Sum-Right –} \\
& \llparenthesis \mathsf{id} + [\mathsf{zero}, \mathsf{succ}] \rrparenthesis \circ [\mathsf{zero}, \mathsf{succ}] \circ \mathsf{inr} \\
= & \quad \text{– Isomorphism –} \\
& \mathsf{id} \circ \mathsf{inr} \\
= & \quad \text{– Identity-Left –} \\
& \mathsf{inr}
\end{aligned}
$$

$$\begin{tikzcd}
1 + \mathsf{Nat} \arrow[r, "\mathsf{id} + \llparenthesis \mathsf{id} + [\mathsf{zero},\mathsf{succ}] \rrparenthesis"] \arrow[dd, bend right=40, "{[\mathsf{zero},\mathsf{succ}]}"] & 1 + (1 + \mathsf{Nat}) \arrow[dd, "\mathsf{id} + [\mathsf{zero},\mathsf{succ}]"] \\
\mathsf{Nat} \arrow[d, "\mathsf{succ}"] \arrow[rd, bend left, "\mathsf{inr}"] & \\
\mathsf{Nat} \arrow[r, bend left, "\llparenthesis \mathsf{id} + [\mathsf{zero},\mathsf{succ}] \rrparenthesis"] & 1 + \mathsf{Nat} \arrow[l, bend left, "{[\mathsf{zero},\mathsf{succ}]}"] \arrow[loop right, "\mathsf{id}"] \\
1 \arrow[u, "\mathsf{zero}"] \arrow[ru, bend right, "\mathsf{inl}"'] &
\end{tikzcd}$$

Thus, the initial N-algebra is $(\mathsf{Nat}, [\mathsf{zero}, \mathsf{succ}])$. $\square$

Now that we have an initial N-algebra, we can put it to use by defining a few arrows. Let us start with some arrows that take the product of two Nat objects and end up in one.

Example 3.8 (*Add*)

We would like to define an arrow $\mathsf{add} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$, that adds the two numbers together using only the arrows we have defined so far. Let us start by giving the recursive equations of add:

$$\mathsf{add}_y \circ \mathsf{zero} = \mathsf{const}_y$$
$$\mathsf{add}_y \circ \mathsf{succ} = \mathsf{succ} \circ \mathsf{add}_y$$

To create a catamorphism, we need an N-algebra (Nat, φ) such that $\llparenthesis \varphi \rrparenthesis = \mathsf{add}$. We thus need to define a $\varphi : 1 + \mathsf{Nat} \to \mathsf{Nat}$ such that 1 gives back our y and Nat gives back the successor:

$$\mathsf{add}_y = \llparenthesis [\mathsf{const}_y, \mathsf{succ}] \rrparenthesis$$

With the following diagram for the catamorphism:

$$\begin{tikzcd}
1 + \mathsf{Nat} \arrow[r, "\mathsf{id} + \llparenthesis [\mathsf{const}_y,\mathsf{succ}] \rrparenthesis"] \arrow[d, "{[\mathsf{zero},\mathsf{succ}]}"] & 1 + \mathsf{Nat} \arrow[d, "{[\mathsf{const}_y,\mathsf{succ}]}"] \\
\mathsf{Nat} \arrow[r, "\llparenthesis [\mathsf{const}_y,\mathsf{succ}] \rrparenthesis"] & \mathsf{Nat}
\end{tikzcd}$$

We can also define the multiplication arrow $\mathsf{mul} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$ in the same manner:

$$\mathsf{mul}(x, y) = (\![\mathsf{zero}, \mathsf{add} \circ \langle \mathsf{id}, \mathsf{const}_y \rangle]\!)(x)$$

But we can also create catamorphisms to any other N-algebra we can think of.

Example 3.9 (*Boolean catamorphism*)

Let us say there exists an object Bool, encoding true false values, that has the following three arrows: $\mathsf{true} : 1 \to \mathsf{Bool}$, $\mathsf{false} : 1 \to \mathsf{Bool}$ and $\mathsf{not} : \mathsf{Bool} \to \mathsf{Bool}$. They are defined as follows:

$$\mathsf{not} \circ \mathsf{true} = \mathsf{false}$$
$$\mathsf{not} \circ \mathsf{false} = \mathsf{true}$$

Now any N-algebra we create with Bool is also a catamorphism from Nat to Bool. Lets say we have the N-algebra $(\mathsf{Bool}, [\mathsf{true}, \mathsf{not}])$. The catamorphism from Nat to Bool we can now create is:

$$(\![\mathsf{true}, \mathsf{not}]\!)$$

with the diagram:

$$\begin{tikzcd}
1 + \mathsf{Nat} \arrow[r, "\mathsf{id} + (\![\mathsf{true},\mathsf{not}]\!)"] \arrow[d, "{[\mathsf{zero},\mathsf{succ}]}"] & 1 + \mathsf{Bool} \arrow[d, "{[\mathsf{true},\mathsf{not}]}"] \\
\mathsf{Nat} \arrow[r, "(\![\mathsf{true},\mathsf{not}]\!)"] & \mathsf{Bool}
\end{tikzcd}$$

This is actually the arrow isEven, that returns true if the value is even and false otherwise.

In general, if we have a set of recursive equations of the following form with $c : 1 \to A$ and $h : A \to A$ such that

$$f \circ \mathsf{zero} = c$$
$$f \circ \mathsf{succ} = h \circ f$$

Then we can create a catamorphism for f as follows:

$$f = (\![c, h]\!)$$

The List object

We will now assume we have a List_A object in a category $\mathbf{C}$ that represents a list of things of object A.

Definition 3.10 (List *object*)

An object is a List object if it has the following arrows: $\mathsf{nil} : 1 \to \mathsf{List}_A$, $\mathsf{cons} : A \times \mathsf{List}_A \to \mathsf{List}_A$, $\mathsf{head} : \mathsf{List}_A \to 1 + A$ and $\mathsf{tail} : \mathsf{List}_A \to 1 + \mathsf{List}_A$, where:

$$\begin{aligned}
\mathsf{head} \circ \mathsf{nil} &= \mathsf{inl} \\
\mathsf{tail} \circ \mathsf{nil} &= \mathsf{inl} \\
\mathsf{head} \circ \mathsf{cons} &= \mathsf{inr} \circ \mathsf{fst} \\
\mathsf{tail} \circ \mathsf{cons} &= \mathsf{inr} \circ \mathsf{snd}
\end{aligned}$$

Or the following diagram commutes:

$$\begin{tikzcd}
 & 1 + \mathsf{List}_A & \mathsf{List}_A \arrow[l, "\mathsf{inr}"'] \\
1 \arrow[ur, "\mathsf{inl}"] \arrow[r, "\mathsf{nil}"] \arrow[dr, "\mathsf{inl}"'] & \mathsf{List}_A \arrow[u, "\mathsf{tail}"] \arrow[d, "\mathsf{head}"] & A \times \mathsf{List}_A \arrow[l, "\mathsf{cons}"'] \arrow[u, "\mathsf{snd}"'] \arrow[d, "\mathsf{fst}"] \\
 & 1 + A & A \arrow[l, "\mathsf{inr}"']
\end{tikzcd}$$

And

$$[\mathsf{nil}, \mathsf{cons}] : 1 + A \times \mathsf{List}_A \to \mathsf{List}_A$$

is an isomorphism where the following holds for the inverse:

$$\begin{aligned}
(\mathsf{id} + \mathsf{fst}) \circ [\mathsf{nil}, \mathsf{cons}]^{-1} &= \mathsf{head} \\
(\mathsf{id} + \mathsf{snd}) \circ [\mathsf{nil}, \mathsf{cons}]^{-1} &= \mathsf{tail}
\end{aligned}$$

Thus, the following diagram commutes

$$\begin{tikzcd}
 & 1 + A \times \mathsf{List}_A \arrow[dl, bend right, "(\mathsf{id}+\mathsf{fst})"'] \arrow[dr, bend left, "(\mathsf{id}+\mathsf{snd})"] \arrow[d, bend left, "{[\mathsf{nil},\mathsf{cons}]}"] & \\
1 + A & \mathsf{List}_A \arrow[l, "\mathsf{tail}"] \arrow[r, "\mathsf{head}"'] \arrow[u, bend left, "{[\mathsf{nil},\mathsf{cons}]^{-1}}"] & 1 + \mathsf{List}_A
\end{tikzcd}$$

Example 3.11 (List *object in FPL and Set*)

A List object in **FPL** is the type of a list containing objects of type A with the necessary programs. In **Set**, the List_A object is the set of all finite tuples of elements of the set A. The functions are easily derived from this representation.

We can now find our Nat object in **F-Alg** for some F.

Lemma 3.12 (*Initial* List *object*)

The initial object of $\mathbf{L}_A$*-Alg with* $\mathsf{L}_A(X) = 1 + A \times X$ *is a* List *object.*

Proof. The initial object of $\mathbf{L}_A$**-Alg** is $(\mu\mathsf{L}, \mathsf{in} : 1 + A \times \mu\mathsf{L} \to \mu\mathsf{L})$. Thus, in is of the form $[c : 1 \to \mu\mathsf{L}, h : A \times \mu\mathsf{L} \to \mu\mathsf{L}]$. We will take $c = \mathsf{nil}$ and $h = \mathsf{cons}$.

That $[\mathsf{nil}, \mathsf{cons}]$ is an isomorphism with $[\mathsf{nil}, \mathsf{cons}]^{-1} = (\!|\mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}])|\!)$ follows from Lambeks theorem (Theorem 3.4).

We then have to prove that

$$\begin{aligned}
(\mathsf{id} + \mathsf{fst}) \circ (\!|\mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}])|\!) \circ \mathsf{nil} &= \mathsf{inl} \\
(\mathsf{id} + \mathsf{snd}) \circ (\!|\mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}])|\!) \circ \mathsf{nil} &= \mathsf{inl} \\
(\mathsf{id} + \mathsf{fst}) \circ (\!|\mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}])|\!) \circ \mathsf{cons} &= \mathsf{inr} \circ \mathsf{fst} \\
(\mathsf{id} + \mathsf{snd}) \circ (\!|\mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}])|\!) \circ \mathsf{cons} &= \mathsf{inr} \circ \mathsf{snd}
\end{aligned}$$

$$
\begin{array}{ll}
 & (\mathsf{id} + \mathsf{fst}) \circ \llparenthesis \mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}]) \rrparenthesis \circ \mathsf{nil} \\
= & \quad - \text{ Sum-Left } - \\
 & (\mathsf{id} + \mathsf{fst}) \circ \llparenthesis \mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}]) \rrparenthesis \circ [\mathsf{nil}, \mathsf{cons}] \circ \mathsf{inl} \\
= & \quad - \text{ Isomorphism } - \\
 & (\mathsf{id} + \mathsf{fst}) \circ \mathsf{id} \circ \mathsf{inl} \\
= & \quad - \text{ Identity-Left, Sum-+ } - \\
 & [\mathsf{inl} \circ \mathsf{id}, \mathsf{inr} \circ \mathsf{fst}] \circ \mathsf{inl} \\
= & \quad - \text{ Sum-Left } - \\
 & \mathsf{inl} \circ \mathsf{id} \\
= & \quad - \text{ Identity-Left } - \\
 & \mathsf{inl}
\end{array}
$$

$$
\begin{array}{ll}
 & (\mathsf{id} + \mathsf{snd}) \circ \llparenthesis \mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}]) \rrparenthesis \circ \mathsf{nil} \\
= & \quad - \text{ Sum-Left } - \\
 & (\mathsf{id} + \mathsf{snd}) \circ \llparenthesis \mathsf{id} + (\mathsf{id} \times [\mathsf{nil}, \mathsf{cons}]) \rrparenthesis \circ [\mathsf{nil}, \mathsf{cons}] \circ \mathsf{inl} \\
= & \quad - \text{ Isomorphism } - \\
 & (\mathsf{id} + \mathsf{snd}) \circ \mathsf{id} \circ \mathsf{inl} \\
= & \quad - \text{ Identity-Left, Sum-+ } - \\
 & [\mathsf{inl} \circ \mathsf{id}, \mathsf{inr} \circ \mathsf{snd}] \circ \mathsf{inl} \\
= & \quad - \text{ Sum-Left } - \\
 & \mathsf{inl} \circ \mathsf{id} \\
= & \quad - \text{ Identity-Left } - \\
 & \mathsf{inl}
\end{array}
$$

$$\begin{tikzcd}
1 + A \times \mathsf{List}_A \arrow[r, equal] \arrow[d, "\mathsf{id} + \mathsf{fst}"] & 1 + A \times \mathsf{List}_A \arrow[r, equal] \arrow[d, bend left, "{[\mathsf{nil},\mathsf{cons}]}"] & 1 + A \times \mathsf{List}_A \arrow[d, "\mathsf{id} + \mathsf{snd}"'] \\
1 + A & \mathsf{List}_A \arrow[l, "\mathsf{head}"] \arrow[r, "\mathsf{tail}"] \arrow[u, bend left, "\llparenthesis \mathsf{id} + (\mathsf{id} \times [\mathsf{nil},\mathsf{cons}]) \rrparenthesis"] & 1 + \mathsf{List}_A \\
1 \arrow[r, equal] \arrow[uu, bend left, "\mathsf{inl}"] & 1 \arrow[u, "\mathsf{nil}"] \arrow[ul, "\mathsf{inl}"] \arrow[ur, "\mathsf{inl}"']
\end{tikzcd}$$

$$\left[\begin{aligned}
& (\mathsf{id}+\mathsf{fst}) \circ (\!|\mathsf{id}+(\mathsf{id}\times[\mathsf{nil},\mathsf{cons}])|\!) \circ \mathsf{cons} \\
= & \quad \text{– Sum-}\textsc{Right}\text{ –} \\
& (\mathsf{id}+\mathsf{fst}) \circ (\!|\mathsf{id}+(\mathsf{id}\times[\mathsf{nil},\mathsf{cons}])|\!) \circ [\mathsf{nil},\mathsf{cons}] \circ \mathsf{inr} \\
= & \quad \text{– Isomorphism –} \\
& (\mathsf{id}+\mathsf{fst}) \circ \mathsf{id} \circ \mathsf{inr} \\
= & \quad \text{– Identity-}\textsc{Left}\text{, Sum-+ –} \\
& [\mathsf{inl} \circ \mathsf{id}, \mathsf{inr} \circ \mathsf{fst}] \circ \mathsf{inr} \\
= & \quad \text{– Sum-}\textsc{Right}\text{ –} \\
& \mathsf{inr} \circ \mathsf{fst}
\end{aligned}\right.$$

$$\left[\begin{aligned}
& (\mathsf{id}+\mathsf{snd}) \circ (\!|\mathsf{id}+(\mathsf{id}\times[\mathsf{nil},\mathsf{cons}])|\!) \circ \mathsf{cons} \\
= & \quad \text{– Sum-}\textsc{Right}\text{ –} \\
& (\mathsf{id}+\mathsf{snd}) \circ (\!|\mathsf{id}+(\mathsf{id}\times[\mathsf{nil},\mathsf{cons}])|\!) \circ [\mathsf{nil},\mathsf{cons}] \circ \mathsf{inr} \\
= & \quad \text{– Isomorphism –} \\
& (\mathsf{id}+\mathsf{snd}) \circ \mathsf{id} \circ \mathsf{inr} \\
= & \quad \text{– Identity-}\textsc{Left}\text{, Sum-+ –} \\
& [\mathsf{inl} \circ \mathsf{id}, \mathsf{inr} \circ \mathsf{snd}] \circ \mathsf{inr} \\
= & \quad \text{– Sum-}\textsc{Right}\text{ –} \\
& \mathsf{inr} \circ \mathsf{snd}
\end{aligned}\right.$$

$$\begin{tikzcd}
1+A\times\mathsf{List}_A \arrow[r, equal] \arrow[d, "\mathsf{id}+\mathsf{fst}"] & 1+A\times\mathsf{List}_A \arrow[r, equal] \arrow[d, bend left, "{[\mathsf{nil},\mathsf{cons}]}"] & 1+A\times\mathsf{List}_A \arrow[d, "\mathsf{id}+\mathsf{snd}"'] \\
1+A & \mathsf{List}_A \arrow[l, "\mathsf{head}"'] \arrow[r, "\mathsf{tail}"] \arrow[u, bend left, "(\!|\mathsf{id}+(\mathsf{id}\times[\mathsf{nil},\mathsf{cons}])|\!)"] & 1+\mathsf{List}_A \\
A \arrow[u, "\mathsf{inr}"] & \mathsf{List}_A \arrow[ur, "\mathsf{inr}"] & A\times\mathsf{List}_A \arrow[l, "\mathsf{snd}"] \arrow[ul, "\mathsf{cons}"'] \arrow[ll, bend left, "\mathsf{fst}"] \arrow[u, bend right=60, "\mathsf{inr}"']
\end{tikzcd}$$

Thus, the initial L-algebra for $\mathbf{L}_A$**-Alg** is $(\mathsf{List}, [\mathsf{nil}, \mathsf{succ}])$. □

Now that we have an initial L-algebra we can put it to use by defining a few arrows. We will start with some arrows on lists of numbers.

Example 3.13 (***Sum***)

We would like to take the sum of a list of numbers. The recursive equations defined for $\mathsf{sum} : \mathsf{List}_{\mathsf{Nat}} \to \mathsf{Nat}$:

$$\begin{aligned} \mathsf{sum} \circ \mathsf{nil} &= \mathsf{zero} \\ \mathsf{sum} \circ \mathsf{cons} &= \mathsf{add} \circ (\mathsf{id} \times \mathsf{sum}) \end{aligned}$$

To create sum as a catamorphism we need to create a L_A-algebra $(\mathsf{Nat}, \varphi : 1+\mathsf{Nat}\times\mathsf{Nat} \to \mathsf{Nat})$ such that $(\!|\varphi|\!) = \mathsf{sum}$. φ has to consist of a join, $[c, h]$, thus we have to create two arrows $c : 1 \to \mathsf{Nat}$ and $h : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat}$. Because c is the case $\mathsf{sum} \circ \mathsf{nil}$ and h is the case $\mathsf{sum} \circ \mathsf{cons}$, we can derive both from the recursive equations:

$$\begin{aligned} c &= \mathsf{zero} \\ h &= \mathsf{add} \end{aligned}$$

Thus, we have defined φ and we can define sum:

$$\mathsf{sum} = (\!|[\mathsf{zero}, \mathsf{sum}]|\!)$$

With the following diagram:

$$\begin{tikzcd}
1 + \mathsf{Nat} \times \mathsf{List}_{\mathsf{Nat}} \arrow[r, "\mathsf{id} + (\mathsf{id} \times \llparenthesis[\mathsf{zero},\mathsf{sum}]\rrparenthesis)"] \arrow[d, "{[\mathsf{nil},\mathsf{cons}]}"] & 1 + \mathsf{Nat} \times \mathsf{Nat} \arrow[d, "{[\mathsf{zero},\mathsf{sum}]}"] \\
\mathsf{List}_{\mathsf{Nat}} \arrow[r, dashed, "\llparenthesis[\mathsf{zero},\mathsf{sum}]\rrparenthesis"] & \mathsf{Nat}
\end{tikzcd}$$

However, there are also several arrows we can define for a list with an arbitrary carrier.

Example 3.14 (*Length*)

We will define the arrow $\mathsf{length} : \mathsf{List}_A \to \mathsf{Nat}$, this arrow will count the amount of elements in the list. This arrows has the following recursive equations:

$$\begin{aligned}
\mathsf{length} \circ \mathsf{nil} &= \mathsf{zero} \\
\mathsf{length} \circ \mathsf{cons} &= \mathsf{succ} \circ \mathsf{snd} \circ (\mathsf{id} \times \mathsf{length})
\end{aligned}$$

The catamorphism associated with it is

$$\mathsf{length} = \llparenthesis[\mathsf{zero}, \mathsf{succ} \circ \mathsf{snd}]$$

The diagram is

$$\begin{tikzcd}
1 + \mathsf{Nat} \times \mathsf{List}_A \arrow[r, "\mathsf{id} + (\mathsf{id} \times \llparenthesis[\mathsf{zero},\mathsf{succ} \circ \mathsf{snd}]\rrparenthesis)"] \arrow[d, "{[\mathsf{nil},\mathsf{cons}]}"] & 1 + \mathsf{Nat} \times \mathsf{Nat} \arrow[d, "{[\mathsf{zero},\mathsf{succ} \circ \mathsf{snd}]}"] \\
\mathsf{List}_A \arrow[r, dashed, "\llparenthesis[\mathsf{zero},\mathsf{succ} \circ \mathsf{snd}]\rrparenthesis"] & \mathsf{Nat}
\end{tikzcd}$$

Example 3.15 (*Map*)

We will define the arrow $\mathsf{map} : B^A \times \mathsf{List}_A \to \mathsf{List}_B$, this arrow will apply another arrow to every element of the list. This arrow has the following recursive equations for an arrow $f : A \to B$:

$$\begin{aligned}
\mathsf{map}_f \circ \mathsf{nil} &= \mathsf{nil} \\
\mathsf{map}_f \circ \mathsf{cons} &= \mathsf{cons} \circ (f \times \mathsf{id}) \circ (\mathsf{id} \times \mathsf{map}_f)
\end{aligned}$$

We will be defining a catamorphism for the partially applied version of map, map_f with $f : A \to B$. To define this catamorphism we need to find a suitable L_A-algebra $(\mathsf{List}_B, \varphi : 1 + A \times \mathsf{List}_B)$. As previously we need to do a join for φ, $[c, h]$. We can then derive c and h from the recursive equations:

$$c = \mathsf{nil}$$
$$h = \mathsf{cons} \circ (f \times \mathsf{id})$$

Thus, we have defined φ and we can give the definition of map_f:

$$\mathsf{map}_f = (\!|[\mathsf{nil}, \mathsf{cons} \circ (f \times \mathsf{id})]|\!)$$

with the following diagram:

$$\begin{tikzcd}
1 + \mathsf{Nat} \times \mathsf{List}_A \arrow[r, "\mathsf{id} + (\mathsf{id} \times (\!|[\mathsf{nil},\mathsf{cons} \circ (f \times \mathsf{id})]|\!))"] \arrow[d, "{[\mathsf{nil},\mathsf{cons}]}"] & 1 + \mathsf{Nat} \times \mathsf{List}_B \arrow[d, "{[\mathsf{nil},\mathsf{cons} \circ (f \times \mathsf{id})]}"] \\
\mathsf{List}_A \arrow[r, dashed, "(\!|[\mathsf{nil},\mathsf{cons} \circ (f \times \mathsf{id})]|\!)"] & \mathsf{List}_B
\end{tikzcd}$$

For List and L we can also create a general structure. If we have a set of recursive equations with $c : 1 \to B$ and $h : A \times B \to B$

$$f \circ \mathsf{nil} = c$$
$$f \circ \mathsf{cons} = h \circ (\mathsf{id} \times f)$$

Then we get the following catamorphism:

$$f = (\!|[c, h]|\!)$$

3.2 Anamorphism

Just as we have seen with initial objects and products, we can also look at the dual version of the catamorphism. However, to reach that point we will first have to cover a few subjects that support it.

3.2.1 Terminal F-coalgebras

F-coalgebras are F-algebras but with the arrow reversed.

Definition 3.16 (*F-coalgebra*)

Given a functor $\mathsf{F} : \mathbf{C} \to \mathbf{C}$, an F-coalgebra is a pair of $(A, \varphi : A \to \mathsf{F}(A))$

$$\begin{tikzcd} A \arrow[d, "\varphi"] \\ \mathsf{F}(A) \end{tikzcd}$$

We can again create a category of F-coalgebras.

Definition 3.17 (*Category of F-coalgebras*)

The category of F-coalgebras, **F-Coalg** consists of:

- $\mathsf{Ob}_{\textbf{F-Coalg}}$ contains F-coalgebras (C, φ).
- $\mathsf{Arr}_{\textbf{F-Coalg}}$ contains F-coalgebra homomorphism. A homomorphism from F-coalgebra (C, φ) to F-coalgebra (D, ψ) is an arrow $f : C \to D$, such that

$$\mathsf{F}(f) \circ \varphi = \psi \circ f \qquad \text{(FCoalg-Arrow)}$$

In other words, the following diagram should commute

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

The laws for a category hold analogous to those of **F-Alg**.

Finally, we will look at terminal objects in **F-Coalg**. A terminal object in **F-Coalg** should have a unique arrow from every F-coalgebra that ends up in it. Thus, for every F-coalgebra (C, φ) and the terminal F-coalgebra $(\nu\mathsf{F}, \mathsf{out})$, the follow diagram should commute:

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

Just as with the catamorphism, we have a special name for f, an anamorphism $[\!(\varphi)\!]$. The brackets are called lenses, both these names are again devised by Meijer et al. [8]. We can again state the uniqueness of f in the form of an equation:

$$f \circ \varphi = \mathsf{out} \circ \mathsf{F}(f) \quad \equiv \quad f = [\!(\varphi)\!] \tag{ana-Charn}$$

There is also a set of useful lemmas we can now prove about terminal F-coalgebras. These will be very similar to the lemmas of the catamorphism and all proofs will go analogous.

Lemma 3.18 (*Ana-Self*)

For any F*-coalgebra* $(C, \varphi : C \to \mathsf{F}(C))$ *and terminal* F*-coalgebra* $(\nu\mathsf{F}, \mathsf{out} : \nu\mathsf{F} \to \mathsf{F}(\nu\mathsf{F}))$*, we get the anamorphism* $[\!(\varphi)\!]$*, following the equation*

$$[\!(\varphi)\!] \circ \varphi = \mathsf{out} \circ \mathsf{F}[\!(\varphi)\!] \tag{ana-Self}$$

In other words, the following diagram commutes:

$$\begin{tikzcd}
C \arrow[r, dashed, "{[\!(\varphi)\!]}"] \arrow[d, "\varphi"] & \nu\mathsf{F} \arrow[d, "\mathsf{out}"] \\
\mathsf{F}(C) \arrow[r, "{\mathsf{F}[\!(\varphi)\!]}"] & \mathsf{F}(\nu\mathsf{F})
\end{tikzcd}$$

Lemma 3.19 (*Ana-Repl*)

For any terminal F*-coalgebra* $(\nu\mathsf{F}, \mathsf{out} : \nu\mathsf{F} \to \mathsf{F}(\nu\mathsf{F})$

$$\mathsf{id} = [\!(\mathsf{out})\!] \tag{ana-Repl}$$

In other words, the following diagrams commutes:

$$\begin{tikzcd}
\nu\mathsf{F} \arrow[r, dashed, bend left, "\mathsf{id}"] \arrow[r, dashed, bend right, "{[\!(\mathsf{out})\!]}"] \arrow[d, "\mathsf{out}"] & \nu\mathsf{F} \arrow[d, "\mathsf{out}"] \\
\mathsf{F}(\nu\mathsf{F}) \arrow[r, dashed, bend left, "\mathsf{F}(\mathsf{id})"] \arrow[r, dashed, bend right, "{\mathsf{F}[\!(\mathsf{out})\!]}"] & \mathsf{F}(\nu\mathsf{F})
\end{tikzcd}$$

Lemma 3.20 (*Ana-Fusion*)

For any F*-coalgebras* $(C, \varphi : C \to \mathsf{F}(C))$ *and* $(D, \psi : D \to \mathsf{F}(D))$ *with arrow* $f : C \to D$ *and terminal* F*-coalgebra* $(\nu\mathsf{F}, \mathsf{out} : \nu\mathsf{F} \to \mathsf{F}(\nu\mathsf{F}))$

$$\psi \circ f = \mathsf{F}(f) \circ \varphi \quad \Rightarrow \quad \llbracket\psi\rrbracket \circ f = \llbracket\varphi\rrbracket \qquad \text{(ana-Fusion)}$$

In other words, the following diagram commutes:

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "\varphi"] \arrow[rr, bend left, dashed, "\llbracket\varphi\rrbracket"] & D \arrow[r, dashed, "\llbracket\psi\rrbracket"] \arrow[d, "\psi"] & \nu\mathsf{F} \arrow[d, "\mathsf{out}"] \\
\mathsf{F}(C) \arrow[r, "\mathsf{F}(f)"] \arrow[rr, bend right, dashed, "\mathsf{F}\llbracket\varphi\rrbracket"'] & \mathsf{F}(C) \arrow[r, dashed, "\llbracket\psi\rrbracket"] & \mathsf{F}(\nu\mathsf{F})
\end{tikzcd}$$

There also exists a dualisation of Lambeks theorem, the proof is analogous to that of the original.

Theorem 3.21 (*Dual of Lambek [6]*)

The terminal F*-coalgebra* $\mathsf{out} : \nu\mathsf{F} \to \mathsf{F}(\nu\mathsf{F})$ *is an isomorphism with* out^{-1} *defined as:*

$$\mathsf{out}^{-1} = \llbracket\mathsf{F}(\mathsf{out})\rrbracket$$

With all this work done we can define the anamorphism properly:

$$\llbracket\varphi\rrbracket = \mathsf{out}^{-1} \circ \mathsf{F}\llbracket\varphi\rrbracket \circ \varphi$$

3.2.2 Application

An anamorphism creates a possible infinite structure from a seed. The arrow generates from an object C, a new part of the infinite structure and a new seed. We will see how this comes into action by looking at streams.

The Stream object

Definition 3.22 (*Anamorphism on streams*)

An object is a Stream object if it has the following arrows, $\mathsf{cons} : A \times \mathsf{Stream}_A \to \mathsf{Stream}_A$, $\mathsf{head} : \mathsf{Stream}_A \to A$ and $\mathsf{tail} : \mathsf{Stream}_A \to A$.

Where

$$\begin{aligned}\mathsf{head} \circ \mathsf{cons} &= \mathsf{fst} \\ \mathsf{tail} \circ \mathsf{cons} &= \mathsf{snd}\end{aligned}$$

Such that

$$\langle \mathsf{head}, \mathsf{tail} \rangle : A \times \mathsf{Stream}_A \to \mathsf{Stream}_A$$

is an isomorphism with

$$\langle \mathsf{head}, \mathsf{tail} \rangle^{-1} = \mathsf{cons}$$

Or the following diagram commutes:

$$\begin{tikzcd}
 & \mathsf{Stream}_A \arrow[dl, "\mathsf{tail}"'] \arrow[dr, "\mathsf{head}"] \arrow[d, bend left, "\langle \mathsf{head},\mathsf{tail} \rangle"] & \\
A & A \times \mathsf{Stream}_A \arrow[l, "\mathsf{fst}"] \arrow[r, "\mathsf{snd}"'] \arrow[u, bend left, "\mathsf{cons}"] & \mathsf{Stream}_A
\end{tikzcd}$$

Example 3.23 (Stream *object in Set*)

In **Set** the Stream_A object is the set of infinite tuples carrying elements of A. The functions are then trivially defined.

The initial object of the functor $\mathsf{S}_A(X) = A \times X$ happens to have these properties

Lemma 3.24 (*Initial* S_A*-algebra*)

The initial object of a S_A*-algebra is a* Stream *object.*

Proof. This proof is analogous to the proof of the dual of Lemma 3.7. □

Now that we have the terminal S_A-coalgebra $(\mathsf{Stream}_A, \langle \mathsf{head}, \mathsf{tail} \rangle$, we will show how to go from a recursive scheme to an anamorphism:

Lemma 3.25 (*Anamorphism from recursive equations*)

For any $f : A \to \mathsf{Stream}_B$ *with the following recursive equations and* $c : A \to B$, $h : A \to A$

$$\begin{aligned}\mathsf{head} \circ f &= c \\ \mathsf{tail} \circ f &= f \circ h\end{aligned}$$

It holds that with the S_A*-coalgebra* $(A, \langle c, h \rangle)$,

$$f = [\![\langle c, h \rangle]\!]$$

Proof. We assume that we have the arrows $c : A \to B$, $h : A \to A$ and $f : A \to \mathsf{Stream}_B$ with

$$\begin{aligned} \mathsf{head} \circ f &= c \\ \mathsf{tail} \circ f &= f \circ h \end{aligned}$$

We now want to prove that

$$f = [\![\langle c, h \rangle]\!]$$

Following from $\mathsf{head} \circ f = c$, it holds that

$$\mathsf{head} \circ f = \mathsf{id} \circ \mathsf{fst} \circ \langle c, h \rangle$$

And following from $\mathsf{tail} \circ f = f \circ h$, it holds that

$$\mathsf{tail} \circ f = f \circ \mathsf{snd} \circ \langle c, h \rangle$$

We can combine these two equations into

$$\langle \mathsf{head}, \mathsf{tail} \rangle \circ f = (\mathsf{id} \times f) \circ \langle c, h \rangle$$

Then following from ana-Charn we get that

$$f = [\![\langle c, h \rangle]\!]$$

□

Example 3.26 (*Stream of natural numbers*)

We would like to create the infinite stream of natural numbers following the recursive equations with $\mathsf{nats} : \mathsf{Nat} \to \mathsf{Stream}_{\mathsf{Nat}}$:

$$\begin{aligned} \mathsf{head} \circ \mathsf{nats} &= \mathsf{id} \\ \mathsf{tail} \circ \mathsf{nats} &= \mathsf{nats} \circ \mathsf{succ} \end{aligned}$$

To create an anamorphism for nats, we need a $\mathsf{S}_{\mathsf{Nat}}$-coalgebra $(\mathsf{Nat}, \varphi : \mathsf{Nat} \to \mathsf{Nat} \times \mathsf{Nat})$ such that

$$\mathsf{nats} = [\![\varphi]\!]$$

Following from Lemma 3.25, $\varphi = \langle \mathsf{id}, \mathsf{succ} \rangle$. Thus,

$$f = [\![\langle \mathsf{id}, \mathsf{succ} \rangle]\!]$$

$$\begin{tikzcd}
\mathsf{Nat} \arrow[r, dashed, "[\![\langle \mathsf{id},\mathsf{succ}\rangle]\!]"] \arrow[d, "\langle \mathsf{id},\mathsf{succ}\rangle"] & \mathsf{Stream}_{\mathsf{Nat}} \arrow[d, "\langle \mathsf{head},\mathsf{tail}\rangle"] \\
\mathsf{Nat} \times \mathsf{Nat} \arrow[r, "\mathsf{id} \times [\![\langle \mathsf{id},\mathsf{succ}\rangle]\!]"] & \mathsf{Nat} \times \mathsf{Stream}_{\mathsf{Nat}}
\end{tikzcd}$$

Example 3.27 (*Zip two streams*)

We would like to zip two streams such that they alternate. The arrow $\mathsf{zip} : \mathsf{Stream}_A \times \mathsf{Stream}_A \to \mathsf{Stream}_A$ has the following recursive equations:

$$\begin{aligned} \mathsf{head} \circ \mathsf{zip} &= \mathsf{head} \circ \mathsf{fst} \\ \mathsf{tail} \circ \mathsf{zip} &= \langle \mathsf{snd}, \mathsf{tail} \circ \mathsf{fst} \rangle \end{aligned}$$

Thus, we get the following anamorphism

$$\mathsf{zip} = [\![\langle \mathsf{head} \circ \mathsf{fst}, \langle \mathsf{snd}, \mathsf{tail} \circ \mathsf{fst} \rangle \rangle$$

$$\begin{tikzcd}
\mathsf{Stream}_A \times \mathsf{Stream}_A \arrow[r, dashed, "[\![\langle \mathsf{head} \circ \mathsf{fst},\langle \mathsf{snd},\mathsf{tail} \circ \mathsf{fst}\rangle\rangle]\!]"] \arrow[d, "\langle \mathsf{head} \circ \mathsf{fst},\langle \mathsf{snd},\mathsf{tail} \circ \mathsf{fst}\rangle\rangle"] & \mathsf{Stream}_A \arrow[d, "\langle \mathsf{head},\mathsf{tail}\rangle"] \\
A \times \mathsf{Stream}_A \times \mathsf{Stream}_A \arrow[r, "\mathsf{id} \times [\![\langle \mathsf{head} \circ \mathsf{fst},\langle \mathsf{snd},\mathsf{tail} \circ \mathsf{fst}\rangle\rangle]\!]"] & A \times \mathsf{Stream}_A
\end{tikzcd}$$

3.3 Primitive recursion

Now that we have shown how iteration works in category theory, we attempt to do the same for more interesting types of recursion. This section focus on primitive recursion. Using the natural numbers, primitive recursion looks as follows:

$$\begin{aligned} f(0) &= c() \\ f(n+1) &= h(f(n), n) \end{aligned}$$

Thus, not only does a step in our function depend upon the result of the previous step, it also depends on the argument currently given.

Thus, for a step in an arrow $\mu\mathsf{F} \to C$ every step in our recursive function should not just take $\mathsf{F}(C)$ but also $\mathsf{F}(\mu\mathsf{F})$. Our evaluation arrow φ thus becomes an arrow of type $\mathsf{F}(C \times \mu\mathsf{F}) \to C$ and has the following diagram:

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

We can create the factorial function in this recursive scheme by taking $\mathsf{F} = \mathsf{N}$, $\mu\mathsf{F} = \mathsf{Nat}$ and $C = \mathsf{Nat}$ with

$$\varphi = [\mathsf{one}, \mathsf{mul} \circ (\mathsf{id} \times \mathsf{succ})]$$

resulting in the following diagram

$$\begin{tikzcd}
1 + \mathsf{Nat} \arrow[r, "\mathsf{id} + \langle \mathsf{fact},\mathsf{id}\rangle"] \arrow[d, "{[\mathsf{zero},\mathsf{succ}]}"] & 1 + (\mathsf{Nat} \times \mathsf{Nat}) \arrow[d, "{[\mathsf{one},\mathsf{mul} \circ (\mathsf{id} \times \mathsf{succ})]}"] \\
\mathsf{Nat} \arrow[r, dashed, "\mathsf{fact}"] & \mathsf{Nat}
\end{tikzcd}$$

However, as this is not a well-defined catamorphism, we do not yet have a definition for fact. Fortunately if we have a diagram such as the one above the catamorphism exists.

Corollary 3.28 (*Primitive recursion in a catamorphism [7]*)

For any initial F*-algebra* $(\mu\mathsf{F}, \mathsf{in})$ *and two arrows* $f : \mu\mathsf{F} \to C$ *and* $\varphi : \mathsf{F}(C \times \mu\mathsf{F}) \to C$*, the following holds:*

$$f \circ \mathsf{in} = \varphi \circ \mathsf{F}\langle f, \mathsf{id}\rangle \quad \equiv \quad f = \mathsf{fst} \circ (\!|\langle \varphi, \mathsf{in} \circ \mathsf{F}(\mathsf{snd})\rangle|\!) \qquad \text{(cata-PrimRec)}$$

Or the following diagram commutes

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

iff

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, "\mathsf{F}(\!|\langle\varphi, \mathsf{in}\circ\mathsf{F}(\mathsf{snd})\rangle|\!)"] \arrow[d, "\mathsf{in}"] & \mathsf{F}(C \times \mu\mathsf{F}) \arrow[d, "\langle\varphi, \mathsf{in}\circ\mathsf{F}(\mathsf{snd})\rangle"] \\
\mu\mathsf{F} \arrow[r, "(\!|\langle\varphi, \mathsf{in}\circ\mathsf{F}(\mathsf{snd})\rangle|\!)"] \arrow[dr, "f"] & C \times \mu\mathsf{F} \arrow[d, "\mathsf{fst}"] \\
 & C
\end{tikzcd}$$

Thus, we can create a proper catamorphism for our fact arrow:

$$\mathsf{fact} = \mathsf{fst} \circ (\!|\langle\varphi, \mathsf{in} \circ \mathsf{F}(\mathsf{snd})\rangle|\!)$$

To be able to reason about paramorphisms, a bit of notation is introduced to capture this recursion more compactly.

Definition 3.29 (*Paramorphism*)

For a functor F with an initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$ and an F-algebra (C, φ), the paramorphism is defined as

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

Remark. The paramorphism named as such originally appeared in a paper by Meertens called Paramorphisms [7]. Primitive recursion was also analysed in a paper by Geuvers [2]. He analysed the precise differences in a categorical sense between iteration and primitive recursion and dualised them.

Example 3.30 (*Factorial as paramorphism*)

We can now very easily write our factorial function as a paramorphism

$$\mathsf{fact} = \langle\!|[\mathsf{one}, \mathsf{mul} \circ (\mathsf{id} \times \mathsf{succ})]|\!\rangle$$

We can create many types of recursion using catamorphisms and anamorphisms. We have shown how to create iteration and primitive recursion. However, catamorphisms are actually more powerful than just those two options. We will be looking at an expansion on cata- and anamorphisms in the next chapter.

Chapter 4

Hylomorphism

The Hylomorphism was first coined in a paper by Meijer et al. [8]. However, to have the hylomorphism be the unique solution to its recursive equations, the category and funtor had to have coinciding initial F-algebras and terminal F-coalgebras. This method will be discussed first. Recently however, Capretta et al. [1] did work on finding a way around these restrictions by proving that for certain F-coalgebras the recursive equations are unique. We will show these proofs and theorems too.

4.1 Hylomorphism

We have seen how to create arrows from an initial F-algebra to any F-algebra and how to create an arrow from any F-coalgebra to a terminal F-coalgebra. However, sometimes we would like to create a recursive arrow from any F-coalgebra to any F-algebra. One example of when this is useful, is with quick sort.

$\mathsf{qsort} : \mathsf{List}_A \to \mathsf{List}_A$ recursive equations are as follows:

$$\begin{aligned} \mathsf{qsort} \circ \mathsf{nil} &= \mathsf{nil} \\ \mathsf{qsort} \circ \mathsf{cons} &= \mathsf{concat} \circ \langle \mathsf{qsort} \circ \mathsf{filter}^{\leq}, \mathsf{cons} \circ \langle \mathsf{fst}, \mathsf{qsort} \circ \mathsf{filter}^{>} \rangle \rangle \\ &\text{which is also written as} \\ \mathsf{qsort} \circ \mathsf{cons}(a, l) &= \mathsf{concat}(\mathsf{qsort}(l_{\leq a}), \mathsf{cons}(a, \mathsf{qsort}(l_{>a}))) \end{aligned}$$

Where $\mathsf{filter}^{\leq} : (A \times \mathsf{List}_A) \to \mathsf{List}_A$ creates a list of all elements smaller than or equal to a and $\mathsf{filter}^{>} : (A \times \mathsf{List}_A) \to \mathsf{List}_A$ does the same but with elements larger than a.

One serious problem with transforming these equations into a catamorphism, is that the input is transformed before we do our recursive call and after. Thus a simple catamorphism or anamorphism is insufficient. What we can do instead, is combine the cata and anamorphism. We do this by first using an anamorphism to build up a temporary useful structure which we then collapse using a catamorphism.

We find the structure for the catamorphism and anamorphism by rewriting our second recursive equations.

$$\begin{aligned}\mathsf{qsort} \circ \mathsf{cons} = \mathsf{concat} \circ &\langle \mathsf{fst} \circ \mathsf{snd}, \mathsf{cons} \circ (\mathsf{id} \times \mathsf{snd})\rangle \\ &\circ (\mathsf{id} \times \mathsf{qsort} \times \mathsf{qsort}) \\ &\circ \langle \mathsf{fst}, \langle \mathsf{filter}^{\leq}, \mathsf{filter}^{>}\rangle\rangle\end{aligned}$$

The middle line looks very much like the structure of a binary tree. Thus, we may use a recursive function like:

$$\mathsf{qsort} = \mathsf{qconcat} \circ \mathsf{T}_A(\mathsf{qsort}) \circ \mathsf{qpartition}$$

with

$$\begin{aligned}\mathsf{T}_A(X) &= 1 + A \times X \times X \\ \mathsf{T}_A(f) &= [\mathsf{id}, \mathsf{id} \times f \times f]\end{aligned}$$

$$\begin{aligned}\mathsf{qpartition} &: \mathsf{List}_A \to 1 + A \times \mathsf{List}_A \times \mathsf{List}_A \\ \mathsf{qpartition} &= [\mathsf{inl}, \langle \mathsf{fst}, \langle \mathsf{filter}^{\leq}, \mathsf{filter}^{>}\rangle\rangle]\end{aligned}$$

$$\begin{aligned}\mathsf{qconcat} &: 1 + A \times \mathsf{List}_A \times \mathsf{List}_A \to \mathsf{List}_A \\ \mathsf{qconcat} &= [\mathsf{nil}, \mathsf{concat} \circ \langle \mathsf{fst} \circ \mathsf{snd}, \mathsf{cons} \circ (\mathsf{id} \times \mathsf{snd})\rangle]\end{aligned}$$

and a diagram that looks like:

$$\begin{tikzcd}
1 + A \times \mathsf{List}_A \times \mathsf{List}_A \arrow[r, "\mathsf{T}_A(\mathsf{qsort})"] & 1 + A \times \mathsf{List}_A \times \mathsf{List}_A \arrow[d, "\mathsf{qconcat}"] \\
\mathsf{List}_A \arrow[u, "\mathsf{qpartition}"] \arrow[r, "\mathsf{qsort}"] & \mathsf{List}_A
\end{tikzcd}$$

Here, qsort is the unique solution to the equation $f = \mathsf{qconcat} \circ \mathsf{T}_A(f) \circ \mathsf{qpartition}$. Thus, because qsort is unique and only depends on qconcat and qpartition we will define qsort as $[\![\mathsf{qpartition}, \mathsf{qconcat}]\!]$. This is called a hylomorphism and the $[\![]\!]$ are called envelopes as named by Meijer et al. [8]. However, there are some restrictions when talking about hylomorphisms.

That qsort is the unique solution to the equation $f = \mathsf{qconcat} \circ \mathsf{T}_A(f) \circ \mathsf{qpartition}$, is not as trivial as with the cata- and anamorphism. The cata- and anamorphism could use the properties of the initial and terminal F-algebras, however both $(\mathsf{List}_A, \mathsf{qpartition})$ and $(\mathsf{List}_A, \mathsf{qconcat})$ are not terminal or initial. In some categories this is easily solved, when the initial F-algebra and terminal F-coalgebra coincide we can very easily show that the equation has a unique solution.

If we split up qsort in two parts, an anamorphism and a catamorphism, then

$$\mathsf{qsort} = (\!|\mathsf{qconcat}|\!) \circ [\![\mathsf{qpartition}]\!]$$

with the following diagram, where Tree is the object of the initial T_A-algebra and terminal T_A-coalgebra

$$\begin{tikzcd}
1 + A \times \mathsf{List}_A \times \mathsf{List}_A \arrow[r, "\mathsf{T}_A[\![\mathsf{qpartition}]\!]"] & 1 + A \times \mathsf{Tree} \times \mathsf{Tree} \arrow[r, "\mathsf{T}_A(\!|\mathsf{qconcat}|\!)"] \arrow[d, bend left, "\mathsf{in}"] & 1 + A \times \mathsf{List}_A \times \mathsf{List}_A \arrow[d, "\mathsf{concat}"] \\
\mathsf{List}_A \arrow[u, "\mathsf{qpartition}"] \arrow[r, "{[\![\mathsf{qpartition}]\!]}"] & \mathsf{Tree}_A \arrow[u, bend left, "\mathsf{out}"] \arrow[r, "(\!|\mathsf{qconcat}|\!)"] & \mathsf{List}_A
\end{tikzcd}$$

This is, as mentioned above, the technique commonly used when describing hylomorphisms and this the way that Meijer et al. originally coined them [8]. However, having coinciding initial F-algebras and terminal F-coalgebras is a very strong restriction, our main category of interrest **FPL** does not have this property for example.

For this reason Capretta et al. came up with the idea of using so-called recursive F-coalgebras to be able to prove uniqueness of a hylomorphism. A recursive F-coalgebra intuitively breaks up its object into smaller parts, thus always resulting in a well defined function.

Definition 4.1 (*Recursive* F*-coalgebra*)

An F-coalgebra (C, φ) is recursive iff for every F-algebra (D, ψ) there exists a hylomorphism. Thus, the following equation holds:

$$f = \psi \circ \mathsf{F}(f) \circ \varphi \quad \equiv \quad f = [\![\psi, \varphi]\!] \qquad \text{(hylo-Charn)}$$

Or in other words

$$\begin{tikzcd}
\mathsf{F}(C) \arrow[r, "\mathsf{F}[\![\psi{,}\varphi]\!]"] & \mathsf{F}(D) \arrow[d, "\psi"] \\
C \arrow[u, "\varphi"] \arrow[r, dashed, "{[\![\psi{,}\varphi]\!]}"] & D
\end{tikzcd}$$

Remark. Capretta et al. used a different notation in their paper [1]. Hylomorphism are instead called coalgebra-to-algebra morphisms and are denoted by $\mathsf{fix}_{\mathsf{F},\varphi}(\psi)$. However, we will be using the original notation from Meijer et al.

Thus, we can actually use any T_A-algebra as the second part of a hylomorphism if we have a recursive F-coalgebra. If we for example take the T_A-algebra $(\mathsf{Bool}, \mathsf{qsearch}_p : 1 + A \times \mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool})$ with $p : A \to \mathsf{Bool}$ and

$$\mathsf{qsearch}_p = [\mathsf{false}, \mathsf{or} \circ (p \times \mathsf{id} \times \mathsf{id})]$$

Then $[\![\mathsf{qpartition}, \mathsf{qsearch}_p]\!]$ will check if there is an element in a list that satisfies p. Thus, the trick now becomes finding recursive F-coalgebras.

4.2 Recursive F-coalgebras

The first and very obvious way to find a recursive F-coalgebra is by using the technique used by Meijer et al.

Lemma 4.2 (*Recursiveness in special cases*)

If for a functor F it holds that the initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$ and terminal F-coalgebra $(\nu\mathsf{F}, \mathsf{out})$ coincide, they have the following properties

$$\nu\mathsf{F} = \mu\mathsf{F}$$
$$\mathsf{in}^{-1} = \mathsf{out}$$

Any F-coalgebra is recursive.

Proof. We take an arbitrary functor F where the initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$ and terminal F-coalgebra $(\nu\mathsf{F}, \mathsf{out})$ coincide. We then take an arbitrary F-coalgebra (C, φ), we have to prove that for any F-algebra (D, ψ), the hylomorphism $[\![\varphi, \psi]\!]$ exists.

We can construct the anamorphism from (C, φ) to our terminal F-coalgebra: $[\!(\varphi)\!] : C \to \nu\mathsf{F}$

$$\begin{tikzcd}
\mathsf{F}(C) \arrow[r, "\mathsf{F}[\!(\varphi)\!]"] & \mathsf{F}(\nu\mathsf{F}) \\
C \arrow[u, "\varphi"] \arrow[r, dashed, "[\!(\varphi)\!]"] & \nu\mathsf{F} \arrow[u, "\mathsf{out}"']
\end{tikzcd}$$

We can also construct the catamorphism from our initial F-algebra to (D, ψ): $(\!|\psi|\!) : \mu\mathsf{F} \to D$

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, "\mathsf{F}(\!|\psi|\!)"] \arrow[d, "\mathsf{in}"] & \mathsf{F}(D) \arrow[d, "\psi"] \\
\mu\mathsf{F} \arrow[r, dashed, "(\!|\psi|\!)"] & D
\end{tikzcd}$$

We can then combine our anamorphism and catamorphism to create an arrow from C to D: $[\![(\varphi]\!] \circ (\!|\psi]\!]\!)$, we will denote $\mu\mathsf{F}$ and $\nu\mathsf{F}$ as $\mu\nu\mathsf{F}$

$$\begin{tikzcd}
\mathsf{F}(C) \arrow[r, "\mathsf{F}[\![(\varphi)]\!]"] & \mathsf{F}(\mu\nu\mathsf{F}) \arrow[r, "\mathsf{F}(\!|\psi|\!)"] \arrow[d, bend left, "\mathsf{in}"] & \mathsf{F}(D) \arrow[d, "\psi"] \\
C \arrow[u, "\varphi"] \arrow[r, dashed, "[\![(\varphi)]\!]"] \arrow[rr, dashed, bend right, "[\![(\varphi)]\!] \circ (\!|\psi|\!)"] & \mu\nu\mathsf{F} \arrow[u, bend left, "\mathsf{out}"] \arrow[r, dashed, "(\!|\psi|\!)"] & D
\end{tikzcd}$$

Thus, following from the uniqueness of the anamorphism and catamorphism, it follows that $[\![(\varphi]\!] \circ (\!|\psi]\!]\!)$ is the unique arrow from C to D and thus following from hylo-CHARNthe hylomorphism from an arbitrary C to an arbitrary D exists. $\square$

However, we would also like to say something about the case that an F-coalgebra is recursive and the initial and terminal F-algebra and F-coalgebra don't coincide. We provide a few lemmas with which to show that an F-coalgebra is recursive without that restriction.

We first need a starting point from which we can prove that an F-coalgebra is recursive. This starting point is the initial F-algebra.

Lemma 4.3 (*Recursive inital* F*-algebra*)

If a functor F *has an initial* F*-algebra* $(\mu\mathsf{F}, \mathsf{in})$*, then* $(\mu\mathsf{F}, \mathsf{in}^{-1})$ *is a recursive* F*-coalgebra.*

Proof. For a functor F with an initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$ and F-algebra (C, φ), there exists a unique arrow from $\mu\mathsf{F}$ to C. This arrow is the catamorphism $(\!|\varphi|\!)$. This catamorphism is the unique arrow, thus it is also also the hylomorphism from $(\mu\mathsf{F}, \mathsf{in}^{-1})$ to (C, φ). $\square$

We now want to have some way to prove that another F-coalgebra is recursive from an existing recursive F-coalgebra.

Lemma 4.4 (*Reduction of recursiveness*)

Let $\mathsf{F} : \boldsymbol{C} \to \boldsymbol{C}$ *be a functor,* (A, α) *a recursive* F*-coalgebra and* (B, β) *a* F*-coalgebra.*

If there are F*-coalgebra morphisms* $h : A \to B$ *and* $k : B \to \mathsf{F}(A)$ *such that* $\beta = \mathsf{F}(h) \circ k$*, then* (B, β) *is recursive.*

In other words, if the following diagram commutes for any F*-algebra* (C, φ)

$$\begin{tikzcd}
C & A \arrow[l, "{[\![\alpha,\varphi]\!]}"'] \arrow[r, "h"] \arrow[d, "\alpha"] & B \arrow[dl, "k"] \arrow[d, "\beta"] \\
\mathsf{F}(C) \arrow[u, "\varphi"] & \mathsf{F}(A) \arrow[l, "{\mathsf{F}[\![\alpha,\varphi]\!]}"'] \arrow[r, "\mathsf{F}(h)"] \arrow[d, "\mathsf{F}(\alpha)"] & \mathsf{F}(B) \arrow[dl, "\mathsf{F}(K)"] \\
 & \mathsf{F}(\mathsf{F}(A)) &
\end{tikzcd}$$

Then so does this diagram

$$\begin{tikzcd}
C \arrow[d, "\varphi"] & B \arrow[l, "{[\![\beta,\varphi]\!]}"'] \arrow[d, "\beta"] \\
\mathsf{F}(C) & \mathsf{F}(B) \arrow[l, "{\mathsf{F}[\![\beta,\varphi]\!]}"']
\end{tikzcd}$$

Proof. For a functor F with a recursive F-coalgebra (A, α) and a F-coalgebra (B, β) with F-coalgebra morphisms $h : A \to B$ and $k : B \to \mathsf{F}(A)$ such that $\beta = \mathsf{F}(h) \circ k$. We take an arbitrary F-algebra (C, φ). We then want to prove that the hylomorphism $[\![\beta, \varphi]\!]$ exists.

Thus, we want to prove that there exists a unique f equal to $[\![\beta, \varphi]\!]$.

▷ $\mathsf{F}(\alpha) \circ k = \mathsf{F}(k) \circ \beta$

f

$=$ – ▷, hylo-Charn –

$\varphi \circ \mathsf{F}[\![\alpha, \varphi]\!] \circ k$

$=$ – hylo-Charn –

$\varphi \circ \mathsf{F}(\varphi \circ \mathsf{F}[\![\alpha, \varphi]\!] \circ \alpha) \circ k$

$=$ – F-functor –

$\varphi \circ \mathsf{F}(\varphi \circ \mathsf{F}[\![\alpha, \varphi]\!]) \circ \mathsf{F}(\alpha) \circ k$

$=$ – ▷ –

$\varphi \circ \mathsf{F}(\varphi \circ \mathsf{F}[\![\alpha, \varphi]\!]) \circ \mathsf{F}(k) \circ \beta$

$=$ – F-functor –

$\varphi \circ \mathsf{F}(\varphi \circ \mathsf{F}[\![\alpha, \varphi]\!] \circ k) \circ \beta$

$=$ – ▷, hylo-Charn –

$\varphi \circ \mathsf{F}(f) \circ \beta$

$=$ – hylo-Charn –

$[\![\beta, \varphi]\!]$

$$\begin{tikzcd}
C & A \arrow[l, "{[\![\alpha,\varphi]\!]}"'] \arrow[d, "\alpha"] & & B \arrow[lll, bend right=40, "{[\![\beta,\varphi]\!]}"'] \arrow[lll, bend right=25, "f"'] \arrow[dll, "k"] \arrow[d, "\beta"] \\
\mathsf{F}(C) \arrow[u, "\varphi"] & \mathsf{F}(A) \arrow[l, "{\mathsf{F}[\![\alpha,\varphi]\!]}"'] \arrow[d, "\mathsf{F}(\alpha)"'] & \arrow[r, phantom, "\triangleright"] & \mathsf{F}(B) \arrow[lll, bend left=20, "\mathsf{F}(f)"] \arrow[dll, "\mathsf{F}(K)"] \\
\mathsf{F}(\mathsf{F}(C)) \arrow[u, "\mathsf{F}(\varphi)"] & \mathsf{F}(\mathsf{F}(A)) \arrow[l, "{\mathsf{F}(\mathsf{F}[\![\alpha,\varphi]\!])}"'] & &
\end{tikzcd}$$

To show that f is unique, suppose that there exists another hylomorphism $f' : B \to C$. Then if we take $f' \circ h$, we can prove this is equal to $\varphi \circ \mathsf{F}(f' \circ h) \circ \alpha$. But from the unicity of $[\![\alpha, \varphi]\!]$, it follows that $f' = f$.

▷ $\beta \circ h = \mathsf{F}(h) \circ \alpha$

$f' \circ h$

$=$ – hylo-Charn –

$\varphi \circ \mathsf{F}(f') \circ \beta \circ h$

$=$ – ▷ –

$\varphi \circ \mathsf{F}(f') \circ \mathsf{F}(h) \circ \alpha$

$=$ – F-functor –

$\varphi \circ \mathsf{F}(f' \circ h) \circ \alpha$

$$\begin{tikzcd}
C & A \arrow[l, "{[\![\alpha,\varphi]\!]}"'] \arrow[r, "h"] \arrow[d, "\alpha"] & B \arrow[ll, bend right=30, "f'"'] \arrow[d, "\beta"] \\
\mathsf{F}(C) \arrow[u, "\varphi"] & \mathsf{F}(A) \arrow[l, "{\mathsf{F}[\![\alpha,\varphi]\!]}"'] \arrow[r, "\mathsf{F}(h)"] & \mathsf{F}(B) \arrow[ll, bend left=30, "\mathsf{F}(f')"]
\end{tikzcd}$$

□

The previous is one of the most import ones, we will use it prove some small lemmas to allow us to prove recursiveness of F-coalgebras more easily.

Lemma 4.5 (*Recursiveness over functors*)

If a functor F *has a recursive* F*-coalgebra* (C, φ) *then* $(\mathsf{F}(C), \mathsf{F}(\varphi))$ *is also* *recursive.*

Proof. We apply Lemma 4.4 with $h = \varphi$ and $k = \mathsf{id}_{\mathsf{F}(C)}$. □

To demonstrate how this recursiveness property of F-coalgebras is used, we will show how to create a hylomorphism that goes back one or two steps, instead of just one. We will begin by proving a lemma and then show how we can use this lemma by creating the Fibonacci function.

Lemma 4.6 (*namePararecursive reduction*)

If a functor F *has a recursive* F*-coalgebra* (A, α)*, then* $(A, \mathsf{F}\langle \mathsf{id}_A, \alpha\rangle \circ \alpha)$ *is a recursive* $\mathsf{F}(\mathsf{ID} \times \mathsf{F})$*-algebra.*

Proof. We take a functor F with a recursive F-coalgebra (A, α), and an arbitrary $\mathsf{F}(\mathsf{ID} \times \mathsf{F})$-algebra (C, φ). For clarity we let $\varphi^\times = \langle \varphi, \mathsf{F}(\mathsf{fst})\rangle : \mathsf{F}(C \times \mathsf{F}(C)) \to C \times \mathsf{F}(C)$ We also let $f = \mathsf{fst} \circ [\![\alpha, \varphi^\times]\!]$, we then want to prove that

$$f = [\![\mathsf{F}\langle \mathsf{id}, \alpha\rangle \circ \alpha, \varphi]\!]$$

.

$$\begin{tikzcd}
\mathsf{F}(C \times \mathsf{F}(C)) \arrow[d, equal] & \mathsf{F}((C \times \mathsf{F}(C)) \times \mathsf{F}(C \times \mathsf{F}(C))) \arrow[l, "\mathsf{F}(\mathsf{fst} \times \mathsf{F}(\mathsf{fst}))"'] \arrow[d, "\mathsf{F}(\mathsf{id} \times \langle \varphi, \mathsf{F}(\mathsf{fst})\rangle)"] & & \mathsf{F}(A \times \mathsf{F}(A)) \arrow[lll, bend right, "\mathsf{F}(f \times \mathsf{F}(f))"'] \arrow[ll, "\mathsf{F}([\![\alpha,\varphi^\times]\!] \times \mathsf{F}([\![\alpha,\varphi^\times]\!]))"'] \\
\mathsf{F}(C \times \mathsf{F}(C)) \arrow[d, equal] & \mathsf{F}((C \times \mathsf{F}(C)) \times (C \times \mathsf{F}(C))) \arrow[l, "\mathsf{F}(\mathsf{fst} \times \mathsf{snd})"'] & \mathsf{F}(A \times A) \arrow[l, "\mathsf{F}([\![\alpha,\varphi^\times]\!] \times [\![\alpha,\varphi^\times]\!])"'] \arrow[ur, "\mathsf{F}(\mathsf{id} \times \alpha"] & \\
\mathsf{F}(C \times \mathsf{F}(C)) \arrow[r, equal] \arrow[d, "\varphi"] & \mathsf{F}(C \times \mathsf{F}(C)) \arrow[u, "\mathsf{F}\langle \mathsf{id}, \mathsf{id}\rangle"] \arrow[d, "\varphi^\times"] & & \mathsf{F}(A) \arrow[ll, "\mathsf{F}[\![\alpha,\varphi^\times]\!]"'] \arrow[ul, "\mathsf{F}\langle \mathsf{id} \times \mathsf{id}\rangle"'] \arrow[uu, "\mathsf{F}\langle \mathsf{id}, \alpha\rangle"] \\
C & C \times \mathsf{F}(C) \arrow[l, "\mathsf{fst}"'] & & A \arrow[ll, "[\![\alpha,\varphi^\times"'] \arrow[u, "\alpha"] \arrow[lll, bend left, "f"]
\end{tikzcd}$$

$\triangleright \quad \varphi^{\times} = \langle \varphi, \mathsf{F}(\mathsf{fst}) \rangle$

$\blacktriangleright \quad f = \mathsf{fst} \circ [\![\alpha, \varphi^{\times}]\!]$

$$
\begin{aligned}
& f \\
= & \quad - \blacktriangleright - \\
& \mathsf{fst} \circ [\![\alpha, \varphi^{\times}]\!] \\
= & \quad - \text{hylo-}\textsc{Charn} - \\
& \mathsf{fst} \circ \varphi^{\times} \circ \mathsf{F}[\![\alpha, \varphi^{\times}]\!] \circ \alpha \\
= & \quad - \triangleright - \\
& \mathsf{fst} \circ \langle \varphi, \mathsf{F}(\mathsf{fst}) \rangle \circ \mathsf{F}[\![\alpha, \varphi^{\times}]\!] \circ \alpha \\
= & \quad - \text{Product-}\textsc{Left} - \\
& \varphi \circ \mathsf{F}[\![\alpha, \varphi^{\times}]\!] \circ \alpha \\
= & \quad - \text{paring} - \\
& \varphi \circ \mathsf{F}(\mathsf{fst} \times \mathsf{snd}) \circ \mathsf{F}\langle \mathsf{id}, \mathsf{id} \rangle \circ \mathsf{F}[\![\alpha, \varphi^{\times}]\!] \circ \alpha \\
= & \quad - \text{pairing} - \\
& \varphi \circ \mathsf{F}(\mathsf{fst} \times \mathsf{snd}) \circ \mathsf{F}([\![\alpha, \varphi^{\times}]\!] \times [\![\alpha, \varphi^{\times}]\!]) \circ \mathsf{F}\langle \mathsf{id}, \mathsf{id} \rangle \circ \alpha \\
= & \quad - \text{hylo-}\textsc{Charn} - \\
& \varphi \circ \mathsf{F}(\mathsf{fst} \times \mathsf{snd}) \circ \mathsf{F}([\![\alpha, \varphi^{\times}]\!] \times (\varphi^{\times} \circ \mathsf{F}[\![\alpha, \varphi^{\times}]\!] \circ \alpha)) \circ \mathsf{F}\langle \mathsf{id}, \mathsf{id} \rangle \circ \alpha \\
= & \quad - \text{F-functor} - \\
& \varphi \circ \mathsf{F}(\mathsf{fst} \times \mathsf{snd}) \circ \mathsf{F}(\mathsf{id} \times \varphi^{\times}) \circ \mathsf{F}([\![\alpha, \varphi^{\times}]\!] \times \mathsf{F}[\![\alpha, \varphi^{\times}]\!]) \circ \mathsf{F}(\mathsf{id} \times \alpha) \circ \mathsf{F}\langle \mathsf{id}, \mathsf{id} \rangle \circ \alpha \\
= & \quad - \triangleright - \\
& \varphi \circ \mathsf{F}(\mathsf{fst} \times \mathsf{snd}) \circ \mathsf{F}(\mathsf{id} \times \langle \varphi, \mathsf{F}(\mathsf{fst}) \rangle) \circ \mathsf{F}([\![\alpha, \varphi^{\times}]\!] \times \mathsf{F}[\![\alpha, \varphi^{\times}]\!]) \circ \mathsf{F}(\mathsf{id} \times \alpha) \circ \mathsf{F}\langle \mathsf{id}, \mathsf{id} \rangle \circ \alpha \\
= & \quad - \text{Product-}\textsc{Left} - \\
& \varphi \circ \mathsf{F}(\mathsf{fst} \times \mathsf{F}(\mathsf{fst})) \circ \mathsf{F}([\![\alpha, \varphi^{\times}]\!] \times \mathsf{F}[\![\alpha, \varphi^{\times}]\!]) \circ \mathsf{F}(\mathsf{id} \times \alpha) \circ \mathsf{F}\langle \mathsf{id}, \mathsf{id} \rangle \circ \alpha \\
= & \quad - \blacktriangleright, \text{Product-}\textsc{Left} - \\
& \varphi \circ \mathsf{F}(f \times \mathsf{F}(f)) \circ \mathsf{F}\langle \mathsf{id}, \alpha \rangle \circ \alpha \\
= & \quad - \text{hylo-}\textsc{Charn} - \\
& [\![\mathsf{F}\langle \mathsf{id}, \alpha \rangle \circ \alpha, \varphi]\!]
\end{aligned}
$$

f is unique as a consequence of $[\![\alpha, \varphi^{\times}]\!]$ being unique. □

Using these lemmas, we will define an example function to show how these lemmas can be applied.

Example 4.7 (*Fibonacci using a hylomorphism*)

We will define the Fibonacci function $\mathsf{fib} : \mathsf{Nat} \to \mathsf{Nat}$. fib has the following well known recursion scheme:

$$\begin{aligned}\mathsf{fib} \circ \mathsf{zero} &= \mathsf{one} \\ \mathsf{fib} \circ \mathsf{succ} \circ \mathsf{zero} &= \mathsf{one} \\ \mathsf{fib} \circ \mathsf{succ} \circ \mathsf{succ} &= \mathsf{add} \circ \langle \mathsf{fib} \circ \mathsf{succ}, \mathsf{fib} \rangle\end{aligned}$$

We start by applying Lemma 4.3 to the functor N with initial N-algebra $(\mathsf{Nat}, [\mathsf{zero}, \mathsf{succ}])$ as per Lemma 3.7. This results in the recursive F-coalgebra $(\mathsf{Nat}, \mathsf{prev})$. By Lemma 4.6 we get the recursive N-coalgebra $(\mathsf{Nat}, (\mathsf{id} + \langle \mathsf{id}, \mathsf{prev} \rangle) \circ \mathsf{prev})$, we will call the arrow of the N-coalgebra, $\mathsf{fibpre} : \mathsf{Nat} \to 1 + \mathsf{Nat} \times (1 + \mathsf{Nat})$. Concretely fibpre applies prev either one or two times depending on its input and thus gives back a sort of tuple of the previous two numbers.

Now that we have our recursive N-coalgebra we can find an N-algebra such that we follow our recursion scheme.

$$\mathsf{fibpost} = [\mathsf{zero}, [\mathsf{succ} \circ \mathsf{zero} \circ \mathsf{snd}, \mathsf{add}] \circ \mathsf{distr}]$$

Thus, our fibonacci function is

$$\mathsf{fib} = [\![\mathsf{fibpre}, \mathsf{fibpost}]\!]$$

$$\begin{tikzcd}
\mathsf{F}(\mathsf{Nat}) \arrow[r, "\mathsf{F}[\![\mathsf{fibpre},\mathsf{fibpost}]\!]"] & \mathsf{F}(\mathsf{Nat}) \arrow[d, "\mathsf{fibpost}"] \\
\mathsf{Nat} \arrow[u, "\mathsf{fibpre}"] \arrow[r, dashed, "{[\![\mathsf{fibpre},\mathsf{fibpost}]\!]}"] & \mathsf{Nat}
\end{tikzcd}$$

Chapter 5

Histomorphism

In this chapter we will be discussing course-of-value recursion as described by Uustalu and Vene in [11] and later expanded upon by Vene in his PhD thesis [13].

Course-of-value recursion is recursion where every previous results can be used. Using the natural numbers course-of-value recursion has the following recursion scheme:

$$\begin{aligned} f(0) &= c() \\ f(n+1) &= h(f(0), f(1), \cdots, f(n)) \end{aligned}$$

This recursion scheme allows us to define some more new functions like the Fibonacci function.

However, to define a structure that captures this new information, like we did before, is a bit more difficult in this case.

5.1 The codata structure

If we have an arrow $f : \mathsf{Nat} \to C$ that uses course-of-value recursion, we would have a diagram

$$\begin{tikzcd} \mathsf{N}(\mathsf{Nat}) \arrow[r, "\mathsf{N}(g)"] \arrow[d, "\mathsf{in}"] & \mathsf{N}(X) \arrow[d, "\varphi"] \\ \mathsf{Nat} \arrow[r, "f"] & C \end{tikzcd}$$

We would like X to capture the result of f for every previous value and g to create that structure. We will first focus on X.

Thus, we would like to create an object that captures the structure of Nat but annotated with every previous value of f. This would result in a tree like structure, where every node has a value and zero or one branch. A layer of this tree could thus be represented by an object X with the arrow

$$\langle \mathsf{value}, \mathsf{branch} \rangle : X \to C \times (1 + X)$$

Where value goes to the value of the function for the number represented by the current layer of X. branch goes to either nothing if we represent 0 or another layer for any other number. We can write this more generally in terms of N as follows:

$$\langle \mathsf{value}, \mathsf{branch} \rangle : X \to C \times \mathsf{N}(X)$$

We can now abstract away from our original Nat object and look at any F with initial F-algebra $(\mu\mathsf{F}, \mathsf{in})$ and arrow $f : \mu\mathsf{F} \to A$ with the diagram

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, "\mathsf{N}(g)"] \arrow[d, "\mathsf{in}"] & \mathsf{F}(X) \arrow[d, "\varphi"] \\
\mu\mathsf{F} \arrow[r, "f"] & A
\end{tikzcd}$$

We would thus get an object X with an arrow

$$\langle \mathsf{value}, \mathsf{branch} \rangle : X \to A \times \mathsf{F}(X)$$

Now every layer of our tree branches with the amount of branches the functor creates. For N this is 1, however e.g. for Tree this would be 2.

Now that we have some specifications for our object X, we can start looking at our arrow g. g should, for a given $\mu\mathsf{F}$, create our object X. Building up a tree from some value happens to be something an anamorphism is very good at. However, for an anamorphism to work, we first need a functor where our X is the terminal object.

Definition 5.1 ($\mathsf{F}_\mathsf{A}^\times$ *functor*)

Given a functor F and an object A, $\mathsf{F}_\mathsf{A}^\times$ is defined as

$$\mathsf{F}_\mathsf{A}^\times(X) = A \times \mathsf{F}(X)$$
$$\mathsf{F}_\mathsf{A}^\times(f) = \mathsf{id} \times \mathsf{F}(f)$$

With the terminal $\mathsf{F}_\mathsf{A}^\times$-algebra

$$(\nu\mathsf{F}_\mathsf{A}^\times, \mathsf{out} : \nu\mathsf{F}_\mathsf{A}^\times \to \mathsf{F}_\mathsf{A}^\times(\nu\mathsf{F}_\mathsf{A}^\times))$$

From this definition it follows that $\mathsf{out} = \langle \mathsf{value}, \mathsf{branch} \rangle$ and out^{-1} is an arrow that constructs a tree from an A and a $\mathsf{F}(\nu\mathsf{F}^{\times}_{\mathsf{A}})$.

Now that we have all parts, we can finally construct our g as an anamorphism on the functor $\mathsf{F}^{\times}_{\mathsf{A}}$:

$$g = [\![\langle f, \mathsf{in}_{\mu}^{-1}\, \mathsf{F} \rangle]\!]$$

This anamorphism is explained by the following diagram

$$\begin{tikzcd}
\nu\mathsf{F}^{\times}_{\mathsf{A}} \arrow[d, "\mathsf{out}"] & & \mu\mathsf{F} \arrow[ll, "{[\![\langle f,\mathsf{in}^{-1}\rangle]\!]}"'] \arrow[r, "f"] \arrow[d, "\mathsf{in}^{-1}"] \arrow[dr, dashed, "{\langle f,\mathsf{in}^{-1}\rangle}"] & A \\
\mathsf{F}^{\times}_{\mathsf{A}}(\nu\mathsf{F}^{\times}_{\mathsf{A}}) & & \mathsf{F}(\mu\mathsf{F}) & \mathsf{F}^{\times}_{\mathsf{A}}(\mu\mathsf{F}) \arrow[l, "\mathsf{snd}"'] \arrow[u, "\mathsf{fst}"] \arrow[lll, bend left, "{\mathsf{F}^{\times}_{\mathsf{A}}[\![\langle f,\mathsf{in}^{-1}\rangle]\!]}"]
\end{tikzcd}$$

Thus, we have now arrived at our finished diagram for course-of-value recursion:

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, "{\mathsf{F}[\![\langle f,\mathsf{in}^{-1}\rangle]\!]}"] \arrow[d, "\mathsf{in}"] & \mathsf{F}(\nu\mathsf{F}^{\times}_{\mathsf{A}}) \arrow[d, "\varphi"] \\
\mu\mathsf{F} \arrow[r, "f"] & A
\end{tikzcd}$$

This diagram consists of a morphism between the initial F-algebra and what is called a F-cv$_A$-algebra.

Definition 5.2 (F-*cv$_A$-algebra*)

Given a functor F, an object A and the companying $\mathsf{F}^{\times}_{\mathsf{A}}$ functor, an F-cv$_A$-algebra is a pair

$$(A, \varphi : \mathsf{F}(\nu\mathsf{F}^{\times}_{\mathsf{A}}) \to A)$$

With this final definition, we can define our histomorphism as the unique solution of f for a course-of-value recursion diagram.

Definition 5.3 (*Histomorphism*)

Given a functor F and a F-cv$_A$-algebra (A, φ), $\{\!|\varphi|\!\}$, called a histomorphism, is the unique arrow making the following diagram commute

$$\begin{tikzcd}
\mathsf{F}(\mu\mathsf{F}) \arrow[r, "\mathsf{F}\llbracket \langle \{\!|\varphi|\!\}, \mathsf{in}^{-1} \rangle \rrbracket"] \arrow[d, "\mathsf{in}"] & \mathsf{F}(\nu\mathsf{F}^{\times}_{\mathsf{A}}) \arrow[d, "\varphi"] \\
\mu\mathsf{F} \arrow[r, dashed, "\{\!|\varphi|\!\}"] & A
\end{tikzcd}$$

Or the following equation holds

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

Thus, the histomorphism computes the histomorphism for all smaller pieces of the the argument in an anamorphism and then computes the final step using φ.

5.2 Application

We will show how a histomorphism is applied to a few problems in the functor N. With the functor N and an object C, we get the following definition for our $\mathsf{N}^{\times}_{\mathsf{C}}$ functor

$$\mathsf{N}^{\times}_{\mathsf{C}}(X) = C \times (1 + X)$$

And we get the terminal object $\nu\mathsf{N}^{\times}_{\mathsf{C}}$, with the arrow as seen in the previous section

$$\langle \mathsf{value}, \mathsf{branch} \rangle : X \to C \times (1 + X)$$

Our anamorphism diagram will look like

$$\begin{tikzcd}
\mathsf{Nat} \arrow[r, "\llbracket \langle \{\!|\varphi|\!\}, \mathsf{in}^{-1} \rangle \rrbracket"] \arrow[d, "\langle \{\!|\varphi|\!\}, \mathsf{in}^{-1} \rangle"] & \nu\mathsf{N}^{\times}_{C} \arrow[d, "\mathsf{out}"] \\
C \times (1 + \mathsf{Nat}) \arrow[r, "\mathsf{id} \times (\mathsf{id} + \llbracket \langle \{\!|\varphi|\!\}, \mathsf{in}^{-1} \rangle \rrbracket)"] & C \times (1 + \nu\mathsf{N}^{\times}_{C})
\end{tikzcd}$$

And the histomorphism diagram will look like

$$\begin{tikzcd}
1 + \mathsf{Nat} \arrow[r, "\mathsf{N}[\![\langle\{\!|\varphi|\!\},\mathsf{in}^{-1}\rangle]\!]"] \arrow[d, "\mathsf{in}"] & 1 + \nu\mathsf{N}^{\times}_{\mathsf{C}} \arrow[d, "\varphi"] \\
\mathsf{Nat} \arrow[r, "\{\!|\varphi|\!\}"] & C
\end{tikzcd}$$

Example 5.4 (*Fibonacci using a Histomorphism*)

We will use the same definition of the Fibonacci function as in Example 4.7:

$$\begin{aligned}
\mathsf{fib} &: \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{fib} \circ \mathsf{zero} &= \mathsf{one} \\
\mathsf{fib} \circ \mathsf{succ} \circ \mathsf{zero} &= \mathsf{one} \\
\mathsf{fib} \circ \mathsf{succ} \circ \mathsf{succ} &= \mathsf{add} \circ \langle \mathsf{fib} \circ \mathsf{succ}, \mathsf{fib} \rangle
\end{aligned}$$

We will use a histomorphism from Nat to Nat, thus we get the following diagram, with $\varphi : \mathsf{N}\ \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \to \mathsf{Nat}$:

$$\begin{tikzcd}
1 + \mathsf{Nat} \arrow[r, "\mathsf{N}\ [\![\langle\{\!|\varphi|\!\},\mathsf{in}^{-1}\rangle]\!]"] \arrow[d, "\mathsf{in}"] & 1 + \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \arrow[d, "\varphi"] \\
\mathsf{Nat} \arrow[r, "\{\!|\varphi|\!\}"] & \mathsf{Nat}
\end{tikzcd}$$

Thus, we now need to define φ. It has to start with a join between the 1 and the $\nu\mathsf{N}^{\times}_{\mathsf{Nat}}$. In the case of 1 the result is one as per the first line of the recursion scheme. In the case of $\nu\mathsf{N}^{\times}_{\mathsf{Nat}}$ we have to do additional work.

$$\varphi = [\mathsf{one}, s_0 : \mathsf{N}^{\times}_{\mathsf{Nat}} \to \mathsf{Nat}]$$

We can first unpack the colist with out and then distribute the output from the previous application over $1 + \mathsf{N}^{\times}_{\mathsf{Nat}}$ to obtain the type $\mathsf{Nat} \times 1 + \mathsf{Nat} \times \mathsf{N}^{\times}_{\mathsf{Nat}}$. We can again join this type. In the case of $\mathsf{Nat} \times 1$, the result is again one as per the second line of the recursion scheme. In the other case we again need to do some additional work.

$$s_0 = [\mathsf{one} \circ \mathsf{snd}, s_1 : \mathsf{Nat} \times \mathsf{N}^{\times}_{\mathsf{Nat}} \to \mathsf{Nat}] \circ \mathsf{distr} \circ \mathsf{out}$$

s_1 is the recursive case from the recursion scheme. Thus, we need to add the result of the previous application and the result before that. The first element of the product we get as input is the result of the previous application of the function. The value of the colist is the result of the application that precedes it. Then we add those two values together

$$s_1 = \mathsf{add} \circ (\mathsf{id} \times \mathsf{value})$$

Thus,

$$\varphi = [\mathsf{one}, [\mathsf{one} \circ \mathsf{snd}, \mathsf{add} \circ (\mathsf{id} \times \mathsf{value})] \circ \mathsf{distr} \circ \mathsf{out}]$$

Which we can show diagrammatically

$$\begin{tikzcd}
1 \arrow[r, "\mathsf{inl}"] \arrow[ddddr, "\mathsf{one}"'] & 1 + \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \arrow[dddd, dashed, "\varphi"] & & \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \arrow[ll, "\mathsf{inr}"'] \arrow[d, "\mathsf{out}"] & \\
& & & \mathsf{Nat} \times (1 + \nu\mathsf{N}^{\times}_{\mathsf{Nat}}) \arrow[d, "\mathsf{distr}"] & \\
& & \mathsf{Nat} \times 1 \arrow[r, "\mathsf{inl}"] \arrow[d, "\mathsf{snd}"] & \mathsf{Nat} \times 1 + \mathsf{Nat} \times \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \arrow[ddll, dashed, "{[\mathsf{one}, \mathsf{add} \circ (\mathsf{id} \times \mathsf{head})]}"] & \mathsf{Nat} \times \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \arrow[l, "\mathsf{inr}"'] \arrow[d, "\mathsf{id} \times \mathsf{value}"] \\
& & 1 \arrow[dl, "\mathsf{one}"] & & \mathsf{Nat} \times \mathsf{Nat} \arrow[dlll, "\mathsf{add}"] \\
& \mathsf{Nat} & & &
\end{tikzcd}$$

And our final function becomes:

$$f = \{\!\![[\mathsf{one}, [\mathsf{one} \circ \mathsf{snd}, \mathsf{add} \circ (\mathsf{id} \times \mathsf{head})] \circ \mathsf{distr} \circ \mathsf{out}]]\!\!\}$$

5.3 Colist recursion

However, we can do more with our colist of previous values than just look back a fixed number of times. We can actually look through the entire list every time. It is, however, no longer possible to apply value and branch a set number of times to get the previous values needed. We need some recursion to look through the entire colist. This creates a bit of a problem, the colist we have created is the terminal object of the $\mathsf{F}^{\times}_{\mathsf{C}}$-coalgebra, it is not the initial object. Thus, we cannot perform a catamorphism or any derivative of it on the $\mathsf{F}^{\times}_{\mathsf{C}}$-coalgebra. To bypass this, Kabanov and Vene described in a later paper that they where working in a category where initial and terminal objects coincide [5]. This is however not the case in our category **FPL**.

However, to show how recursion on a colist could be used we will give an example where we do assume that the initial and terminal objects coincide.

Example 5.5 (*Defining $f(n) = 2^n$*)

We will define the function $f(n) = 2^n$. which can be written as

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

Our function will be a histomorphism of Nat, thus we start by deciding how many base cases we need. We need at least one base case with a histomorphism and in this case no more with $f(0) = 1$. Thus, we can start with defining our function

$$\begin{aligned} f(n) &= \{\!|[\mathsf{one}, \varphi]|\!\} \\ \varphi &: \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \to \mathsf{Nat} \end{aligned}$$

We want to add all elements of $\nu\mathsf{N}^{\times}_{\mathsf{Nat}}$ and then take the successor for the +1.

$$\varphi = \mathsf{succ} \circ \mathsf{sum}$$

We would like to define sum as a catamorphism. As our functor we take $\mathsf{N}^{\times}_{\mathsf{C}}$ and the initial $\mathsf{N}^{\times}_{\mathsf{C}}$-algebra is $(\nu\mathsf{N}^{\times}_{\mathsf{C}}, \mathsf{out}^{-1})$. We can create our catamorphism diagram with the F-algebra $(C, [\mathsf{fst}, \mathsf{add}] \circ \mathsf{distr})$ to define sum

$$\begin{tikzcd}
A \times (1 + \nu\mathsf{N}^{\times}_{\mathsf{C}}) \arrow[r, "\mathsf{N}^{\times}_{\mathsf{C}}\, (\![\mathsf{fst},\mathsf{add}]\circ\mathsf{distr}]\!)"] \arrow[d, "\mathsf{out}^{-1}"] & A \times (1 + C) \arrow[d, "{[\mathsf{fst},\mathsf{add}]\circ\mathsf{distr}}"] \\
\nu\mathsf{N}^{\times}_{\mathsf{C}} \arrow[r, "(\![\mathsf{fst},\mathsf{add}]\circ\mathsf{distr}]\!)"] & C
\end{tikzcd}$$

Thus, our entire function becomes

$$f = \{\!|[\mathsf{one}, \mathsf{succ} \circ (\![\mathsf{fst}, \mathsf{add}] \circ \mathsf{distr}]\!)]|\!\}$$

We can show that the type of this function is correct with the following diagram

$$\begin{tikzcd}
1 \arrow[r, "\mathsf{inr}"] \arrow[ddr, "\mathsf{one}"] & 1 + \nu\mathsf{N}^{\times}_{\mathsf{Nat}} \arrow[dd, dashed, "{[\mathsf{one},\mathsf{succ} \circ \mathsf{sum}]}"] & \mathsf{N}^{\times}_{\mathsf{Nat}} \arrow[l, "\mathsf{inl}"'] \arrow[d, "\mathsf{sum}"] \\
 & & \mathsf{Nat} \arrow[dl, "\mathsf{succ}"] \\
 & \mathsf{Nat} &
\end{tikzcd}$$

Thus, we have defined our function using a histomorphism.

Chapter 6

Conclusions

We have seen how we can define recursive functions in a functional programming language using the building blocks of category theory. We have detailed a few methods of achieving this. We started with the work of Meijer et al. [8] in which the authors describe the basic building blocks of recursion and co-recursion, catamorphisms and anamorphism. We showed how these building blocks work and what laws apply to them. We then used them to create a few example functions with different functors as carriers. We showed how these two arrows can be combined to create many different functions. We also studied when we can combine them and when not as per the work of Capretta et al. [1]. Lastly, we showed a fourth technique to create course-of-value recursion by Uustalu and Vene in their paper [11] and later in Vene's PhD thesis [13].

6.1 Future work

There are more canned recursion schemes out there. Future work could focus on the line of work initiated by Vene with both Uustalu and Capretta. A closer look at monads and comonads as described in both the paper about hylomorphism from Capretta et al. [1] and a paper about recursion schemes from comonads from the three of them [12] could be fruitful. Kabanov and Vene analysed a expansion on histomorphisms, dynamorphism [5]. Hinze et al. proposed a new scheme [4], the conjugate hylomorphism. Hinze et al. also worked on expanding the histo- and dynamorphism in their paper [3].

All these canned recursion schemes and expansions of existing ones could be incorporated in a larger overview.

Bibliography

[1] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. "Recursive coalgebras from comonads". In: *Information and Computation.* Seventh Workshop on Coalgebraic Methods in Computer Science 2004 204.4 (Apr. 1, 2006), pp. 437–468. ISSN: 0890-5401. DOI: 10.1016/j.ic.2005.08.005. URL: http://www.sciencedirect.com/science/article/pii/S0890540105001963 (visited on 10/30/2019).

[2] J. H. Geuvers. "Inductive and coinductive types with iteration and recursion". In: *Informal Proceedings of the Workshop on Types for Proofs and Programs* (June 1992 1992), pp. 193–217.

[3] Ralf Hinze and Nicolas Wu. "Histo- and Dynamorphisms Revisited". In: *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming.* WGP '13. event-place: Boston, Massachusetts, USA. New York, NY, USA: ACM, 2013, pp. 1–12. ISBN: 978-1-4503-2389-5. DOI: 10.1145/2502488.2502496. URL: http://doi.acm.org/10.1145/2502488.2502496 (visited on 12/16/2019).

[4] Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. "Conjugate Hylomorphisms – Or: The Mother of All Structured Recursion Schemes". In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '15. event-place: Mumbai, India. New York, NY, USA: ACM, 2015, pp. 527–538. ISBN: 978-1-4503-3300-9. DOI: 10.1145/2676726.2676989. URL: http://doi.acm.org/10.1145/2676726.2676989 (visited on 09/26/2019).

[5] Jevgeni Kabanov and Varmo Vene. "Recursion Schemes for Dynamic Programming". In: *Mathematics of Program Construction.* Ed. by Tarmo Uustalu. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 235–252. ISBN: 978-3-540-35632-5. DOI: 10.1007/11783596_15.

[6] Joachim Lambek. "A fixpoint theorem for complete categories". In: *Mathematische Zeitschrift* 103.2 (Apr. 1, 1968), pp. 151–161. ISSN: 1432-1823. DOI: 10.1007/BF01110627. URL: https://doi.org/10.1007/BF01110627 (visited on 10/16/2019).

[7] Lambert Meertens. "Paramorphisms". In: *Formal Aspects of Computing* 4.5 (Sept. 1, 1992), pp. 413–424. ISSN: 1433-299X. DOI: 10.1007/BF01211391. URL: https://doi.org/10.1007/BF01211391 (visited on 12/07/2019).

[8] Erik Meijer, Maarten Fokkinga, and Ross Paterson. "Functional programming with bananas, lenses, envelopes and barbed wire". In: *Functional Programming Languages and Computer Architecture*. Ed. by John Hughes. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 124–144. ISBN: 978-3-540-47599-6.

[9] Bartosz Milewski. *Category Theory for Programmers*. Bartosz Milewski's Programming Cafe. Oct. 28, 2014. URL: https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/.

[10] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991. 124 pp. ISBN: 978-0-262-66071-6.

[11] Tarmo Uustalu and Varmo Vene. "Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically". In: *Informatica* 10.1 (Jan. 1, 1999), pp. 5–26. ISSN: 0868-4952. DOI: 10.3233/INF-1999-10102. URL: https://content.iospress.com/articles/informatica/inf10-1-02 (visited on 12/07/2019).

[12] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. "Recursion schemes from comonads". In: *Nordic Journal of Computing* 8.3 (2001), pp. 366–390.

[13] Varmo Vene. "Categorical programming with inductive and coinductive types". PhD thesis. Estonia: University of Tartu, May 26, 2000. 118 pp.

List of Theorems