Universiteit Utrecht

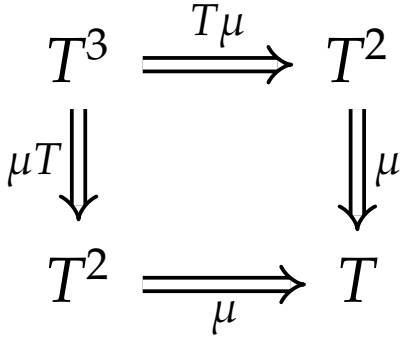
Faculteit Bètawetenschappen

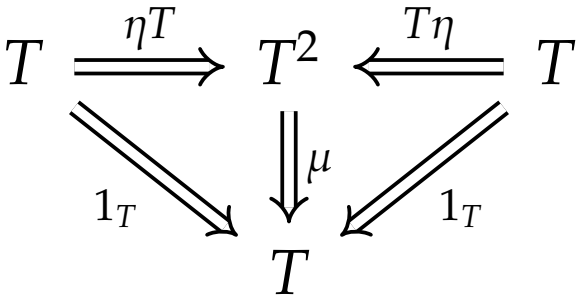
The categorical origin of monads in Haskell

Bachelor Thesis

Marien Matser

Wiskunde





Supervisor:

Dr. Lennart Meier
Departement wiskunde

June 11, 2021

Abstract

Category theory is an abstract part of mathematics. It is based on categories, which consist of just objects and morphisms between these functions. When talking about categories you will most likely also be talking about functors, which are the morphisms between the categories. These functors can be combined to create a relation between two categories called adjunctions. Using all these concepts we can create some extra structure on categories called monads. These monads can then be used to create algebra that can suddenly introduce very known concepts like groups. We can also use the monads to create other interesting categories, namely the Kleisli categories.

The type system of the functional programming language Haskell can actually be seen as a category. Because of this most of our categorical concepts also have their place in Haskell. Some concepts like functors and monads are very commonly used, other concepts like adjunctions and algebras are defined but not that commonly used.

Contents

1 Introduction

Most of the popular programming languages are object oriented programming languages, like C, C# and Java. This means that these are based on objects that can be passed on and edited by different parts of the code. Another popular kind of programming languages are functional programming languages. These programming languages are based on functions and composing these functions. One of the most popular functional programming languages is Haskell.

Most people who have learned programming in Haskell, or at least tried to, have come across a concept called monads. In Haskell these monads are often used for things like working with side effects or in other ways adding extra functionalities to functions. Monads however are a very complicated concept to properly understand. It is often not that hard to just use them but actually understanding what they are and how they work can be very difficult. This can partially be explained by the fact that monads are actually based on a very abstract concept within mathematics and most programmers are more focused on the more practical side of things.

The fact that these programming languages are based on functions can already give the intuition that it might have something in common with mathematics, since a lot of mathematics is also based on functions and compositions of these functions. Haskell is in fact based on a part of mathematics called category theory. This is an abstract part of mathematics that is able to encapsulate a lot of other parts of mathematics.

In this thesis we will be exploring how we can view Haskell from category theory. So we will be investigating how we can see Haskell as a category of its own and how a lot of common definitions in category theory are implemented in Haskell. Monads are also one of these definitions within category theory that we will be looking at. So the aim of this thesis is to have a better understanding of where these monads in Haskell came from and what they can be used for.

For this thesis it is not necessary to know how to program in Haskell, since the necessary concepts in Haskell will be discussed. There will however be a lot of important concepts of Haskell that will not be discussed in this thesis, like laziness, concurrency and more. To learn more about Haskell there are a lot of tutorials on the internet to follow. Some of these are listed on Haskells website [1].

The category theory in this thesis is based on the book Category Theory In Context by Emily Riehl [2]. The Haskell part is based on the book Category Theory for Programmers by Bartosz Milewski [3]. These books talk about category theory and how it is used in Haskell respectively. This thesis will describe the most important definitions from these books.

2 Categories

Before we are able to talk about monads and other concepts within category theory we will first have to discuss the basis of this area of mathematics. As the name suggests it is based on categories, so in this chapter we will be looking at what these categories are [2, Section 1.1]. After this we will look at how we can see the programming language Haskell as a category itself [3, Chapter 2] and finally we will be looking at the concept of monoids [2, Section 1.6] [3, Section 3.4].

2.1 Definition

We first need the definition of a category.

Definition 2.1.1. A **category** consists of the following:

- a collection of **objects**, and
- a collection of **morphisms**,

such that:

- each morphism has specified **domain** and **codomain** objects. The notation $f : X \to Y$ here means that f is a morphism with domain X and codomain Y,
- each object X has a designated **identity morphism** $1_X : X \to X$,
- for any pair of morphisms f, g where the codomain of f is equal to the domain of g, there exists a specified **composite morphism** $g \cdot f$ or gf with the domain equal to that of f and codomain equal to that of g.

This data is subject to the following two axioms:

- for any $f : X \to Y$, the composites $1_Y f$ and $f 1_X$ are both equal to f, and
- for any composable triple of morphisms f, g, h the composites $h(gf)$ and $(hg)f$ are equal and thus denoted as hgf.

If we have two objects X, Y in a category C we can write the set of morphisms from X to Y in C as $C(X, Y)$. A category is an abstract definition that encapsulates a lot of different concepts within mathematics. We give some examples of these concepts.

Example 2.1.2. The category **Set** where as the name might suggest objects are sets. The morphism are functions between sets. △

Example 2.1.3. Similarly to sets we can also have topological spaces which form the category **Top**, where morphisms are continuous functions. △

Example 2.1.4. Another example is the category **Group** formed by groups as the objects and group homomorphisms as morphisms. △

We can also look at whether objects are very similar. For this we can look at isomorphisms.

Definition 2.1.5. An **isomorphism** is a morphism $f\colon X \to Y$ for which there exists a morphisms $g\colon Y \to X$ such that $gf = 1_X$ and $fg = 1_Y$. If there is an isomorphism between objects X and Y, we call them **isomorphic**, which we write as $X \cong Y$.

2.2 Hask

Another example of a category is the category **Hask** which is the category used by Haskell. In this category the objects are the types, like integers, characters and a lot more. These objects can be seen as sets with all the possible values for a certain type. The morphisms are the functions that are defined in Haskell or that you write yourself. In Haskell we use the notation of `square :: Int -> Int` as a type reference for the function `square` from type `Int` to type `Int`. This function that squares the given integer can be defined as follows:

```
square :: Int -> Int
square x = x * x
```

Here we define our function by saying that if `square` is applied to `x` the result will be `x * x`. Inside the type definition we can also give a type variable instead of a specific type which allows us to define a functions for every type. This can be used to very easily define the identity function for every type as follows:

```
id :: a -> a
id x = x
```

These functions we are creating can also be called inside other functions. This can be used to define the function composition. Given two function `f :: a -> b` and `g :: b -> c` we can define a function `h :: a -> c` as follows:

```
h :: a -> c
h x = g (f x)
```

In Haskell we also have a special function `.` defined for composition, which can be used to rewrite `h` as `h = g . f`. Looking at the first definition of `h` we can see that if we have the identity for `f` we can just replace `f a` by `a`. This shows us that `g . id` will be the same as `g`. Similarly we can see that `id . f` will be the same as `f`.

We can also have type definitions that have more than two types, like with the function `add` that adds two integers:

```
add :: Int -> Int -> Int
add x y = x + y
```

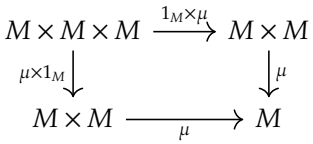
We can interpret this function as defining for every integer a function that adds that integer to a given integer. So you could see it as `add :: Int -> (Int -> Int)` where `(Int -> Int)` will be the returned function.

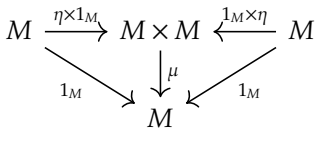
The **Hask** category is very similar to the category **Set**. The only thing is that if we would be using **Set** for Haskell we would have an issue with non terminating functions, which are functions that won't always give back a result, in mathematics this is not a thing but when programming it is very possible. Because of this in **Hask** every type also has an extra value called the bottom which represents a non terminating function. Most of the times we won't be seeing this value since it is just an extra value for every type.

2.3 Monoids

Since we now know what a category is we can look at a concept within category theory called monoids:

Definition 2.3.1. A **monoid** is a set M together with two morphisms $\mu : M \times M \to M$ and $\eta : 1 \to M$ so that the following diagrams commute:





Here μ defines a "multiplication" on M and η identifies an element in M. The diagrams require that this multiplication is associative and that the element identified by η acts as an identity element. Not only sets can have monoids. We can for example also have monoids on topological spaces, where it is just the same definition as monoids on sets, but on the underlying sets.

A monoid can also be seen as a category with a single object. The set of the monoid will then be the morphisms in this category. The identity will be defined by the identity morphism and the multiplication is defined by the composition of morphisms. The diagrams of a monoid then also correspond with the axioms of a category.

The definition of a monoid in Haskell uses a typeclass, which is a definition that every type that follows this typeclass has to have the specific requirements from that class implemented. A simple typeclass is the typeclass for all types that can be evaluated for equality:

```
class Eq a where
    (==) :: a -> a -> Bool
```

This means that for every type that follows this typeclass has the operator `(==)` defined on it. The definition of a monoid in Haskell is:

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m
```

We can clearly see that `mempty` corresponds to the identity element and the `mappend` corresponds to the multiplication. As the names suggest in Haskell a monoid is often used with an empty element and a function that can append two values together. Since this definition uses a typeclass we can define an example as an instance of this.

Example 2.3.2. We can for example look at strings, which are lists of characters. The instance of a string will be:

```
instance Monoid String where
    mempty  = ""
    mappend = (++)
```

We can see that the `mempty` is an empty string and the `mappend` is the concatenation of strings where it takes two strings and concatenates them together. △

3 Functors

We now know what categories are, but now we want to consider a concept similar to morphisms except now between categories. For that we will be looking at functors in this chapter [2, Section 1.3]. We will also be looking at the concept of functors in Haskell where they are viewed as containers [3, Chapter 7]. After this we will also be taking it one step further and introduce morphisms between these functor by looking at natural transformations [2, Section 1.4] [3, Chapter 10].

3.1 Functors

To look at relations between categories we need the definition of a functor:

Definition 3.1.1. Given two categories C and D, a **functor** $F : C \to D$ between these two categories consists of the following data:

- An object $Fc \in D$, for each object $c \in C$.
- A morphism $Ff : Fc \to Fc' \in D$, for each morphism $f : c \to c' \in C$.

These components need to satisfy the following **functoriality axioms**:

- For any composable pair of morphisms $f, g \in C$, we need $Fg \cdot Ff = F(g \cdot f)$.
- For each object $c \in C$, $F(1_c) = 1_{Fc}$.

We can see here that a functor will map every component of a category to a corresponding component in the target category. So objects will be mapped to objects and morphism are mapped to morphisms. The functoriality axioms make sure that axioms for a category are kept nicely. So we want composable morphisms to still be composable after applying the functor and the composition should be mapped to the composition of the images. We also want identity morphisms to still be the identity morphism for the image of the object.
A functor where the domain is equal to its codomain we call an **endofunctor**.
Here are some examples of functors:

Example 3.1.2. We can have an endofunctor on **Set** that maps every object to its power set. A function $f : A \to B$ will be mapped to the function that maps every subset of a power set to the image of this power set, so $A' \subset A$ will be mapped to $f(A')$. △

Example 3.1.3. For most categories that are based on sets, like Group or Top, we can create a **forgetful functor** by just forgetting about the extra structure defined in these categories. For example for Group it will just send a group to its set of elements and a homomorphism to its function. And for Top a topological space will be mapped to its set of elements and the continuous function will be mapped to that same function. △

3.2 Functors in Haskell

Haskell also has its own definition of a functor. Since Haskell uses a single category, a functor in Haskell will be an endofunctor. The definition of such a functor is as follows:

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

This means that in order to be a functor you will need this `fmap` function. The first argument in the type definition `(a -> b)` is a function that this function takes in. We can look at this function as returning a function of type `f a -> f b`. If we look at it this way we can see that this is the same as the requirements for the morphisms in the definition of a functor where for every morphism in C we need a corresponding morphism in D.

The Haskell definition of a functor does not require a functor to satisfy the functoriality axioms, this is something the person implementing a functor should look at. In Haskell functors are often viewed as containers where `fmap` can be used to apply a function to every element inside this functor. We can now look at some examples:

Example 3.2.1. We can look at the maybe functor, which uses the maybe datatype which can be either nothing or just a value:

```
data Maybe a = Nothing | Just a
```

The functor will then be:

```
instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

Here we are pattern matching on the value to apply our function on. So if this value is Nothing we use the first definition and if it is just a value we use the second definition. The underscore in the first definition means that this can be any value that we won't be using. So if the value is nothing this function will return nothing and if it is a value the passed function is applied to the value inside this `Just`. △

Example 3.2.2. We can now also look at the list functor. This functor uses the lists in Haskell. These lists are written using square brackets (e.g. `[]`, `[1]`, `[1,2,3]`). Lists are created by adding element in front of lists using a colon for the notation. So a singleton list can be created using `1:[]` and a longer list can be created using `1:[2,3]`. Lists are very common in Haskell, because of they have their own name for `fmap`, namely `map`. This function just applies a given function to every element in the list. So the functor instance will then be:

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

△

3.3 Natural transformations

For functors we can also define some sort of morphisms. These morphisms are natural transformations:

Definition 3.3.1. Given categories C and D and functors $F, G : C \rightrightarrows D$, a **natural transformation** $\alpha : F \Rightarrow G$ consists of: an arrow $\alpha_c : Fc \to Gc$ in D for each object $c \in C$, such that for any morphism $f : c \to c'$ in C, the following square of morphisms commutes in D:

$$\begin{array}{ccc} Fc & \xrightarrow{\alpha_c} & Gc \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ Fc' & \xrightarrow[\alpha_{c'}]{} & Gc' \end{array}$$

The collection of these arrows defines the **components** of the natural transformation,

Since we can see these natural transformations as morphisms between functors we can also use these to create a category of functors, using functors as objects and natural transformations as morphisms. Such a category D^C contains functors from C to D as objects and natural transformations between these functors as morphisms.

Example 3.3.2. We can have a natural transformation from the identity functor on set which just maps every set to itself and the power set functor from Example 3.1.2. An arrow $\alpha_A : A \to PA$ in this transformation will be a function that maps an element a in A to its singleton set in the power set of A. △

In Haskell there is not really a definition for natural transformations. We can however have a look at what they would look like if there was. We know that a natural transformation will be between two functors F and G. We will first need a function for every object in the source category. Such a function would have a type signature like this:

```
alpha :: F a -> G a
```

The square diagram of our definition can be rewritten to the requirement $Gf \cdot \alpha_c = \alpha_{c'} \cdot Ff$. If we would write this in Haskell we would get `fmap f . alpha = alpha . fmap f`. Due to the way Haskell evaluates the `alpha` function this will always hold.

Since we know that in Haskell functors are used as containers we can think of this as moving the elements to a different container. Because of this we can also intuitively see that the square diagram from the definition of a natural transformation holds. Because it does not matter whether you first apply a function to every element in a container and then reorganize these element into a different container or you first reorganize the elements and then apply the function to every element.

Example 3.3.3. We can look at a natural transformation from a list to the maybe functor. We can create this natural transformation by using it to safely take the first element of a list. In Haskell the separate functions would be defined as:

```
safeHead :: [a] -> Maybe a
safeHead []     = Nothing
safeHEad (x:xs) = Just x
```

Here the first line is the type signature. The second line says that if we have an empty list the result should be `Nothing`. The last line uses pattern matching where it says that the list should be of the form with first an element and then the rest of the list. So here `x` is the first element of the list and `xs` is the rest of the list. Now if this is the case we just take this first element. △

4 Adjunctions

Before we will be talking about monads, we will first have a look at adjunctions [2, Chapter 4]. These can be used to create some sort of relation between two categories. This relation can later be used to create a monad. Haskell does not really use adjunctions partly because it is only a single category, but we will still look at a definition of adjunctions in Haskell [3, Chapter 18].

4.1 Adjunctions

Definition 4.1.1. An **adjunction** consists of a pair of functors $F\colon C \to D$ and $G\colon D \to C$, together with for every $c \in C$ and $d \in D$ an isomorphism

$$D(Fc, d) \cong C(c, Gd)$$

that is natural in both variables. In this definition F is called the **left adjoint** to G and G is the **right adjoint** to F. The morphisms that correspond under the isomorphism are called **adjunct** or the **transpose** of each other.

An adjunction with functors F and G can also be written as $F \dashv G$. An example of some adjunctions is:

Example 4.1.2. For an example of an adjunction we can look at the forgetful functor $U\colon \mathbf{Top} \to \mathbf{Set}$ for topological spaces. This functor admits both a left and a right adjoint. To get the corresponding left adjoint we need a functor V from **Set** to **Top** for which holds that if we have a set S and a topological space T we want to have a correspondence between continuous functions $V(S) \to T$ and functions $S \to U(T)$. We can use the discrete topology for this, because any function of the form $S \to U(T)$ is also a continuous function from the discrete space of S to T.

To construct a right adjoint we also need a functor W from **Set** to **Top**. We now however need a correspondence between functions $U(T) \to S$ and continuous functions $T \to W(S)$. We can now use the trivial topology for this, because any function of the form $U(T) \to S$ is also a continuous function from T to the trivial space of S. △

We can also have a simpler definition of an adjunction as follows:

Definition 4.1.3. An **adjunction** consists of a pair of functors $F\colon C \to D$ and $G\colon D \to C$, together with two natural transformations $\eta\colon 1_C \Rightarrow GF$ and $\epsilon\colon FG \Rightarrow 1_D$, such that they satisfy the **triangle identities:**

$$\begin{array}{ccc} F & \xRightarrow{F\eta} & FGF \\ & \searrow^{1_F} & \Downarrow \epsilon F \\ & & F \end{array} \qquad \begin{array}{ccc} G & \xRightarrow{\eta G} & GFG \\ & \searrow^{1_G} & \Downarrow G\epsilon \\ & & G \end{array}$$

Here η is called the **unit** and ϵ the **counit** of the adjunction.

Now we want to prove that these definitions are actually equal using the following proposition:

Proposition 4.1.4. *Given a pair of functors $F\colon C \rightleftarrows D\colon G$, there exists a natural isomorphism $D(Fc, d) \cong C(c, Gd)$ if an only if there exists a pair of natural transformations $\eta\colon 1_C \Rightarrow GF$ and $\epsilon\colon FG \Rightarrow 1_D$ satisfying the triangle identities.*

Proof. Given a natural isomorphism $\phi\colon D(Fc, d) \cong C(c, Gd)$, we want to prove that there is a pair of natural transformations $\eta\colon 1_C \Rightarrow GF$ and $\epsilon\colon FG \Rightarrow 1_D$, such that $1_F = \epsilon F \cdot F\eta$ and $1_G = G\epsilon \cdot \eta G$. If we now take d to be Fc in ϕ we get $D(Fc, Fc)$ on the left side and $C(c, GFc)$ on the right side. So we can now take the morphism in the right set corresponding to the identity on Fc: $\phi(1_{Fc})$. We see that this morphism will be precisely of the form $\eta_c\colon c \to GFc$, so we can use this as a component for our natural transformation. Similarly, if the take c to be Gd we can get the components $\epsilon_d\colon d \to FGd$ to be the morphism corresponding to the identity on Gd: $\phi^{-1}(1_G d)$. Since we define our components according to ϕ, we can also write ϕ in terms of η as $\phi(f) = Gf \cdot \eta_c$ for $f\colon Fc \to d$ or in terms of ϵ as $\phi^{-1}(g) = \epsilon_d \cdot Fg$ for $g\colon d \to Gc$. If we now take $c = Gd$, our definition of ϵ gives us using the formula of ϕ in terms of η:

$$1_G = \phi(\epsilon) = G\epsilon \cdot \eta G.$$

This is precisely our triangle identity. Similarly if we take $d = Fc$ we get

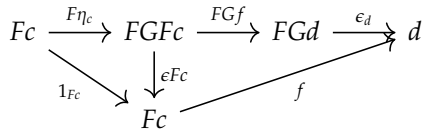
$$1_F = \phi^{-1}(\eta) = \epsilon F \cdot F\eta.$$

So we now know that given a natural isomorphism $\phi\colon D(Fc, d) \cong C(c, Gd)$ there is a pair of natural transformations $\eta\colon 1_C \Rightarrow GF$ and $\epsilon\colon FG \Rightarrow 1_D$, such that $1_F = \epsilon F \cdot F\eta$ and $1_G = G\epsilon \cdot \eta G$.[4, Section IV.1]

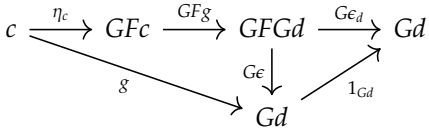
Now we only need to prove that given a pair of natural transformations that satisfy the triangle identities, there is a natural isomorphism $\phi\colon D(Fc, d) \cong C(c, Gd)$. Given $f\colon Fc \to d$ and $g\colon c \to Gd$, we can define this ϕ in terms of our natural transformations to be the following:

$$\phi f := \; c \xrightarrow{\eta_c} GFc \xrightarrow{Gf} Gd \qquad\qquad \phi^{-1} g := \; Fc \xrightarrow{Fg} FGd \xrightarrow{\epsilon_d} Gd$$

Now we need to see whether these are indeed each others inverses. If we look at $\phi^{-1}(\phi f)$ and the first triangle identity we can get the following diagram:



We can see that the top composite is our $\phi^{-1}(\phi f)$, while the rest is added using the triangle identity. We can now see that $\phi^{-1}(\phi f) = f \cdot 1_{Fc}$. So we can indeed see that in this direction these definitions are each others inverses. In the same way we can use the other triangle identity to see that in the other direction they are also each others inverses:



So we now know that we can indeed define a natural isomorphism $\phi\colon D(Fc, d) \cong C(c, Gd)$ from a pair of natural transformations that satisfy the triangle identities. □

Using this proposition we can see that our two definitions of adjunctions are indeed equal.

4.2 Adjunctions in Haskell

In Haskell adjunctions are not used very often. This is partly because we are considering only a single category **Hask**, so all adjoints will be endofunctors. There is however a definition for adjunctions. We have seen that in category theory there are two possible definitions for adjunctions. Haskell also has two possible definitions:

```
class (Functor f, Representable u) => Adjunction f u | f -> u, u -> f where
    leftAdjunct  :: (f a -> b) -> a -> u b
    rightAdjunct :: (a -> u b) -> f a -> b

class (Functor f, Representable u) => Adjunction f u | f -> u, u -> f where
    unit   :: a -> u (f a)
    counit :: f (u a) -> a
```

These definitions are based on two parameters `f` and `u`. The first one is a functor as we would expect. The representable is also a functor, but one with some extra things defined on it to make sure it is suitable as a right adjoint. Furthermore the conditions after the vertical bar specify some of the required dependencies. For instance, `f -> u` requires that `u` is determined by `f`. Conversely, `u -> f` requires that if we know `u`, we can determine `f`. This ensures that we can't define multiple adjunctions with the same left or right adjoint. So if you know the left or right adjoint, you also know the other one if there is an adjunction for it.

When implementing an adjunction we can choose either one of these definitions. This works because we can define one in terms of the other and vice versa. This is done using the following definitions:

```
unit           = leftAdjunct id
counit         = rightAdjunct id
leftAdjunct f  = fmap f . unit
rightAdjunct f = counit . fmap f
```

An example of an adjunction in Haskell is currying and uncurrying:

Example 4.2.1. In Haskell, tuples are commonly used. These are types that consist of multiple items, like `(Int, Int)` is the type and an instance is `(1,2)`. Some functions take in these tuples as argument. We could however also define the same function but passing the elements of the tuple separately. For this we can curry a function to take in two separate elements instead of a tuple. The opposite is called uncurry, where we change a function to take in a tuple instead of two separate elements. There are functions to do this in Haskell, but we can also define an adjunction for this:

```
instance Adjunction ((,) e) ((->) e) where
  leftAdjunct f a e     = f (e, a)
  rightAdjunct f (e, a) = f a e
```

△

5 Monads

Now that we know what categories, functors and adjunctions are we can finally look at monads. So in this chapter we will look at the way monads are defined in category theory [2, Section 5.1]. After this we will look at how these monads are used within Haskell [3, Chapter 20, 22]. After this we will also have a look at defining monads from adjunctions in both category theory and Haskell [2, Section 5.1] [3, Chapter 20, 22].

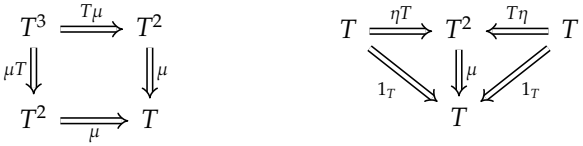
5.1 Monads

The technical definition of a monad is as follows:

Definition 5.1.1. A **monad** on a category C consists of

- an endofunctor $T : C \to C$,
- a **unit** natural transformation $\eta : 1_C \Rightarrow T$ and
- a **multiplication** natural transformation $\mu : T^2 \Rightarrow T$,

such that the following diagrams commute in C^C:



Note here that for every object in C the unit transformation η gives us a morphism from it to the object it is mapped to by T. While the multiplication transformation μ gives us for every object in C a morphism between the object we get after applying T twice and the object after applying T once.
Also note that the left diagram ensures that it does not matter whether we see the T^3 as T^2T and use μ on the first two applications of T or see it as TT^2 and use μ on the last two when we want to reduce it from three applications to two and then one.
Finally we can also note that these diagrams are very similar to the diagrams in the definition of a monoid(2.3.1). Because of this we can also see a monad on C as a monoid in the category of endofunctors C^C.
Some examples of monads are:

Example 5.1.2. We can create the **maybe monad** using the endofunctor $(-)_+ : \mathbf{Set} \to \mathbf{Set}$ that adds a new disjoint point to a set. The arrows in the unit transformation are given by $\eta_A : A \to A_+$ since A is included in A_+. The arrows of the multiplication transformation can be given by $\mu_A : (A_+)_+ \to A_+$ which will map A to itself and the two extra points to the same extra point in A_+. △

Example 5.1.3. The **free monoid monad** or **list monad** can be created using the endofunctor $T : \mathbf{Set} \to \mathbf{Set}$ defined as

$$TA \coloneqq \coprod_{n \geq 0} A^n,$$

here $\coprod$ is the disjoint union. This functor will create a set of all finite lists of elements in A. The arrows of the unit transformation $\eta_A : A \to TA$ will just map all elements in A to the corresponding singleton list in TA. For the multiplication we have arrows $\mu_A : T^2A \to TA$ that will map a list of lists to the concatenation of these lists. △

Example 5.1.4. Similar to monoids there is also a **free group monad**. This monad can be created using the free group functor that maps a set to its free group and the forgetful functor on groups, which maps a group to its underlying set. The composition of these functors creates an endofunctor on **Set** that maps a set A to the set of finite words with letters in A and their inverses that are not equal to another word following the group axioms (like xyy^{-1} equals x). The unit transformation will map every element in A to the corresponding word of length one. The multiplication transformation maps a string of strings to the concatenation of the strings that from the letters in the overall string. △

5.2 Monads in Haskell

Now lets look at the way Haskell uses monads. The standard definition for a monad in Haskell is:

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

In this definition the (>>=) operator is often called the bind operator. Although programmers often do not really care about demanding our extra requirements from category theory, a monad does have some requirements that an instance of a monad should satisfy:

```
return a >>= k          = k a
m >>= return            = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

The first two lines require that the `return` function works as an identity for the bind operator. The last line uses a lambda function. This means that it will return a function that takes in a variable `x` and uses this variable to create a result. This line demands associativity for the bind operator.

Looking at the return function we can notice that this is very similar to our unit transformation in the definition of a monad. The bind operator however does not really look like the multiplication transformation. For this we can create another function:

```
join :: m (m a) -> m a
```

This function does look a lot like our multiplication transformation. We can also actually define our bind operator in terms of this join operator using the following definition:

```
ma >>= f = join (fmap f ma)
```

This new definition first uses `fmap` to go from a monad of a type (`m a`) to the monad twice applied to `b` type (`m (m b)`), after this the join is used to make an `m b` out of this. This leads us to another way of defining a monad using this new `join` function:

```
class Functor m => Monad m where
    join   :: m (m a) -> m a
    return :: a -> m a
```

This definition also requires `m` to be a functor, just like our categorical definition of a monad. The other definition doe not require `m` to be a functor because the fact that it has the bind operator defined on it means that it has to be a functor. We can even define the `fmap` in terms of the bind operator:

```
fmap f ma = ma >>= return . f
```

We can also define our `join` function in terms of the bind operator as follows:

```
join x = x >>= id
```

We now need to check whether if we define a monad using one of our definitions and then use our cross references of join and bind twice we can get the same monad back. When we take a monad defined by `join` we can get:

```
join' x = join (fmap id x)
```

We see that `fmap id x` will just be `x` because `id` is the identity. Now the other way around when also using our definition of `fmap` in terms of bind we get:

```
ma >>=' f = (ma >>= return . f) >>= id
```

We see that because `return` behaves as an identity we get that `(ma >>= return . f)` is equal to `(ma >>= f)`. We also see that using the bind operator with the identity will just yield the thing it is applies to.

So we now know that these two definitions are equal, since if we have one we can create the other from it and if we do this twice in a row we get the same monad back. When programming it is often easier to use the bind operator which is why that definition is more common, but as we have seen it does not matter which definition you would use.

Now we do know the definition, but we do not yet exactly know how one of these monads would work. For this we will look at two examples:

Example 5.2.1. The monad we have seen in Example 5.1.2 is actually also a common monad in Haskell. The accompanying functor we have also already seen in Example 3.2.1. The extra value we add in this monad is called `Nothing` in Haskell. We have seen that the unit transformation maps an element in a set to that same element in the set with a extra element. For the `return` function this would mean that it will have to return the element inside a `Just`. This give us the following:

```
return x = Just x
```

For the multiplication transformation we have seen that if it is one of the extra points it should be mapped to the extra point in the result and if it is just a value it should be mapped to that value. We can define this in Haskell by pattern matching on the value of the outside maybe monad. If this is `Nothing` the result should be `Nothing` and if this is `Just` we can take the value of the inside monad as our result. This gives us the following definition:

```
join Nothing = Nothing
join Just x  = x
```

If we now combine these functions we get the following instance of a monad:

```
instance Monad Maybe where
    join Nothing = Nothing
    join Just x  = x
    return x     = Just x
```

△

Example 5.2.2. We can now also look at the monad defined in Example 5.1.3. This monad uses the list functor from Example 3.2.2. As we have seen the unit transformation maps an element to its singleton list, so the `return` function will be:

```
return x = [x]
```

The multiplication transformation will map a list of list to a list by concatenating the internal lists. Haskell again already has a standard function for this called `concat` which will do exactly this. The definition of this monad is therefore:

```
instance Monad [] where
    join x   = concat x
    return x = [x]
```

△

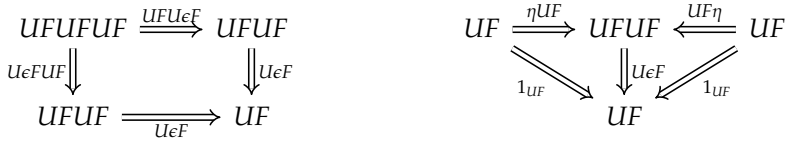
5.3 Monads from adjunctions

We can also create monads from adjunctions. This can be done using the following lemma:

Lemma 5.3.1. *Given an adjunction $F \dashv U$ from C to D with unit $\eta\colon 1_C \Rightarrow UF$ and counit $\epsilon\colon FU \Rightarrow 1_D$, we can create a monad on category C with:*

- *the endofunctor T is defined as UF,*
- *the unit η serves as the unit transformation $\eta\colon 1_C \Rightarrow T$ of the monad,*
- *the multiplication transformation is defined as $U\epsilon F\colon UFUF \Rightarrow UF$.*

Proof. To prove that this will actually give us a monad, we have to prove that the following diagrams commute in C^C:



We can see that the right diagram will commute because of the triangle identities of an adjunction. The left diagram commutes because of the naturality of $U\epsilon\colon UFU \Rightarrow U$. □

We can see that this definition might make it easier to define a monad. We can now look at some adjunctions that can be used to induce the monads from Section 5.1:

Example 5.3.2. We can have a free $\dashv$ forgetful adjunction between pointed sets that have an extra elements and sets. This adjunction induces the monad from Example 5.1.2. △

Example 5.3.3. There also is a free $\dashv$ forgetful adjunction between monoids and sets. As expected this adjunction induces the monad from Example 5.1.3. △

Example 5.3.4. Finally the monad from Example 5.1.4 can be induced by the free $\dashv$ forgetful adjunction between sets and groups. We can notice that this is actually also what we used to define the endofunctor by using the free and forgetful functors for **Group**. △

As Haskell does not use adjunctions that much, there is no definition for creating a monad form adjunctions. We can however create one ourselves using the definition from category theory. The endofunctor is just the composition of the functors in our adjunctions. The `return` is the unit form our adjunction. Finally the `join` is a bit more complicated, but can be created using the lifted version of our counit. The would lead us to the following instance of a monad:

```
instance Adjoint f u => Monad (u.f) where
    join x   = fmap counit x
    return x = unit x
```

6 Algebras

Now that we know what monads are we can look at algebras over these monads [2, Section 5.2]. We will see that other concepts within category theory and other parts of mathematics can be seen as these algebras over monads. We will also again be looking at how these algebras can be used in Haskell [3, Chapter 24, 25]. After this we will have a look at constructing an adjunction to the category of algebras using a monad [2, Section 5.2].

6.1 T-algebras

If we want to look at algebras we will first need the definition of the Eilenberg-Moore category:

Definition 6.1.1. Let C be a category with a monad (T, η, μ). The **Eilenberg-Moore category** for T also called the **category of T-algebras** is the category C^T consisting of:

- objects that are pairs $(A \in C, a : TA \to A)$ so that the following diagrams commute in C:

$$\begin{array}{ccc} A & \xrightarrow{\eta_A} & TA \\ & \searrow^{1_A} & \downarrow{\scriptstyle a} \\ & & A \end{array} \qquad \begin{array}{ccc} T^2A & \xrightarrow{\mu_A} & TA \\ {\scriptstyle Ta}\downarrow & & \downarrow{\scriptstyle a} \\ TA & \xrightarrow[a]{} & A \end{array}$$

- morphisms $f\colon (A, a) \to (B, b)$ that are T-algebra homomorphisms. This means they are maps $f\colon A \to B$ in C such that the square

$$\begin{array}{ccc} TA & \xrightarrow{Tf} & TB \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A & \xrightarrow[f]{} & B \end{array} \tag{6.1}$$

  commutes. The composition and identities are as in C.

Note that the objects in this category can be seen as the algebras. So we can see that an algebra will be a pair $(A \in C, a\colon TA \to A)$. Some examples of these categories of algebras are:

Example 6.1.2. We can look at algebras over the free monoid monad from Example 5.1.3. An algebra will be a set A with a map $a\colon \coprod_{n\geq 0} A^n \to A$. This map consists of components of the form $a_n : A^n \to A$ that specify n-ary operations on A for each n. The triangle diagram tells us that the unary operation $a_1\colon A \to A$ has to be the identity on A. The square however tells us that it will be the same to first concatenate the sublists inside the list of lists using μ_A and then use a on the resulting list as first applying a to all sub lists and then applying a to this result. We can also see that there will be a nullary operation that picks an element in A.

We now notice that this is very similar to the definition of a monoid (2.3.1). So if we have a monoid on a set M with multiplication $\mu\colon M \times M \to M$ and unit $\eta\colon 1 \to M$ we can look at creating one of these algebras from this. We can see that the unit can be used for the nullary operation: $a_0 = \eta$. The unary operation is the identity by definition. For the binary operation a_2 we can use the multiplication since this is also a binary operation $a_2 = \mu$. The higher order operations can be determined by iterating over this binary multiplication multiple times, for example a_3 will just be μ applied two times. Because we create the higher order operations in this way we can also see that the right square diagram for an algebra will hold.

We can now also look at if we have an algebra (A, a) over the free monoid monad, whether we can create a monoid from this. For the unit of the monoid we can simply use the nullary operation: $\eta = a_0$. Now for the multiplication we can use the binary operation: $\mu = a_2$. Since the square diagram demands associativity we also know that the diagrams for a monoid commute. So we can now see that an algebra over the free monoid monad is precisely a monoid.

Now we can look at the morphisms in the Eilenberg-Moore category over the free monoid monad. We see that these are morphisms based on functions $f : A \to B$. So Given algebras (A, a) and (B, b) and a function $f : A \to B$, we want to make sure that if we have a list of elements in A it it does not matter whether we first map all the elements in this list to the corresponding element in B using f and then evaluate this list using b or first evaluate this list using a and then map this result to the corresponding value in B using f, so $b \cdot Tf = f \cdot a$. This is exactly the same as the requirements for a monoid homomorphisms. We now know that objects in the Eilenberg-Moore category of the free monoid monad are precisely monoids and the morphisms in this category are precisely monoid homomorphisms. We can therefore see that the Eilenberg-Moore category for the free monoid monad is precisely the category **Monoid** of monoids. △

Example 6.1.3. We can now also look at the free group monad from Example 5.1.4. An algebra on this monad will be a set with a function $a : TA \to A$ from the finite words of letters in A to A. Since a word can be seen as the product of letters, we can see a as a function that evaluates these products. So a word will be mapped to the result of that multiplication. The triangle diagram of an algebra makes sure that a word of length 1 with a non inverse letter will be mapped to that element in A. The square diagram makes sure that if you have a word of words it does not matter whether you first evaluate the internal words and then the result or whether you first concatenate the words to a single word to then evaluate the result, so $a \cdot Ta = a \cdot \mu_A$. Since we also have the words of length one that consist of an inverse element, we also know that these elements are mapped to an element in A, so we know the inverse element of all elements in A. Finally we also have a word of length zero which will be mapped to the identity element.

These requirements are very similar to that of a group itself. We can look at whether if we have a group G with binary operator $(\circ)$, we can create an algebra over the free group monad from this. As the set we can just take the elements of G. Now we see that we can use the group multiplication as our function for the algebra, so $a = (\circ)$. Since the group multiplication is just an evaluation of products or words. We know from groups that single letter words that are not inverses will just be mapped to itself, which makes sure we satisfy our triangle diagram. The associativity of a group multiplication also makes sure it satisfies our square diagram, since this means it does not matter in what order we evaluate the sub words. So we see that we can create an algebra over the free group monad from a group.

Now we can also look at whether we can construct a group from one of our algebras (A, a). Again the set of the group will just be the set A of our algebra. Now the group multiplication can be created using our algebra function, so $(\circ) = a$. The associativity is required by our square diagram. If we have three elements $a, b, c \in A$ we see that it does not matter whether we see the product as abc, $(ab)(c)$ or $(a)(bc)$ according to our square diagram, since it does not matter whether we first evaluate the sub words or first concatenate the letters, so $a \cdot Ta = a \cdot \mu_A$. This is exactly what the associativity of a group demands. For the identity element we just take the element the empty word is mapped to, since we can always add this after or before a word without changing the result. Finally the inverse elements can be retrieved by looking at what element in A the inverses are mapped to. So we now know that an algebra over the free group monad is precisely a group.

For morphisms in the Eilenberg-Moore category over the free group monad we see that a morphism from (A, a) to (B, b) is based on a map $f : A \to B$ that has to comply to the square diagram (6.1). This diagram requires that it will be the same to first evaluate the words of A and then map this to B as to first map the words of A to the words of B by mapping the letters and then evaluating these results, so $f \cdot a = b \cdot Tf$. So for two elements u and v in A this means that $f(uv) = f(u)f(v)$. We now see that this is the same as group homomorphisms. So we now know that the objects in the Eilenberg-Moore category for the free group monad are precisely groups and that the morphisms in this category are precisely group homomorphisms. We can therefore see that the Eilenberg-Moore category for the free group monad is the category **Group**. △

6.2 F-algebras

In Haskell we can also speak of algebras. When talking about Haskell the algebras are often called F-algebras. This is mostly because in Haskell we often use F for our functors. Since you can derive the object from the function of your algebra, it is enough to just define a function on a type in Haskell. The definition of an algebra in Haskell uses a type synonym. Here you simply give a different name to an already existing type. The definition of an algebra in Haskell is:

```
type Algebra f a = f a -> a
```

Here we see that in Haskell an algebra is a function from `f a` to `a`. Since the definitions in Haskell often don't really care about the extra requirements, like our diagrams, we can see here that they only really need the functor and not the complete monad. In practice you should however often make sure that the extra requirements are met, which often goes automatically if you make an algebra that makes practical sense.

Example 6.2.1. We have seen that in category theory monoids can be seen as algebras, so we can now check whether this is also true in Haskell. It does however work a bit different in Haskell, since in Haskell we don't use monads but just the endofunctor. So we use the functor with the data type:

```
data MonF a = MEmpty | MAppend a a
```

So this is either empty or the concatenation of two others. This concatenation can be seen as the multiplication. Now for an algebra we have to make sure we implement it for the empty case or the concatenated case. So the implementation will have a definition for `mempty` and `mappend`. So we see that this is precisely the definition of a monoid. For example our string monoid would be defined as an algebra over our `MonF` functor in the following way:

```
str :: Algebra MonF String
str MEmpty        = ""
str (MAppend x y) = x ++ y
```

We see that this is precisely the same as our definition of the string monoid in Example 2.3.2. △

6.3 Adjunctions from monad

In Section 5.3 we have seen that we can create monads from adjunctions. We will now be looking at deriving adjunctions from monads. For this we will be proofing that there is an adjunction between the category the monad is acting on and the corresponding Eilenberg-Moore category. To do this we will be proofing the following Lemma:

Lemma 6.3.1. *Given a monad (T, η, μ) on a category C, there is an adjunction $F \dashv G$ between C and the Eilenberg-Moore category C^T over T.*

Proof. For functor $G\colon C^T \to C$ we can use the forgetful functor, which just forgets the algebraic structure. For the functor $F\colon C \to C^T$ we take the functor that maps an object $A \in C$ to the free T-algebra

$$FA \coloneqq (TA, \mu_A\colon T^2A \to TA)$$

and a morphism $f\colon A \to B$ to the free T-algebra morphism

$$Ff \coloneqq (TA, \mu_A) \xrightarrow{Tf} (TB, \mu_B).$$

We can see that FG is indeed equal to T. Now for the unit of the adjunction we can just use the unit natural transformation $\eta\colon 1_C \Rightarrow T$. For the counit $\epsilon\colon FG \Rightarrow 1_{C^T}$ we will define the components to be:

$$\epsilon_{(A,a)} \coloneqq (TA, \mu_A) \xrightarrow{a} (A, a).$$

So the components of the counit use the algebra structure of that algebra for the morphism. To see that these components are indeed T-algebra morphisms we see that the following square diagram commutes:

$$\begin{array}{ccc} T^2A & \xrightarrow{Ta} & TA \\ {\scriptstyle \mu_A}\downarrow & & \downarrow{\scriptstyle a} \\ TA & \xrightarrow{a} & A \end{array}$$

We now also see that indeed $G\epsilon F = \mu$. So we know that the underlying monad of this adjunction is indeed T. Now we only need to prove that the triangle identities hold for the unit and counit. If we only look at the first element of an algebra these triangle identities are:

$$\begin{array}{ccc} TA & \xrightarrow{T\eta_A} & TTA \\ & {\scriptstyle 1_{TA}}\searrow & \downarrow{\scriptstyle \mu_A} \\ & & TA \end{array} \qquad\qquad \begin{array}{ccc} A & \xrightarrow{\eta_A} & TA \\ & {\scriptstyle 1_A}\searrow & \downarrow{\scriptstyle a} \\ & & TA \end{array}$$

We can see that the left diagram commutes because of the unit diagram from the definition of a monad. The right diagram commutes because of the unit triangle diagram from the definition of an algebra. □

We can now also note that if we would construct the adjunction for the examples in Section 6.1 the functors would be the free and forgetful functors. These are also precisely the functors we used to construct the endofunctor for these algebras.

Since Haskell uses a single category, we can't have functors between different categories. Since these adjunctions from monads use functors between different categories we won't be able to define these adjunctions in Haskell.

7 Kleisli

We have now looked at one way of constructing a category based on a monad. In this chapter we will be looking at another way of constructing a category from a monad called the Kleisli category [2, Section 5.2]. The morphisms in this category use something that is sometimes called a Kleisli arrow. We will be looking at how we can use these Kleisli arrow in Haskell [3, Chapter 20, 22]. After this we will again be looking at constructing an adjunction between a category and the Kleisli category based on a monad [2, Section 5.2].

7.1 Kleisli categories

Let us start with the definition of a Kleisli category:

Definition 7.1.1. Let (T, η, μ) be a monad over category C. The **Kleisli category** C_T is defined such that:

- objects are the objects in C
- a morphism from A to B in C_T, denoted as $A \rightsquigarrow B$, is defined as a morphism $A \to TB$ in C.

Now the identities $A \rightsquigarrow A \in C_T$ are defined using the unit $\eta_A : A \to TA$. The composition of morphisms $f : A \to TB$ from A to B and $g : B \to TC$ from B to C is defined as:

$$A \xrightarrow{f} TB \xrightarrow{Tg} T^2C \xrightarrow{\mu_C} TC$$

Some examples of Kleisli categories over monads we have seen before are:

Example 7.1.2. We can look at what the Kleisli category over the maybe monad will be. We can see that since it is a monad over **Set**, our objects will be sets. A morphism $A \rightsquigarrow B$ will be a function $A \to B_+$. These morphisms can be thought of as partially defined functions where the result of the elements mapped to the extra part is undefined. We also see that the composition is the maximal partially-defined function, so elements that are in one of the morphisms mapped to the extra point will be mapped to the extra point in the result. △

Example 7.1.3. We can now also look at the Kleisli category over the free monoid monad. We can again see that the objects are sets. A morphism $A \rightsquigarrow B$ is a function $A \to \coprod_{n \geq 0} B^n$. So for every $a \in A$ the result will be a list of elements from B. We can see that for the composition we will first map an element to a list. After this we will map all elements in this list to again a list of elements. Finally this list of lists is concatenated to a single list. △

7.2 Kleisli arrows

In Haskell there is no definition for a Kleisli category. We can however use the notion of a Kleisli category to create another definition of a monad. We know that if we would be looking at Haskell as a Kleisli category that the objects would still be sets, but the morphisms would not just be regular functions but functions with signature: `a -> m b`. In Haskell these functions are often called Kleisli arrows. If we would now define the unit of a monad together with the composition of these arrows, we can use this to define a monad. So the new definition would be:

```
class Monad m where
    (>=>)  :: (a -> m b) -> (b -> m c) -> (a -> m c)
    return :: a -> m a
```

Here the `(>=>)` operator, also called the fish operator, defines the composition of Kleisli arrows. Since we are now using a new function, we can also redefine the requirements for a monad as follows:

```
return >=> f    = f
f >=> return    = f
(f >=> g) >=> h = f >=> (g >=> h)
```

Here again the first two lines require that the return function works as a unit for the fish operator and the last line requires associativity for the fish operator.

We can also use our definition of composition in the Kleisli category to go from the definition of a monad using `join` to this definition. This can be done using the following definition for the fish operator:

```
f >=> g = join . fmap g . f
```

We can see that this is precisely our definition of function composition in the Kleisli category. We can also define the fish operator in term of the bind operator as follows:

```
f >=> g = \x -> f x >>= g
```

Now to prove that this definition of a monad is indeed equal to the definitions we saw earlier we will write the bind operator in terms of the fish operator as follows:

```
ma >>= g = (id >=> g) ma
```

Because the fish operator is a function composition we can see that if we have `(id >=> g) (f x)`, this is precisely first applying `f` to `x` and then `id` and `g`. Since `id` is the identity we see that this is the same as applying `f` and then `g`. So we see that this is the same as `f >=> g`.

To now see that this definition using the fish operator is indeed the same as the definitions of monads we have seen before we will first look at the extra requirements defined on our monads. If we use the above definition of the bind operator an our previous requirements we get:

```
(id >=> k) (return a)                = k a
(id >=> return) m                    = m
(id >=> (\x -> (id >=> h) (k x))) m = (id >=> h) ((id >=> k) m)
```

If we now use the fact that we have seen that `(id >=> g) (f x)` is equal to `f >=> g` we get:

```
(return >=> k)       = k
(id >=> return) m    = m
(id >=> (k >=> h)) m = ((id >=> k) >=> h) m
```

We can now see that here `m` can be seen as an object with a Kleisli arrow `f` applied to it, since in the requirements using the bind operator it is already a monad, so it already has a Kleisli arrow applied to it. So if we use this we get:

```
(return >=> k)    = k
(f >=> return)    = f
(f >=> (k >=> h)) = ((f >=> k) >=> h)
```

We can now see that these requirements are the same as we defined earlier. Now we also need to check whether using the cross references twice we indeed get the same monad back. So if we start with a monad defined using the bind operator we get:

```
ma >>=' g = (\x -> id x >>= g) ma
```

We see that if we apply the lambda function of `ma` we get `id ma >>= g`. Because `id` is the identity we get indeed `ma >>= g`. Now for a monad defined using the fish operator we get:

```
f >=>' g = \x -> (id >=> g) (f x)
```

We see that we again have the fish operator with `id`, so we see that this is precisely `f >=> g`, as we want. So we now know that this new definition is equal to our earlier definitions of a monad from Section 5.2.

We can now look at some examples using this definition of a monad:

Example 7.2.1. We have seen that our categorical definition of a Kleisli category the composition is just generally defined. In Haskell this works similarly since we can just use our other declarations of the fish operator to create this monad from other functions. For most monads using these declarations also does not yield any new interesting function. So for the list monad the new instance will just be:

```
instance Monad [] where
    f >=> g  = concat . map g . f
    return x = [x]
```

△

7.3 Adjunctions from monads

In Section 6.3 we have already seen a way to construct an adjunction from a monad. We will now look at another way to construct an adjunction from a monad. This adjunction will be between the category the monad is acting on and the corresponding Kleisli category.

Lemma 7.3.1. *Given a monad (T, η, μ) acting on C, there is an adjunction $F \dashv G$ between C and the Kleisli category C_T over T.*

Proof. For our functor F we can use the identity for the objects. For the morphisms we can have it map a morphism $f\colon A \to B$ in C to the morphism $A \rightsquigarrow B$ defined as follows:

$$Ff \coloneqq A \xrightarrow{f} B \xrightarrow{\eta_B} TB.$$

Now for the functor U we can have it send an object $A \in C_T$ to $TA \in C$. It also maps a morphism $A \rightsquigarrow B$ based on $g\colon A \to TB$ to the morphism defined as follows:

$$Gg \coloneqq TA \xrightarrow{Tg} T^2B \xrightarrow{\mu_B} TB.$$

We can now see that indeed $GF = T$. Because morphisms in the Kleisli category are base on morphisms of the form $A \to TB$ in C we can create a natural isomorphism to create and adjunction in the following way:

$$C_T(FA, B) \cong C(A, TB) \cong C(A, GB).$$

So we can see that we indeed have created an adjunction $F \dashv G$. □

8 Conclusion

We have taken a look at the categorical origin of the functional programming language Haskell. For this we have seen that the complete language can be seen as a category. This gave us insight into looking at some other basic concepts within category theory and the way they can be implemented in Haskell. We have seen that if one of these concepts has useful characteristics in category theory on a single category it is often also useful in Haskell, like functors and monads. If however the categorical concepts are mostly of use between different categories it usually does not really has its use in Haskell since it is based on a single category, like adjunctions. There are also quite some categorical concepts which do have a definition in Haskell, which could be useful, but that are not generally used, like algebras. This is often because there are easier ways to get the same result.

By looking at this categorical origin of monads in Haskell we have gotten a better understanding of where these monad came from and why they are defined the way they are. We have however not taken a very deep dive into the uses of monads in Haskell. We could for example still look at the way they can be used to create side effects, or the way they are often used with a special `do` notation to make it easier to use. Some more uses are described in [3], but to really understand the use of monads it is best to get some practical experience by just using them in creating a program.

There are also a lot more categorical concepts that we could look at. These are things like duality, which gives us a second similar definition for most definitions, limits, the Yoneda lemma and Kan extensions, which can be used to view almost every other concept as a Kan extension. These concepts and more are described in [2] for the categorical definition and in [3] for the way they are used in Haskell.

References

[1] *Haskell documentation*. URL: https://www.haskell.org/documentation/ (visited on May 15, 2021).

[2] Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.

[3] Bartosz Milewski. *Category theory for programmers*. Blurb, 2018.

[4] Saunders Mac Lane. *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media, 2013.