Technische Universität München

Department of Mathematics

Bachelor's Thesis

Monads and their Applications in Haskell

Marvin Jahn

Supervisor/Advisor: Prof. Dr. Claudia Scheimbauer

Submission Date: 15.10.2020

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, 14.10.2020 Marvin Jahn

Zusammenfassung

Diese Arbeit beschäftigt sich mit Monaden und ihren Anwendungen in der Programmiersprache Haskell. Zu Beginn geben wir zwei äquivalente Definitionen für Monaden: Zum einen die konventionelle, zum anderen eine abstraktere Definition als "Monoide in der Kategorie der Endofunktoren". Danach betrachten wir die wohlbekannte Konstruktion, die es erlaubt, Monaden aus Adjunktionen zu erhalten und nutzen diese, um einige Beispiele zu konstruieren. So betrachten wir viele Monaden, die von Adjunktionen zwischen freien Funktoren und Vergissfunktoren induziert werden. Eine weitere Klasse von Beispielen erhalten wir durch das Konzept der *Aktionsmonade*. Wir zeigen außerdem, dass jede Monade von einer Adjunktion abgeleitet werden kann und stellen zwei verschiedene Konstruktionen mit dieser Eigenschaft vor, nämlich die *Eilenberg-Moore* und die *Kleisli Kategorie*.

Nachdem wir eine Einführung in die Programmiersprache Haskell gegeben haben, konstruieren wir die kartesisch abgeschlossene Kategorie $\mathscr{H}\!ask$, welche uns erlaubt, einen Großteil von Haskell mittels Kategorientheorie zu beschreiben. Außerdem führen wir die `Monad` Typklasse ein und machen deutlich, wie diese die Definition einer Monade widerspiegelt. In diesem Zusammenhang definieren wir auch *extension systems* (ein deutscher Begriff existiert nicht) und zeigen, dass dieses Konzept äquivalent zur Definition einer Monaden ist. Um einen Einblick in den praktischen Einsatz von Monaden in Haskell zu erhalten, betrachten wir ein paar Beispiele. Schließlich definieren wir starke und angereicherte Monaden und sehen, dass alle Monaden in Haskell, die durch die `Monad` Typklasse dargestellt werden, stark sind.

Contents

1 Introduction

Category theory is one of the most abstract branches of mathematics, which is why it has sometimes jokingly been described as “abstract nonsense”. Due to the abstraction, many concepts in category theory are difficult to understand and one could complain that it is hard to find applications for this abstract framework. However, this text serves as proof that the reputation of category theory as a field with little use in the “real world” is not justified. Namely, we explain how monads are used in programming, more precisely, in the programming language Haskell.

It is surprising, that something as abstract as monads can be used in something “down to earth” like programming. After all, monads were invented in the late 1950s in a purely mathematical setting. The applications of monads in functional programming languages were only discovered roughly 30 years later.

As mentioned, this text focuses on the programming language Haskell, in order to demonstrate the uses of monads in programming. The main reason for the choice of this programming language is that Haskell is generally regarded as the programming language most reliant on monads. Indeed, monads are not “some” construct in the Haskell programming language; instead they form a fundamental part of the language and serve as a means to conveniently perform certain computations. Their relevance is also underlined by the fact that Haskell provides a dedicated syntax, called *do-notation*, in order to make programming with monads more comfortable.

The main goal of this text is to offer an introduction to monads, the Haskell programming language and most importantly, their interplay.

This is done from a very rigorous perspective and special care is taken to explain the underlying mathematical structure that makes the objects work the way they are expected to. Since monads constitute a formidable obstacle to the aspiring Haskell programmer, countless tutorials on their uses in Haskell can be found. However, the vast majority of those texts is aimed at programmers and consequently, the mathematical structures behind monads are usually suppressed.

This text takes the opposite approach: A clear focus is placed on the mathematical background and less emphasis is placed on Haskell code. The text is aimed at mathematicians and the reader should be familiar with basic notions from category theory. However, knowledge about programming is not required, as a short introduction to the Haskell programming language is given.

In the first section, we highlight two definitions of monads; the conventional one and a more abstract definition as “monoids in the category of endofunctors”.

The second section describes the well-known connection between monads and adjunctions. This allows us to construct monads from familiar adjunctions, resulting in many interesting monads. We also define the famous *Eilenberg-Moore* and *Kleisli* categories, both of which offer an answer to the question: “How can we construct an adjunction that induces a given monad?”. Additionally, we obtain another class of monads; namely the *Action* monad.

The next section starts with a brief introduction to Haskell. In order to use category theory to reason about Haskell, we define a suitable category. This is the category $\mathcal{Hask}$ and special care is taken to explain the potential problems arising from similar constructions. We finish the section by looking at the `Functor` type class in Haskell.

The final section puts the established connections between category theory and Haskell to use. First, *extension systems* are defined and it is shown that they are just different description of the concept of a monad. Then we explain the `Monad` type class and give some concrete examples of monads in Haskell, rediscovering some previously defined monads in the process. We conclude by looking at strong and enriched monads. There we see that all monads represented by the `Monad` type class are strong.
I hope this text will convince the reader that the connection between a very abstract concept like monads, and an applied craft such as programming is both useful and beautiful. On a personal note, Haskell was my original motivation to get acquainted with category theory, so the application of categorical thinking in Haskell has a special place in my heart.

2 Monads as Monoids in the Category of Endofunctors

In this section, we define many important notions that will be needed throughout this text. In particular, we give two definitions of monads.

2.1 Monoidal Categories

While the classical definition of a monad in terms of endofunctors and natural transformations is not too hard, there is an alternative definition, which requires more theoretic setup, but in turn yields monads as a special case. This alternative definition is stated in [Mac98, p.138], where it is noted that

> "A monad in X is just a monoid in the category of endofunctors of X, with product $\times$ replaced by composition of endofunctors and unit set by the identity endofunctor."

To understand this definition, we first need to define monoidal categories.

Definition 2.1. A **monoidal category** $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ consists of

- a category $\mathcal{C}$,
- a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ (called *monoidal product*),
- an object $1 \in \mathcal{C}$ (called *unit object*),
- three natural isomorphisms, for $A, B, C \in \mathcal{C}$ arbitrary

$$\begin{aligned} \lambda(A) : &\quad 1 \otimes A \xrightarrow{\cong} A, \quad \text{(called \textit{left unitor})} \\ \rho(A) : &\quad A \otimes 1 \xrightarrow{\cong} A, \quad \text{(called \textit{right unitor})} \\ \alpha(A,B,C) : &\quad (A \otimes B) \otimes C \xrightarrow{\cong} A \otimes (B \otimes C) \quad \text{(called \textit{associator})}, \end{aligned}$$

such that the following two diagrams commute for all $A, B, C, D \in \mathcal{C}$:

$$\begin{tikzcd}
((A \otimes B) \otimes C) \otimes D \arrow[r, "\alpha"] \arrow[d, "\alpha \otimes \mathrm{id}_D"] & (A \otimes B) \otimes (C \otimes D) \arrow[r, "\alpha"] & A \otimes (B \otimes (C \otimes D)) \\
(A \otimes (B \otimes C)) \otimes D \arrow[rr, "\alpha"] & & A \otimes ((B \otimes C) \otimes D), \arrow[u, "\mathrm{id}_A \otimes \alpha"']
\end{tikzcd}$$

$$\begin{tikzcd}
(A \otimes 1) \otimes B \arrow[rr, "\alpha(A,1,B)"] \arrow[dr, "\rho(A) \otimes \mathrm{id}_B"'] & & A \otimes (1 \otimes B) \arrow[dl, "\mathrm{id}_A \otimes \lambda(B)"] \\
& A \otimes B. &
\end{tikzcd}$$

In the first diagram, α denotes the appropriate associator, for example $\alpha(A \otimes B, C, D)$. A **strict monoidal category** is a monoidal category, such that λ, ρ and α are identities.

The diagrams are inspired by the associativity and identity laws of a monoid. To gain some intuition, we briefly examine how one can obtain monoidal categories from (ordinary) monoids. Indeed, a monoid M directly induces a strict monoidal category by considering the discrete category consisting of the elements of the monoid as objects. In that case, the monoidal product is given by the multiplication of the monoid and the unit object is the identity element of the monoid [BJK05, Exa. 1.2].
Moreover, if the monoid M is commutative, it permits another interpretation as a monoidal category, which is obtained by viewing M as a category with a single object $*$. Then the monoidal product $\otimes : M \times M \to M$ is given by the multiplication $\cdot$ of the monoid. The commutativity of M is necessary and sufficient for $\otimes$ to be a functor: Clearly $\mathrm{id}_* \otimes \mathrm{id}_* = \mathrm{id}_*$, but we also need that for $f, f', g, g' \in \mathrm{End}(M)$,

$$(f' \otimes g') \circ (f \otimes g) = (f' \cdot g') \cdot (f \cdot g) = f' \cdot g' \cdot f \cdot g$$

equals

$$(f' \circ f) \otimes (g' \circ g) = (f' \cdot f) \cdot (g' \cdot g) = f' \cdot f \cdot g' \cdot g.$$

By multiplying with f'^{-1} from the left and g^{-1} from the right, we see that this is equivalent to the commutativity of M.

We give some examples of monoidal categories.

Example 2.2.

1. Any category with finite products is monoidal by defining the monoidal product to be that product and 1 to be a terminal object. Such a category is called *cartesian monoidal* category [EK66, p.551].
   Dually, a category with finite coproducts is monoidal, where the monoidal product is the coproduct and 1 is an initial object. Since products and coproducts are only unique up to isomorphism, it is necessary to explicitly choose one of the candidates for every pair of objects [Mac63, p.30].

2. The usual tensor products from algebra give rise to monoidal categories. For example, fix a field K and consider the category $\mathcal{V}\mathit{ect}_K$, consisting of the K-vector spaces as objects and the K-linear maps as morphisms. This category is monoidal; its monoidal product is the tensor product and the unit object is K [Mac63, p.31].

3. As a generalization of the previous example, consider the category $\mathcal{Mod}_R$ of R-modules with R-linear maps, where R is a commutative ring (with unit). Just as in the previous example, this category is monoidal with the tensor product. The same holds for the category of abelian groups $\mathcal{Ab}$, because it is isomorphic (as a category) to the category of $\mathbb{Z}$-modules $\mathcal{Mod}_{\mathbb{Z}}$: Any abelian group A can be endowed with the structure of a $\mathbb{Z}$-module by defining $z \cdot a$ to be the z-times sum of x, where $x = a$ for $z \geq 0$ and $x = -a$ otherwise; i.e. we define $z \cdot a$ inductively by setting $0 \cdot a := 0$ and $(z \pm 1) \cdot a := z \cdot a \pm a$.
   Moreover, this is the only possible way that A can be given a $\mathbb{Z}$-module structure, since the scalar multiplication is uniquely determined by the action of 1, and multiplying by 1 has to yield the identity function on A by the module axioms.
   Since additionally any homomorphism $f : A \to B$ between abelian groups is automatically $\mathbb{Z}$-linear, this defines a functor $\mathcal{Ab} \to \mathcal{Mod}_{\mathbb{Z}}$, which is inverse to the forgetful functor $U : \mathcal{Mod}_{\mathbb{Z}} \to \mathcal{Ab}$, so $\mathcal{Ab}$ is isomorphic to $\mathcal{Mod}_{\mathbb{Z}}$.

2.2 Monoids in Monoidal Categories

As a next step, we can define a monoid in the categorical sense, which, as the name suggests, is inspired by the notion of a usual monoid.

Definition 2.3. A **monoid** (C, μ, η) in a monoidal category $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ is an object $C \in \mathcal{C}$ together with two morphisms $\mu : C \otimes C \to C$ (called *multiplication*) and $\eta : 1 \to C$ (called *unit*), such that the following two diagrams commute:

$$\begin{tikzcd}
C \otimes (C \otimes C) \arrow[d, "\mathrm{id}_C \otimes \mu"] & (C \otimes C) \otimes C \arrow[l, "\alpha(C,C,C)"'] \arrow[r, "\mu \otimes \mathrm{id}_C"] & C \otimes C \arrow[d, "\mu"] \\
C \otimes C \arrow[rr, "\mu"] & & C,
\end{tikzcd}$$

$$\begin{tikzcd}
1 \otimes C \arrow[r, "\eta \otimes \mathrm{id}_C"] \arrow[dr, "\lambda"'] & C \otimes C \arrow[d, "\mu"] & C \otimes 1 \arrow[l, "\mathrm{id}_C \otimes \eta"'] \arrow[dl, "\rho"] \\
& C. &
\end{tikzcd}$$

The first diagram ensures “associativity” of μ, while the second one establishes η as a two-sided “neutral element”.

This abstract concept manifests itself in many special cases, some of which will be highlighted in the following.

Example 2.4. 1. Since the category $\mathcal{Set}$, consisting of the sets as objects and the usual maps as morphisms, admits finite products, it is monoidal by 2.2.1. In that case, a monoid (as in 2.3) precisely amounts to an ordinary monoid: μ corresponds to the multiplication and the unique element in the image of η corresponds to the neutral element of the monoid [Bra14, Exa. 4.1.3].

2. An especially interesting example arises when considering $\mathcal{Vect}_K$ as a monoidal category with the tensor product. Then the monoids are precisely the (unital) associative K-algebras: Given a monoid (V, μ, η), there is the K-linear map $\mu : V \otimes V \to V$, which can be composed with the canonical map $V \times V \to V \otimes V$, yielding a bilinear map $\phi : V \times V \to V, (a, b) \mapsto \mu(a \otimes b)$. For $a, b, c \in V$, if we denote $\phi(a, b)$ by $a \cdot b$, then the first diagram shows that

$$\begin{aligned}(a \cdot b) \cdot c &= \mu(\mu(a \otimes b) \otimes c)\\ &= (\mu \circ (\mu \otimes \mathrm{id}_V))((a \otimes b) \otimes c)\\ &= (\mu \circ (\mathrm{id}_V \otimes \mu) \circ \alpha(V, V, V))((a \otimes b) \otimes c)\\ &= (\mu \circ (\mathrm{id}_V \otimes \mu))(a \otimes (b \otimes c))\\ &= a \cdot (b \cdot c),\end{aligned}$$

so $\cdot$ is associative. Moreover, the unit is given by $\eta(1) \in V$ (here 1 denotes the unit element $1 \in K$ and not the unit object in $\mathcal{C}$). This follows from the second diagram, since

$$\eta(1) \cdot a = \mu(\eta(1) \otimes a) = (\mu \circ (\eta \otimes \mathrm{id}_V))(1 \otimes a) = \lambda(1 \otimes a) = a$$

by the commuting left triangle and analogously and $a \cdot \eta(1) = a$ by the right one. Together with the bilinearity of $\cdot$ and $\eta : K \to V$, this shows that $\cdot$ gives V a K-algebra structure, with unit $\eta(1)$.
On the other hand, let A be a K-algebra with underlying vector space V. We denote the canonical map $K \to V$ by η. By the universal property of the tensor product, the multiplication $\cdot : V \times V \to V$ of A gives rise to a K-vector space homomorphism $\mu : V \otimes V \to V$ with $\mu(a \otimes b) = a \cdot b$ for all $a, b \in V$. With this choice of μ and η, the two diagrams commute by a similar calculation as above, so V is indeed a monoid in $\mathcal{Vect}_K$.

3. More generally, let R be a commutative ring. Then the monoids in the monoidal category of R-modules $\mathcal{Mod}_R$, equipped with the tensor product, are the associative R-algebras [BJT97, Exa. 1.2]. By 2.2.3, the monoids in $\mathcal{Ab}$ are the associative $\mathbb{Z}$-algebras. However, since $\mathbb{Z}$ is initial in the category of commutative rings $\mathcal{CRing}$ with the ring homomorphisms as morphisms, an associative $\mathbb{Z}$-algebras is just a ring. Therefore, the monoids in $\mathcal{Ab}$ are precisely the rings.

4. The category $\mathcal{Cat}$, consisting of the small categories as objects and functors as morphisms, is monoidal with its product $\times$; the unit object 1 is the category consisting of a single object $*$ with a single morphism. The monoids in $\mathcal{Cat}$ are precisely the strict monoidal small categories [Awo10, p.79].
To show this, let $(\mathcal{C}, \mu, \eta)$ be a monoid. We denote the bifunctor $\mu : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ by $\otimes'$ and set $1' := \eta(*)$. Then it is not hard to see that $(\mathcal{C}, \otimes', 1')$ is a strict monoidal category. For instance, by the first diagram in the definition of a monoid, it holds

for $A, B, C \in \mathcal{C}$:

$$\begin{aligned}(A \otimes' B) \otimes' C &= \mu(\mu(A, B), C)\\ &= (\mu \circ (\mu \times \mathrm{id}_C))((A, B), C)\\ &= (\mu \circ (\mathrm{id}_{\mathcal{C}} \times \mu) \circ \alpha(\mathcal{C}, \mathcal{C}, \mathcal{C}))((A, B), C)\\ &= (\mu \circ (\mathrm{id}_{\mathcal{C}} \times \mu))(A, (B, C))\\ &= (A \otimes' B) \otimes' C.\end{aligned}$$

Similar calculations show that every strict monoidal small category is a monoid in $\mathcal{C}at$.

We establish the following notation, as in [Bra17, 3.5.10].
Let $F, G : \mathcal{C} \to \mathcal{D}$ be two functors and let $\alpha : F \Rightarrow G$ be a natural transformation. Given a functor $H : \mathcal{D} \to \mathcal{E}$, we write

$$H \bullet \alpha : H \circ F \Rightarrow H \circ G,$$

for the natural transformation with components $(H \bullet \alpha)(A) := H(\alpha(A))$;
in diagrams:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=60, "F", ""{name=U, below}] \arrow[r, bend right=60, "G"', ""{name=D}] & \mathcal{D} \arrow[r, "H"] & \mathcal{E} \arrow[r, phantom, "\rightsquigarrow"] & \mathcal{C} \arrow[r, bend left=60, "H\circ F", ""{name=U2, below}] \arrow[r, bend right=60, "H\circ G"', ""{name=D2}] & \mathcal{E}.
\arrow[Rightarrow, from=U, to=D, "\alpha"]
\arrow[Rightarrow, from=U2, to=D2, "H\bullet\alpha"]
\end{tikzcd}$$

Analogously, given a functor $K : \mathcal{E} \to \mathcal{C}$, the natural transformation

$$\alpha \bullet K : F \circ K \to G \circ K,$$

has components $(\alpha \bullet K)(A) := \alpha(K(A))$ and can be visualized by

$$\begin{tikzcd}
\mathcal{E} \arrow[r, "K"] & \mathcal{C} \arrow[r, bend left=60, "F", ""{name=U, below}] \arrow[r, bend right=60, "G"', ""{name=D}] & \mathcal{D} \arrow[r, phantom, "\rightsquigarrow"] & \mathcal{E} \arrow[r, bend left=60, "F\circ K", ""{name=U2, below}] \arrow[r, bend right=60, "G\circ K"', ""{name=D2}] & \mathcal{D}.
\arrow[Rightarrow, from=U, to=D, "\alpha"]
\arrow[Rightarrow, from=U2, to=D2, "\alpha\bullet K"]
\end{tikzcd}$$

2.3 Definition of Monads and First Examples

The special case we are interested in arises when fixing a category $\mathcal{C}$ and considering the category of endofunctors $\mathcal{C}^{\mathcal{C}}$ of $\mathcal{C}$. To make this category monoidal, we need a bifunctor $\mathcal{C}^{\mathcal{C}} \times \mathcal{C}^{\mathcal{C}} \to \mathcal{C}^{\mathcal{C}}$, which we obtain from the following lemma.

Lemma 2.5 ([Bra17, 3.5.13])**.** For categories $\mathcal{C}, \mathcal{D}, \mathcal{E}$, there exists a functor

$$\circ : \mathcal{E}^{\mathcal{D}} \times \mathcal{D}^{\mathcal{C}} \to \mathcal{E}^{\mathcal{C}},$$

which acts on objects as functor composition and on natural transformations as horizontal composition; i.e. a pair of natural transformations $(\beta : F' \Rightarrow G', \alpha : F \Rightarrow G)$ gets mapped to $\beta \bullet \alpha : (\beta \bullet G) \circ (F' \bullet \alpha) : F' \circ F \Rightarrow G' \circ G$ with components $\beta(G(A)) \circ F'(\alpha(A))$ for $A \in \mathcal{C}$. In pictures:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "F"{name=F}] \arrow[r, bend right=50, "G"'{name=G}] & \mathcal{D} \arrow[r, bend left=50, "F'"{name=Fp}] \arrow[r, bend right=50, "G'"'{name=Gp}] & \mathcal{E} & \rightsquigarrow & \mathcal{C} \arrow[r, bend left=50, "F'\circ F"{name=FF}] \arrow[r, bend right=50, "G'\circ G"'{name=GG}] & \mathcal{E}.
\arrow[Rightarrow, from=F, to=G, "\alpha"]
\arrow[Rightarrow, from=Fp, to=Gp, "\beta"]
\arrow[Rightarrow, from=FF, to=GG, "\beta\bullet\alpha"]
\end{tikzcd}$$

Proof. A pair of identity natural transformations $(\mathrm{id}_{F'} : F' \Rightarrow F', \mathrm{id}_F : F \Rightarrow F)$ is mapped to the natural transformation $F' \circ F \Rightarrow F' \circ F$ with components

$$\mathrm{id}_{F'}(F(A)) \circ F'(\mathrm{id}_F(A)) = \mathrm{id}_{F'(F(A))} \circ F'(\mathrm{id}_{F(A)}) = \mathrm{id}_{(F'\circ F)(A)} \circ \mathrm{id}_{(F'\circ F)(A)} = \mathrm{id}_{(F'\circ F)(A)},$$

revealing that this is indeed the identity natural transformation $\mathrm{id}_{F'\circ F}$.
It is left to show that $\circ$ is compatible with composition. To prove this, let $F, G, H : \mathcal{C} \to \mathcal{D}$, $F', G', H' : \mathcal{D} \to \mathcal{E}$ be functors and $F \overset{\alpha}{\Rightarrow} G \overset{\beta}{\Rightarrow} H$, $F' \overset{\alpha'}{\Rightarrow} G' \overset{\beta'}{\Rightarrow} H'$ natural transformations; i.e.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=70, "F"{name=F}] \arrow[r, "G"{name=G}] \arrow[r, bend right=70, "H"'{name=H}] & \mathcal{D} \arrow[r, bend left=70, "F'"{name=Fp}] \arrow[r, "G'"{name=Gp}] \arrow[r, bend right=70, "H'"'{name=Hp}] & \mathcal{E}.
\arrow[Rightarrow, from=F, to=G, "\alpha"]
\arrow[Rightarrow, from=G, to=H, "\beta"]
\arrow[Rightarrow, from=Fp, to=Gp, "\alpha'"]
\arrow[Rightarrow, from=Gp, to=Hp, "\beta'"]
\end{tikzcd}$$

The middle triangle of the diagram

$$\begin{tikzcd}
F' \circ F \arrow[r, Rightarrow, "F'\bullet\alpha"] \arrow[rrd, Rightarrow, "F'\bullet(\beta\circ\alpha)"'] & F' \circ G \arrow[r, Rightarrow, "\alpha'\bullet G"] \arrow[rd, Rightarrow, "F'\bullet\beta"] & G' \circ G \arrow[r, Rightarrow, "G'\bullet\beta"] & G' \circ H \arrow[r, Rightarrow, "\beta'\bullet H"] & H' \circ H \\
 & & F' \circ H \arrow[ru, Rightarrow, "\alpha'\bullet H"] \arrow[rru, Rightarrow, "(\beta'\circ\alpha')\bullet H"'] & &
\end{tikzcd}$$

commutes by naturality of α', the left one since F' is a functor and the right one by definition. Therefore, the whole diagram commutes and $\circ$ is indeed a functor. □

By applying the lemma to $\mathcal{C} = \mathcal{D} = \mathcal{E}$, we derive a bifunctor $\circ : \mathcal{C}^{\mathcal{C}} \times \mathcal{C}^{\mathcal{C}} \to \mathcal{C}^{\mathcal{C}}$, which turns $\mathcal{C}^{\mathcal{C}}$ into a strict monoidal category with the identity functor $\mathrm{id}_{\mathcal{C}}$ as the unit object. We can now draw the connection to monads.

Definition 2.6. A monad is a monoid in the monoidal category of endofunctors $(\mathcal{C}^{\mathcal{C}}, \circ, \mathrm{id}_{\mathcal{C}})$ [RJ14, p.10].

Such a monoid consists of an endofunctor $T : \mathcal{C} \to \mathcal{C}$ and two natural transformations $\mu : T^2 \Rightarrow T$, $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$, such that the diagrams

$$\begin{tikzcd}
T \circ T^2 \arrow[d, Rightarrow, "\mathrm{id}_T\bullet\mu"] & T^2 \circ T \arrow[l, Rightarrow, "\mathrm{id}_{T^3}"'] \arrow[r, Rightarrow, "\mu\bullet\mathrm{id}_T"] & T^2 \arrow[d, Rightarrow, "\mu"] \\
T^2 \arrow[rr, Rightarrow, "\mu"] & & T,
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{id}_{\mathcal{C}} \circ T \arrow[r, Rightarrow, "\eta\bullet\mathrm{id}_T"] \arrow[rd, Rightarrow, "\mathrm{id}_T"'] & T^2 \arrow[d, Rightarrow, "\mu"] & T \circ \mathrm{id}_{\mathcal{C}} \arrow[l, Rightarrow, "\mathrm{id}_T\bullet\eta"'] \arrow[ld, Rightarrow, "\mathrm{id}_T"] \\
 & T. &
\end{tikzcd}$$

commute. For endofunctors $S, S' : \mathcal{C} \to \mathcal{C}$ and a natural transformation $\alpha : S \Rightarrow S'$, the natural transformation $\alpha \bullet \mathrm{id}_T : S \circ T \Rightarrow S' \circ T$ has components

$$\alpha(T(A)) \circ S(\mathrm{id}_T(A)) = \alpha(T(A)) \circ S(\mathrm{id}_{T(A)}) = \alpha(T(A)) \circ \mathrm{id}_{(S \circ T)(A)} = \alpha(T(A))$$

for $A \in \mathcal{C}$, so $\alpha \bullet \mathrm{id}_T = \alpha \bullet T$. Similarly, we see $\mathrm{id}_T \bullet \alpha = T \bullet \alpha$. Therefore, the previous diagrams simplify to the diagrams in the following definition, which is the classical definition of a monad mentioned before.

Definition 2.7. A **monad** on a category $\mathcal{C}$ consists of

- an endofunctor $T : \mathcal{C} \to \mathcal{C}$,
- a natural transformation $\mu : T^2 \Rightarrow T$ (called *multiplication*),
- a natural transformation $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$ (called *unit*),

such that the following two diagrams commute:

$$\begin{tikzcd}
T^3 \arrow[r, Rightarrow, "T\bullet\mu"] \arrow[d, Rightarrow, "\mu\bullet T"] & T^2 \arrow[d, Rightarrow, "\mu"] \\
T^2 \arrow[r, Rightarrow, "\mu"] & T,
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, Rightarrow, "\eta\bullet T"] \arrow[dr, Rightarrow, "\mathrm{id}_T"'] & T^2 \arrow[d, Rightarrow, "\mu"] & T \arrow[l, Rightarrow, "T\bullet\eta"'] \arrow[dl, Rightarrow, "\mathrm{id}_T"] \\
& T. &
\end{tikzcd}$$

We finish this section by giving some examples, many more will be presented in the next section.

Example 2.8. 1. Any category $\mathcal{C}$ admits the trivial monad, given by the identity functor $\mathrm{id}_{\mathcal{C}}$ and the identity natural transformations.

2. A monoid $(M, \cdot, 1)$ induces a monad on the category of sets $\mathcal{Set}$ by the following construction: The endofunctor is $M \times - : \mathcal{Set} \to \mathcal{Set}$, the multiplication is defined to be

$$\mu(A) : (M \times (M \times A)) \to M \times A,\ (m, (m', a)) \mapsto (m \cdot m', a),$$

and the unit is given by

$$\eta(A) : A \to M \times A,\ a \mapsto (1, a)$$

[CJ11, Lemma 4].

3. The covariant power set functor $\mathcal{P} : \mathcal{Set} \to \mathcal{Set}$ constitutes the *power set* monad, where the components of the unit are given by $\eta(A) : A \to \mathcal{P}(A), a \mapsto \{a\}$. The components of the multiplication $\mu(A) : \mathcal{P}^2(A) \to \mathcal{P}(A)$ map a set of subsets of A to its union [AHS04, Exa. 20.2].
To show that $(\mathcal{P}, \mu, \eta)$ is a monad, we have to check that the following two diagrams commute:

$$\begin{tikzcd}
\mathcal{P}^3 \arrow[r, Rightarrow, "\mathcal{P}\bullet\mu"] \arrow[d, Rightarrow, "\mu\bullet\mathcal{P}"] & \mathcal{P}^2 \arrow[d, Rightarrow, "\mu"] \\
\mathcal{P}^2 \arrow[r, Rightarrow, "\mu"] & \mathcal{P},
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{P} \arrow[r, Rightarrow, "\eta\bullet\mathcal{P}"] \arrow[dr, Rightarrow, "\mathrm{id}_{\mathcal{P}}"'] & \mathcal{P}^2 \arrow[d, Rightarrow, "\mu"] & \mathcal{P} \arrow[l, Rightarrow, "\mathcal{P}\bullet\eta"'] \arrow[dl, Rightarrow, "\mathrm{id}_{\mathcal{P}}"] \\
& \mathcal{P}. &
\end{tikzcd}$$

Let A be a set and $C \in \mathcal{P}^3(A)$ a subset of $\mathcal{P}^2(A)$. It holds

$$\begin{aligned}\big(\mu(A) \circ \mathcal{P}(\mu(A))\big)(C) &= \mu(A)\big(\{\mu(A)(D) : D \in C\}\big)\\ &= \mu(A)\Bigg(\Bigg\{\bigcup_{E \in D} E : D \in C\Bigg\}\Bigg)\\ &= \bigcup_{D \in C} \bigcup_{E \in D} E\end{aligned}$$

and

$$\begin{aligned}\big(\mu(A) \circ \mu(\mathcal{P}(A))\big)(C) &= \mu(A)\Bigg(\bigcup_{D \in C} D\Bigg)\\ &= \bigcup_{E \in \bigcup_{D \in C} D} E\\ &= \bigcup_{D \in C} \bigcup_{E \in D} E,\end{aligned}$$

so the first diagram commutes. The left triangle of the second diagram commutes, because for $B \subset A$, we calculate

$$\big(\mu(A) \circ \eta(\mathcal{P}(A))\big)(B) = \mu(A)(\{B\}) = B = \mathrm{id}_{\mathcal{P}}(A)(B)$$

and the commutativity of the right triangle follows analogously.

3 Constructions with Monads

In this section, we take a deeper look at monads. First we highlight the well known construction used to generate monads from adjunctions. Exploiting the results, we derive various interesting examples, some of which will be encountered later. We also define the *action monad*, which corresponds to the `Writer` monad in Haskell. We then further investigate the connection between monads and adjunctions, asking if every monad arises from some suitable adjunction.

3.1 Monads from Adjunctions

The following construction, which gives us an easy way to derive monads from adjunctions, is due to Huber, see [Hub61, 4.2].

Theorem 3.1 ([Rie17, 5.1.3])**.** An adjunction

$$\begin{tikzcd}\mathcal{C} \arrow[r, bend left, "F"] & \mathcal{D} \arrow[l, bend left, "U"]\end{tikzcd}$$

with unit $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow U \circ F$ and counit $\epsilon : F \circ U \Rightarrow \mathrm{id}_{\mathcal{D}}$ gives rise to a monad on the category $\mathcal{C}$, where

- the endofunctor T is defined to be $U \circ F$,
- the unit $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow U \circ F$ of the adjunction serves as the unit $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$ of the monad,
- the whiskered counit $U \bullet \epsilon \bullet F : (U \circ F)^2 \Rightarrow U \circ F$ is defined to be the multiplication $\mu : T^2 \Rightarrow T$.

Proof. We have to verify that the diagrams

$$\begin{tikzcd}
(U \circ F)^3 \arrow[r, Rightarrow, "(U\circ F)\bullet\mu"] \arrow[d, Rightarrow, "\mu\bullet(U\circ F)"] & (U \circ F)^2 \arrow[d, Rightarrow, "\mu"] \\
(U \circ F)^2 \arrow[r, Rightarrow, "\mu"] & U \circ F,
\end{tikzcd}
\qquad
\begin{tikzcd}
U \circ F \arrow[r, Rightarrow, "\eta\bullet(U\circ F)"] \arrow[dr, Rightarrow, "\mathrm{id}_{(U\circ F)}"'] & (U \circ F)^2 \arrow[d, Rightarrow, "\mu"] & U \circ F \arrow[l, Rightarrow, "(U\circ F)\bullet\eta"'] \arrow[dl, Rightarrow, "\mathrm{id}_{(U\circ F)}"] \\
& U \circ F &
\end{tikzcd}$$

commute. By the naturality of $U \bullet \epsilon : U \circ F \circ U \to U$, the diagram

$$\begin{tikzcd}
(U \circ F \circ U)(B) \arrow[r, "(U\bullet\epsilon)(B)"] \arrow[d, "(U\circ F\circ U)(f)"] & U(B) \arrow[d, "U(f)"] \\
(U \circ F \circ U)(C) \arrow[r, "(U\bullet\epsilon)(C)"] & U(C)
\end{tikzcd}$$

commutes for objects B, C in $\mathcal{C}$ and $f : B \to C$ a morphism. In particular, taking $B := (F \circ U \circ F)(A)$, $C := F(A)$ and $f := (\epsilon \bullet F)(A) : (F \circ UF)(A) \to F(A)$, this implies

$$U((\epsilon \bullet F)(A)) \circ (U \bullet \epsilon)((F \circ U \circ F)(A)) = (U \bullet \epsilon)(F(A)) \circ (U \circ F \circ U)((\epsilon \bullet F)(A)).$$

For A an object in $\mathcal{C}$, the previous equation shows that

$$\begin{aligned}
\mu(A) \circ ((U \circ F) \bullet \mu)(A) &= (U \bullet \epsilon)(F(A)) \circ (U \circ F \circ U)((\epsilon \bullet F)(A)) \\
&= U((\epsilon \bullet F)(A)) \circ (U \bullet \epsilon)((F \circ U \circ F)(A)) \\
&= \mu(A) \circ \mu((U \circ F)(A)),
\end{aligned}$$

so the first diagram commutes. The commutativity of the second diagram follows directly from the triangle identities of the adjunction. □

This is very useful, as it directly connects adjunctions and monads; and as Mac Lane noted in [Mac98, p.xii],

> “Adjoint functors arise everywhere.”

so 3.1 justifies the (informal) statement

> “Monads arise everywhere.”

We emphasize this by giving a plethora of examples, focusing on monads which arise from free-forgetful adjunctions, which constitute a very important kind of adjunction. Throughout this text, we will encounter many more examples of monads.

Example 3.2. 1. Let $\mathcal{Set}_*$ denote the category of pointed sets; i.e. the objects are pairs (X, x), where X is a set and $x \in X$ is an element, and a morphism $f : (X, x) \to (Y, y)$ is a function $f : X \to Y$ with $f(x) = y$.
There is a free-forgetful adjunction

$$\begin{tikzcd} \mathcal{Set} \arrow[r, bend left, "F"] & \mathcal{Set}_*, \arrow[l, bend left, "U"] \end{tikzcd}$$

given by the forgetful functor $U : \mathcal{Set}_* \to \mathcal{Set}$ with left adjoint $F : \mathcal{Set} \to \mathcal{Set}_*$, which maps a set X to the pointed set $X_+ := X \cup \{\{X\}\}$ with basepoint $\{X\}$ and extends a map $f : X \to Y$ to $X_+ \to Y_+$ via $\{X\} \mapsto \{Y\}$.
By 3.1, this induces a monad on $\mathcal{Set}$. The corresponding endofunctor $(-)_+ : \mathcal{Set} \to \mathcal{Set}$ adds a new disjoint point and extends a map $f : X \to Y$ to $f_+ : X \cup \{X\} \to Y \cup \{Y\}$ via $\{X\} \mapsto \{Y\}$. The components of the unit are given by the obvious natural inclusions $\eta(X) : X \to X_+$. The components of the multiplication $\mu(X) : (X_+)_+ \to X_+$ are the identity on the subset X and map the two new points in $(X_+)_+$ to the new point in X_+ [Rie17, Exa. 5.1.4].
We will rediscover this monad in Haskell; there it is called the `Maybe` monad.

2. The free group $\langle X \rangle$ of a given set X can be defined as follows: Its elements are all finite words, constructed from the elements of X and their formal inverses, modulo some obvious relations. The group operation is concatenation of words and the neutral element is the empty word [Kna06, pp.307].
Let $\mathcal{Grp}$ denote the category of groups with group homomorphisms as morphisms. The forgetful functor $U : \mathcal{Grp} \to \mathcal{Set}$ has a left adjoint $F : \mathcal{Set} \to \mathcal{Grp}$, which maps a set X to the free group $\langle X \rangle$ and extends a function $f : X \to Y$ to a group homomorphism $F(f) : \langle X \rangle \to \langle Y \rangle$ [Law06, p.9].. This extension is unique, since by definition X generates the free group $\langle X \rangle$. The endofunctor $T : \mathcal{Set} \to \mathcal{Set}$ of the induced monad, called the *free group monad*, maps a set X to the free group $\langle X \rangle$ and extends a function $X \to Y$ to $\langle X \rangle \to \langle Y \rangle$.
Now the components of the unit are the natural inclusions $\eta(X) : X \to \langle X \rangle$ and the components of the multiplication are the maps $\mu(X) : \langle \langle X \rangle \rangle \to \langle X \rangle$, given by concatenation.

3. The category $\mathcal{Mon}$ of monoids consists of the monoids as objects and the monoid homomorphisms as morphisms.
Since we did not explicitly use the group structure in the previous example, a completely analogous construction can be performed for the forgetful functor $U : \mathcal{Mon} \to \mathcal{Set}$ and the free monoid, resulting in the *free monoid monad.* This monad will reappear in Haskell as the *list monad.*

4. Let R be a ring. The forgetful functor $U : \mathcal{Mod}_R \to \mathcal{Set}$ admits a left adjoint F, which maps a set X to the free R-module $\bigoplus_X R$ and extends morphisms similar to the previous examples.
This adjunction induces the *free R-module monad*; the corresponding endofunctor

$T : \mathcal{Set} \to \mathcal{Set}$ maps a set X to its free R-module $\bigoplus_X R$ and extends a map $X \to Y$ to $\bigoplus_X R \to \bigoplus_Y R$ by linearity.
The components of the unit $\eta(X) : X \to \bigoplus_X R$ are again the obvious inclusions. The components of the multiplication $\mu(X) : \bigoplus_X(\bigoplus_X R) \to \bigoplus_X R$ are defined by distributing the coefficients in the formal sum of formal sums.
Interesting special cases are the *free abelian group monad* and the *free vector space monad.*

5. For a ring R and a group G, the *group ring* (or *group algebra*) $R[G]$ consists of the set of maps $\alpha : G \to R$ with finite support ($\alpha(g) \neq 0$ for only finitely many $g \in G$), together with pointwise addition and multiplication via

$$(\alpha \cdot \beta)(g) = \sum_{x \cdot y = g, x, y \in G} \alpha(x) \cdot \beta(y),$$

which makes it into a ring. If both R and G are commutative, then so is the group ring $R[G]$. Just as in the previous example, one can alternatively think of the elements of $R[G]$ as formal sums indexed by G, with coefficients in R, which we will do in the following.
The construction works completely analogous for a monoid M, resulting in the *monoid ring* (or *monoid algebra*) $R[M]$. The neutral element of addition is 0 and the neutral element of multiplication is $1 \cdot e$, where e is the neutral element of the monoid. There exists a canonical monoid homomorphism

$$\eta(M) : M \to (R[M], \cdot),\ m \mapsto 1 \cdot m$$

and a ring homomorphism

$$\phi : R \to R[M],\ r \mapsto r \cdot e$$

[Lan02, pp.104]. For example, the polynomial ring $R[x_1, \dots, x_n]$ is just the monoid ring $R[M]$, where M is the free commutative monoid generated by the "symbols" $x_1, \dots, x_n$ and ϕ is just the inclusion $R \subset R[x_1, \dots, x_n]$.
Choosing $R = \mathbb{Z}$, a monoid homomorphism $f : M \to N$ gives rise to a ring homomorphism

$$\mathbb{Z}[f] : \mathbb{Z}[M] \to \mathbb{Z}[N],\ \sum_{m \in M} z_m \cdot m \mapsto \sum_{n \in N} \left(\sum_{f(m)=n} z_m \right) \cdot n. \qquad (*)$$

Denoting the category of (not necessarily commutative) rings with ring homomorphisms as $\mathcal{Ring}$, this defines a functor $\mathbb{Z}[-] : \mathcal{Mon} \to \mathcal{Ring}$, which is left adjoint to the forgetful functor $U : \mathcal{Ring} \to \mathcal{Mon}$, given by "forgetting" the addition of a ring [Mac65, p.61]. The unit $\eta : \mathrm{id}_{\mathcal{Mon}} \Rightarrow U \circ \mathbb{Z}[-]$ turns out to be the above defined family of morphisms. The components of the counit $\epsilon : \mathbb{Z}[-] \circ U \Rightarrow \mathrm{id}_{\mathcal{Ring}}$ are the maps $\epsilon(S) : \mathbb{Z}[(S, \cdot)] \to S, \sum_{s \in S} z_s \cdot s \mapsto \sum_{s \in S} z_s \cdot s$, where the right hand side means evaluating the formal sum in S.
The endofunctor of the induced monad maps a monoid M to its group ring $\mathbb{Z}[M]$,

viewed as a monoid via multiplication. A morphism is transformed as described at $(*)$, but is regarded as a monoid homomorphism. The unit η of the monad is just the unit of the adjunction and the components of the multiplication $\mu(M) : \mathbb{Z}[(\mathbb{Z}[M], \cdot)] \to \mathbb{Z}[M]$ are defined by distributing the coefficients.

6. Let M be a monoid and denote the category of M-actions with equivariant maps by ${}_M\mathcal{Set}$. This means that the objects are pairs (X, α), where X is a set and $\alpha : M \to (\mathrm{End}(X), \circ, \mathrm{id}_X)$ is a homomorphism of monoids. A morphism $f : (X, \alpha) \to (Y, \beta)$ is a map $f : X \to Y$, such that $\beta(m) \circ f = f \circ \alpha(m)$ for all $m \in M$ [Bra17, Def. 2.6.21]. Notice that ${}_M\mathcal{Set}$ is isomorphic to the functor category $\mathcal{Set}^M$, where M is viewed as a category with a single object.
   The forgetful functor $U : \mathcal{Set} \to {}_M\mathcal{Set}$ admits a left adjoint $M \times -$, mapping a set X to $(M \times X, m \mapsto ((m', x) \mapsto (m \cdot m', x)))$ [EM01, Lemma 2.1]. The induced monad turns out to be the monad from 2.8.2.

7. The following example is due to Huber in [Hub61, p.371]. Let R be a ring. For more concise and clearer notation, we write $\langle X \rangle$ for the free R-module $\bigoplus_X R$ generated by X. There is an adjunction between the category of pointed sets $\mathcal{Set}_*$ and the category of R-modules $\mathcal{Mod}_R$, given by the following functors:
   The functor $F : \mathcal{Set}_* \to \mathcal{Mod}_R$ maps a pair (X, x) to the free R-module with the only relation that x is the base element, namely $\langle X \rangle / \langle x \rangle$. A morphism $f : (X, x) \to (Y, y)$ gives rise to an R-module homomorphism $f : \langle X \rangle \to \langle Y \rangle$. By composing f with the canonical map $\langle Y \rangle \twoheadrightarrow \langle Y \rangle / \langle y \rangle$, we derive an R-module homomorphism $f' : \langle X \rangle \to \langle Y \rangle / \langle y \rangle$. Since $f(x) = y$, $\langle x \rangle$ is a subset of the kernel $\ker(f')$, so the isomorphism theorem yields an R-module homomorphism $\langle X \rangle / \langle x \rangle \to \langle Y \rangle / \langle y \rangle$. This defines the action of F on morphisms.
   The functor $U : \mathcal{Mod}_R \to \mathcal{Set}_*$ is forgetful in nature; it maps an R-module to its underlying set with the zero element as basepoint and since a module homomorphism preserves the zero element, its action on the morphisms of $\mathcal{Mod}_R$ is clear.
   The components of the unit $\eta : \mathrm{id}_{\mathcal{Set}_*} \Rightarrow U \circ F$ of the adjunction are given by the inclusion $\eta((X, x)) : (X, x) \to (\langle X \rangle / \langle x \rangle, \langle x \rangle)$. The counit $\epsilon : F \circ U \Rightarrow \mathrm{id}_{\mathcal{Mod}_R}$ has components $\epsilon(M) : \langle M \rangle / \langle 0 \rangle \to M$, which act on the generators (the elements of M) as the inclusion and extend this map by linearity, using that $\langle 0 \rangle$ is in the kernel of the resulting homomorphism.
   The endofunctor $T : \mathcal{Set}_* \to \mathcal{Set}_*$ of the corresponding monad maps a pointed set (X, x) to the pointed set $(\langle X \rangle / \langle x \rangle, \langle x \rangle)$ and extends a map $f : (X, x) \to (Y, y)$ to $(\langle X \rangle / \langle x \rangle, \langle x \rangle) \to (\langle Y \rangle / \langle y \rangle, \langle y \rangle)$. The unit of the monad is the same as the unit of the adjunction, which was already described. The components $\mu((X, x))$ of the multiplication are morphisms

$$(\langle(\langle X \rangle / \langle x \rangle)\rangle / \langle\langle x \rangle\rangle, \langle\langle x \rangle\rangle) \to (\langle X \rangle / \langle x \rangle, \langle x \rangle),$$

   which act on the generators, namely the elements of $\langle X \rangle / \langle x \rangle$, as the inclusion and then extend by linearity (but are regarded as ordinary functions between pointed sets).

3.2 The Action Monad

Definition 3.3. Let (M, μ', η') be a monoid in a monoidal category $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$. The **(left)** M**-action monad** is a monad on $\mathcal{C}$, which is defined as follows:

- The endofunctor is $M \otimes - : \mathcal{C} \to \mathcal{C}$.
- The components $\mu(A) : M \otimes (M \otimes A) \to M \otimes A$ of the multiplication are given by the composition

$$M \otimes (M \otimes A) \xrightarrow{\alpha(M,M,A)^{-1}} (M \otimes M) \otimes A \xrightarrow{\mu' \otimes \mathrm{id}_A} M \otimes A.$$

- The components of the unit $\eta(A) : A \to M \otimes A$ are given by the composition

$$A \xrightarrow{\lambda(A)^{-1}} 1 \otimes A \xrightarrow{\eta' \otimes \mathrm{id}_A} M \otimes A.$$

Lemma 3.4. The construction from the previous definition actually yields a monad on $\mathcal{C}$.

Proof. We have to check that the following two diagrams commute for every object A in $\mathcal{C}$:

$$\begin{tikzcd}
M \otimes (M \otimes (M \otimes A)) \arrow[r, "\mathrm{id}_M \otimes \mu(A)"] \arrow[d, "\mu(M \otimes A)"] & M \otimes (M \otimes A) \arrow[d, "\mu(A)"] \\
M \otimes (M \otimes A) \arrow[r, "\mu(A)"] & M \otimes A,
\end{tikzcd}$$

$$\begin{tikzcd}
M \otimes A \arrow[r, "\eta(M \otimes A)"] \arrow[dr, "\mathrm{id}_{M \otimes A}"'] & M \otimes (M \otimes A) \arrow[d, "\mu(A)"] & M \otimes A \arrow[l, "\mathrm{id}_M \otimes \eta(A)"'] \arrow[dl, "\mathrm{id}_{M \otimes A}"] \\
& M \otimes A. &
\end{tikzcd}$$

By the definition of a monoidal category, it holds

$$\begin{aligned}
&\alpha(M \otimes M, M, A)^{-1} \circ \alpha(M, M, M \otimes A)^{-1} \\
= {}&\alpha(M, M, M)^{-1} \otimes \mathrm{id}_A \circ \alpha(M, M \otimes M, A)^{-1} \circ \mathrm{id}_M \otimes \alpha(M, M, A)^{-1} : \\
&M \otimes (M \otimes (M \otimes A)) \to ((M \otimes M) \otimes M) \otimes A
\end{aligned}$$

and since (M, μ', η') is a monoid, we also have

$$\mu' \circ (\mu' \otimes \mathrm{id}_M) \circ \alpha(M, M, M)^{-1} = \mu' \circ \mathrm{id}_M \otimes \mu'.$$

Using those two identities, we calculate

$$\begin{aligned}\mu(A) \circ \mu(M \otimes A) &= \mu' \otimes \mathrm{id}_A \circ \alpha(M, M, A)^{-1} \circ \mu' \otimes \mathrm{id}_{M \otimes A} \circ \alpha(M, M, M \otimes A)^{-1} \\ &= \mu' \otimes \mathrm{id}_A \circ (\mu' \otimes \mathrm{id}_M) \otimes \mathrm{id}_A \circ \alpha(M \otimes M, M, A)^{-1} \circ \alpha(M, M, M \otimes A)^{-1} \\ &= \mu' \otimes \mathrm{id}_A \circ (\mu' \otimes \mathrm{id}_M) \otimes \mathrm{id}_A \circ \alpha(M, M, M)^{-1} \otimes \mathrm{id}_A \\ &\quad \circ \alpha(M, M \otimes M, A)^{-1} \circ \mathrm{id}_M \otimes \alpha(M, M, A)^{-1} \\ &= (\mu' \circ \mathrm{id}_M \otimes \mu') \otimes \mathrm{id}_A \circ \alpha(M, M \otimes M, A)^{-1} \circ \mathrm{id}_M \otimes \alpha(M, M, A)^{-1} \\ &= \mu' \otimes \mathrm{id}_A \circ (\mathrm{id}_M \otimes \mu') \otimes \mathrm{id}_A \circ \alpha(M, M \otimes M, A)^{-1} \circ \mathrm{id}_M \otimes \alpha(M, M, A)^{-1} \\ &= \mu' \otimes \mathrm{id}_A \circ \alpha(M, M, A)^{-1} \circ \mathrm{id}_M \otimes (\mu' \otimes \mathrm{id}_A) \circ \mathrm{id}_M \otimes \alpha(M, M, A)^{-1} \\ &= \mu(A) \circ \mathrm{id}_M \otimes \mu(A),\end{aligned}$$

where the second and penultimate equality hold since α^{-1} is natural. This shows that the first diagram commutes. The commutativity of the second diagram follows similarly. □

Example 3.5. 1. In 2.4.1, we established that a monoid in the monoidal category of sets $\mathcal{Set}$ is just an ordinary monoid. Thus, we recognize 2.8.2 as the action monad in that category. In Haskell, this monad is called `Writer`.

2. Let K be a field and consider the monoidal category of K-vector spaces $\mathcal{Vect}_K$ with the tensor product as monoidal product. From 2.4.2, we know that the monoids in this monoidal category are the associative K-algebras. For a fixed K-algebra M, the corresponding action monad consists of the endofunctor $M \otimes - : \mathcal{Vect}_K \to \mathcal{Vect}_K$, the multiplication

$$\mu(A) : M \otimes (M \otimes A) \to M \otimes A,\ (m \otimes (m' \otimes a)) \mapsto (m \cdot m') \otimes a,$$

and the unit

$$\eta(A) : A \to M \otimes A,\ a \mapsto 1 \otimes a.$$

3. The previous example can be generalized by fixing a ring R and examining the category of R-modules $\mathcal{Mod}_R$ with the tensor product as its monoidal product. In that case, a monoid is an R-algebra, and analogous formulas hold for the action monad. As a special case, the same holds for the category of abelian groups $\mathcal{Ab}$; here the monoids are just rings, see 2.4.3.

4. By 2.4.4, a monoid in $(\mathcal{Cat}, \times)$ is a strict monoidal small category. Let $(\mathcal{C}, \mu', 1)$ denote such a category. The endofunctor of the corresponding action monad is $\mathcal{C} \times - : \mathcal{Cat} \to \mathcal{Cat}$. The components of the multiplication μ are functors $\mu(\mathcal{A}) : \mathcal{C} \times (\mathcal{C} \times \mathcal{A}) \to \mathcal{C} \times \mathcal{A}$, mapping an object $(A, (A', B))$ to $(\mu'(A, A'), B)$ and a morphism $(f, (f', g))$ to $(\mu'(f, f'), g)$. The unit η is given by the family of functors $\eta(\mathcal{A}) : \mathcal{A} \to \mathcal{C} \times \mathcal{A}$, mapping an object A to $(1, A)$ and a morphism f to (id_1, f).

3.3 The Eilenberg-Moore Category

By 3.1, every adjunction induces a monad on the domain of its left adjoint. However, there are also monads which are not directly constructed from an adjunction; for example, the power set monad from 2.8.3. So it is natural to ask whether all monads arise this way. Eilenberg and Moore proved in 1965 that this is indeed the case, using the following construction, which was later named after them.

Definition 3.6. Let $\mathcal{C}$ be a category and (T, μ, η) be a monad on $\mathcal{C}$. The **Eilenberg-Moore category** $\mathcal{C}^T$ for T or (**category of T-algebras**) is defined as follows:

- Its objects (called *T-algebras*) are pairs $(A \in \mathcal{C}, a : T(A) \to A)$, such that the diagrams

$$\begin{tikzcd} A \arrow[r, "\eta(A)"] \arrow[d, "\mathrm{id}_A"'] & T(A) \arrow[dl, "a"] \\ A, & \end{tikzcd} \qquad \begin{tikzcd} T^2(A) \arrow[r, "\mu(A)"] \arrow[d, "T(a)"] & T(A) \arrow[d, "a"] \\ T(A) \arrow[r, "a"] & A \end{tikzcd}$$

  commute.

- Its morphisms $f : (A, a) \to (B, b)$ (called *T-algebra homomorphisms*) are morphisms $f : A \to B$ in $\mathcal{C}$, such that the square

$$\begin{tikzcd} T(A) \arrow[r, "T(f)"] \arrow[d, "a"] & T(B) \arrow[d, "b"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

  commutes.

Composition and identities are just those of $\mathcal{C}$.

Intuitively, T acts on a T-algebra in such a way that is compatible with the unit and multiplication of the monad. This becomes more apparent by considering some examples.

Example 3.7. 1. For the monad on $\mathcal{S}et$ induced by the free-forgetful adjunction from 3.2.1, an algebra is a set A with a map $a : A_+ \to A$ such that the previous diagrams commute. The commutativity of the triangle means that $a : A_+ \to A$ is the identity on $A \subset A_+$, whereas the square imposes no additional constraints.
Therefore, an algebra is a set with a special point $a \in A$, which is the image of the extra point under the map $a : A_+ \to A$.
A morphism $f : (A, a) \to (B, b)$ is a map $f : A \to B$, such that the diagram

$$\begin{tikzcd} A_+ \arrow[r, "f_+"] \arrow[d, "a"] & B_+ \arrow[d, "b"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

commutes. The function f_+ maps the extra point in A_+ to the extra point in B_+, so this condition demands $f(a) = b$. We conclude that the Eilenberg-Moore category for this monad is isomorphic to $\mathcal{S}et_*$ [Rie17, Exa. 5.2.6].

2. Consider the free group monad from 3.2.2 and write the elements of the free group as words of the form $[x, y, z]$. The group operation is concatenation $\circ$ of words. An algebra with respect to this monad is a set A, together with a map $a : \langle A\rangle \to A$, making the diagrams

$$\begin{tikzcd}
(*) & A \arrow[r, "\eta(A)"] \arrow[d, "\mathrm{id}_A"] & \langle A\rangle \arrow[dl, "a"] & \langle\langle A\rangle\rangle \arrow[r, "\mu(A)"] \arrow[d, "T(a)"] & \langle A\rangle \arrow[d, "a"] & (**) \\
 & A, & & \langle A\rangle \arrow[r, "a"] & A &
\end{tikzcd}$$

commute. We claim that defining

$$\cdot : A \times A \to A,\ (x, y) \mapsto a([x, y])$$

gives A a group structure with neutral element $a(e)$, where e denotes the neutral element of $\langle A\rangle$; that is, the empty word. Indeed, for $x \in A$, it holds

$$\begin{aligned}
a(e) \cdot x &= a([a(e), x]) \\
&\overset{(*)}{=} a([a(e), a(\eta(A)(x))]) \\
&= a([a(e), a([x])]) \\
&= (a \circ T(a))([e, [x]]) \\
&\overset{(**)}{=} (a \circ \mu(A))([e, [x]]) \\
&= a(e \circ [x]) \\
&= a([x]) \overset{(*)}{=} x.
\end{aligned}$$

Analogously, it follows $x \cdot a(e) = x$ for any $x \in A$, so $a(e)$ indeed acts as a neutral element. Associativity holds for any $x, y, z \in A$, as the caluclation

$$\begin{aligned}
(x \cdot y) \cdot z &= a([a([x, y]), z]) \\
&\overset{(*)}{=} a([a([x, y]), a([z])]) \\
&= (a \circ T(a))([[x, y], [z]]) \\
&\overset{(**)}{=} (a \circ \mu(A))([[x, y], [z]]) \\
&= a([x, y, z]) \\
&= (a \circ \mu(A))([[x], [y, z]]) \\
&\overset{(**)}{=} (a \circ T(a))([[x], [y, z]]) \\
&\overset{(*)}{=} a([x, a([y, z])]) = x \cdot (y \cdot z)
\end{aligned}$$

shows. The inverse of an element $x \in A$ is given by $a(x^{-1})$, because

$$\begin{aligned} a(x^{-1}) \cdot x &= a([a(x^{-1}), x]) \\ &\overset{(*)}{=} a([a(x^{-1}), a([x])]) \\ &= (a \circ T(a))([[x^{-1}], [x]]) \\ &\overset{(**)}{=} (a \circ \mu(A))([[x^{-1}], [x]]) \\ &= a([x^{-1}, x]) = a(e) \end{aligned}$$

and similarly $x \cdot a(x^{-1}) = a(e)$. It follows that every T-algebra has a group structure, defined as above.

A T-algebra homomorphism $f : (A, a) \to (B, b)$ is a map $f : A \to B$, making the diagram

$$\begin{tikzcd} \langle A \rangle \arrow[r, "T(f)"] \arrow[d, "a"] & \langle B \rangle \arrow[d, "b"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

commute. For $x, y \in A$, it holds

$$f(x \cdot y) = f(a([x, y])) = b(T(f)([x, y])) = b([f(x), f(y)]) = f(x) \cdot f(y),$$

so any T-algebra homomorphism is a group homomorphism.

Notice that a map $a : \langle A \rangle \to A$ amounts to a collection $(a_n)_{n \geq 0} : A^n \to A$ of maps. With this interpretation, the diagram $(*)$ just states that a_1 has to be the identity. The diagram $(**)$ means that given a word of words

$$[[x_{1,1}, \ldots, x_{1,m_1}], \ldots, [x_{n,1}, \ldots, x_{n,m_n}]],$$

first concatenating the words into one word, and then applying $a_{(m_1 + \cdots + m_n)}$ has to be the same as using a_n on the results of applying the appropriate a_i to each of the subwords.

Using this, it is easy to see that any group G constitutes a T-algebra. For this, define $g : \langle G \rangle \to G$ as follows:

- g_0 chooses the neutral element of G,
- g_1 is the identity,
- g_2 is the multiplication of G,
- g_n for $n > 2$ iterates the multiplication of G.

This is well-defined and makes the diagram $(**)$ commute, because multiplication in G is associative. Since the diagram $(*)$ commutes by definition of g_1, (G, g) is a T-algebra.

Moreover, every group group homomorphism $f : G \to H$ amounts to a T-algebra homomorphism $(G, g) \to (H, h)$, because for $x = [x_1, \dots, x_n] \in \langle G \rangle$, it holds

$$\begin{aligned} f(g(x)) &= f(g_n(x_1, \dots, x_n)) = f(x_1 \cdot \ldots \cdot x_n) = f(x_1) \cdot \ldots \cdot f(x_n) \\ &= h_n(f(x_1), \dots, f(x_n)) = h([f(x_1), \dots, f(x_n)]) = h(T(f)(x)). \end{aligned}$$

We conclude that the category of algebras of the free group monad is isomorphic to the category of groups $\mathscr{Grp}$.

3. With a completely analogous calculation as in the previous example, we see that the category of algebras of the free monoid monad from 3.2.3 is isomorphic to the category of monoids $\mathscr{Mon}$.

4. Let $(M, \cdot, 1)$ be a monoid. For the action monad $M \times - : \mathscr{Set} \to \mathscr{Set}$ from 3.5.1, an algebra is a set A, together with a map $a : M \times A \to A$, such that the diagrams

$$\begin{tikzcd} A \arrow[r, "\eta(A)"] \arrow[d, "\mathrm{id}_A"] & M \times A \arrow[dl, "a"] \\ A, & \end{tikzcd} \qquad \begin{tikzcd} M \times (M \times A) \arrow[r, "\mu(A)"] \arrow[d, "\mathrm{id}_M \times a"] & M \times A \arrow[d, "a"] \\ M \times A \arrow[r, "a"] & A \end{tikzcd}$$

commute. The first diagram states that $a(1, -) = \mathrm{id}_A$, while the second one means that $a(m, a(m', x)) = a(m \cdot m', x)$ for any $x \in A$, $m, m' \in M$. But this just describes a left M-action on A, so the algebras are precisely the left M-sets. For a morphism $f : (A, a) \to (B, b)$, the diagram

$$\begin{tikzcd} M \times A \arrow[r, "\mathrm{id}_M \times f"] \arrow[d, "a"] & M \times B \arrow[d, "b"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

commutes if and only if $f(a(m, x)) = b(m, f(x))$ for all $x \in A$, $m \in M$. Writing $\cdot$ for a and b, this means that $f(m \cdot x) = m \cdot f(x)$.
In conclusion, the Eilenberg-Moore category for the $M \times -$ monad is isomorphic to ${}_M\mathscr{Set}$.

5. Let R be a ring. Consider the monad $R \otimes - : \mathscr{Ab} \to \mathscr{Ab}$ from 3.5.3. An algebra is an abelian group A, equipped with a group homomorphism $a : R \otimes A \to A$, such that the diagrams

$$\begin{tikzcd} A \arrow[r, "\eta(A)"] \arrow[d, "\mathrm{id}_A"] & R \otimes A \arrow[dl, "a"] \\ A, & \end{tikzcd} \qquad \begin{tikzcd} R \otimes (R \otimes A) \arrow[r, "\mu(A)"] \arrow[d, "\mathrm{id}_R \otimes a"] & R \otimes A \arrow[d, "a"] \\ R \otimes A \arrow[r, "a"] & A \end{tikzcd}$$

commute. By the universal property of the tensor product, a is equivalent to a bilinear map $\cdot : R \times A \to A, (r,x) \mapsto a(r \otimes x)$. The first diagram means that $1 \cdot x = x$ for all $x \in A$ and the second diagram translates to $r \cdot (r' \cdot x) = (r \cdot r') \cdot x$ for all $r, r' \in R$, $x \in A$. A morphism $f : (A,a) \to (B,b)$ of algebras is a group homomorphism $f : A \to B$, making the diagram

$$\begin{tikzcd}
R \otimes A \arrow[r, "\mathrm{id}_R \otimes f"] \arrow[d, "a"] & R \otimes B \arrow[d, "b"] \\
A \arrow[r, "f"] & B
\end{tikzcd}$$

commute; that is, it satisfies $f(r \cdot x) = r \cdot f(x)$ for all $r \in R$, $x \in A$. We conclude that the Eilenberg-Moore category of this monad is isomorphic to $\mathcal{Mod}_R$.

Generalizing the last three examples, with a general enough notion of "M-actions", it is not hard to show that the Eilenberg-Moore category of an action monad is isomorphic to the category of M-actions, see [Sea13, Prop. 3.2.1].

The motivation for the definition of the Eilenberg-Moore category lies in the following theorem, which was proven by Eilenberg and Moore in 1965.

Theorem 3.8 ([EM65, Thm. 2.2])**.** For a monad (T, μ, η) on a category $\mathcal{C}$, there exists an adjunction

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left, "F^T"] & \mathcal{C}^T \arrow[l, bend left, "U^T"]
\end{tikzcd}$$

between $\mathcal{C}$ and the Eilenberg-Moore category $\mathcal{C}^T$, whose induced monad is (T, μ, η) and such that U^T is faithful.

Proof. The functor $U^T : \mathcal{C}^T \to \mathcal{C}$ is forgetful in nature; an object $(A \in \mathcal{C}, T(A) \to A)$ gets mapped to A and a morphism $f : (A,a) \to (B,b)$ becomes $f : A \to B$. It is clear that U^T is faithful.
The functor $F^T : \mathcal{C} \to \mathcal{C}^T$ is defined by

$$F^T(A) = \big(T(A), \mu(A) : T^2(A) \to T(A)\big), \quad F^T(f) = T(f),$$

where A is an object and $f : A \to B$ is a morphism in $\mathcal{C}$. It is easy to check that this is indeed a functor; for example $T(f)$ is a morphism in $\mathcal{C}^T$ by the naturality of μ.
Since $U^T \circ F^T = T$, the unit of the adjunction $F^T \dashv U^T$ can be defined to be the unit $\eta : \mathrm{id}_{\mathcal{C}} \Rightarrow T$ of the monad T. The counit of the adjunction is characterized as follows:

$$\epsilon : F^T \circ U^T \Rightarrow \mathrm{id}_{\mathcal{C}^T}, \quad \epsilon((A,a)) = a : (T(A), \mu(A)) \to (A,a).$$

The diagram

$$\begin{tikzcd}
T^2(A) \arrow[r, "T(a)"] \arrow[d, "\mu(A)"] & T(A) \arrow[d, "a"] \\
T(A) \arrow[r, "a"] & A
\end{tikzcd}$$

commutes by the definition of a T-algebra and thus shows that the components of ϵ are morphisms in $\mathcal{C}^T$. Moreover, ϵ is natural, because for $f : (A, a) \to (B, b)$ a morphism in $\mathcal{C}^T$, the diagram

$$\begin{tikzcd}
(T(A), \mu(A)) \arrow[r, "a"] \arrow[d, "T(f)"] & (A, a) \arrow[d, "f"] \\
(T(B), \mu(B)) \arrow[r, "b"] & (B, b)
\end{tikzcd}$$

commutes by the definition of a T-algebra homomorphism. It is left to show that $F^T \dashv U^T$ is actually an adjunction; i.e. that η and ϵ satisfy the triangle identities

$$\begin{tikzcd}
F^T \arrow[r, Rightarrow, "F^T \bullet \eta"] \arrow[dr, Rightarrow, "\mathrm{id}_{F^T}"'] & F^T \circ U^T \circ F^T \arrow[d, Rightarrow, "\epsilon \bullet F^T"] & U^T \arrow[r, Rightarrow, "\eta \bullet U^T"] \arrow[dr, Rightarrow, "\mathrm{id}_{U^T}"'] & U^T \circ F^T \circ U^T \arrow[d, Rightarrow, "U^T \bullet \epsilon"] \\
& F^T, & & U^T.
\end{tikzcd}$$

The first diagram commutes, because for $A \in \mathcal{C}$, it holds

$$\epsilon(F^T(A)) \circ F^T(\eta(A)) = \mu(A) \circ T(\eta(A)) = \mathrm{id}_T(A) = \mathrm{id}_{T(A)} = \mathrm{id}_{F^T}(A)$$

by the definition of a monad. In the same vein, the second diagram commutes, since for $(A, a) \in \mathcal{C}^T$, we observe

$$U^T(\epsilon((A, a))) \circ \eta(U^T((A, a))) = a \circ \eta(A) = \mathrm{id}_A = \mathrm{id}_{U^T}((A, a))$$

by the definition of a T-algebra. Finally, the whiskered counit $U^T \bullet \epsilon \bullet F^T : T^2 \Rightarrow T$ turns out to have the desired components due to the calculation

$$\big(U^T \bullet \epsilon \bullet F^T\big)(A) = U^T(\epsilon(T(A), \mu(A))) = U^T(\mu(A)) = \mu(A),$$

so we conclude that $F^T \dashv U^T$ is an adjunction which induces the monad (T, μ, η). □

3.4 The Kleisli Category

In the same year that Eilenberg and Moore published their paper solving the problem of finding an adjunction that induces a given monad, Kleisli found another construction which also solves this problem.

Definition 3.9. Let $\mathcal{C}$ be a category and (T, μ, η) be a monad on $\mathcal{C}$. The **Kleisli category** $\mathcal{C}_T$ consists of the following:

- Its objects are the objects of $\mathcal{C}$.
- A morphism $f : A \to B$ in $\mathcal{C}_T$ is a morphism $A \to T(B)$ in $\mathcal{C}$. To avoid confusion, we denote such a morphism by $A \rightsquigarrow B$.

For $A \in \mathcal{C}_T$, the unit $\eta(A) : A \to T(A)$ defines the identity morphism $A \rightsquigarrow A$ in $\mathcal{C}_T$. The composition of morphisms $f : A \to T(B)$ and $g : B \to T(C)$ is given by

$$A \xrightarrow{f} T(B) \xrightarrow{T(g)} T^2(C) \xrightarrow{\mu(C)} T(C).$$

We call the composition of morphisms in the Kleisli category *Kleisli composition*, in order to differentiate it from the composition in the original category $\mathcal{C}$.

The Kleisli category is especially important when using monads in functional programming languages, as we will soon see. It is also interesting to note that the original category defined by Kleisli in his paper [Kle65] was slightly different, in particular the morphisms $A \rightsquigarrow B$ in his category were defined to be morphisms $T(A) \to B$.
We give some examples to highlight the interesting structures that arise.

Example 3.10. 1. The category of relations $\mathcal{Rel}$ consists of sets as objects and a morphism $R : A \to B$ is a relation $R \subset A \times B$. The composition of two morphisms $R : A \to B$, $S : B \to C$ is given by

$$S \circ R = \{(a, c) : \exists b \in B : (a, b) \in R \wedge (b, c) \in S\} \subset A \times C$$

and the identity is the diagonal $\mathrm{id}_A = \{(a, a) : a \in A\} \subset A \times A$ [Bra17, Exa. 2.2.25]. The Kleisli category $\mathcal{Set}_{\mathcal{P}}$ of the power set monad from 2.8.3 is isomorphic to $\mathcal{Rel}$: The objects of $\mathcal{Set}_{\mathcal{P}}$ are just the sets and a morphism $f : A \rightsquigarrow B$, represented by $f : A \to \mathcal{P}(B)$, can be viewed as the relation $f = \{(a, b) : b \in f(a)\} \subset A \times B$. The composition $g \circ f$ of two morphism $f : A \to \mathcal{P}(B)$ and $g : B \to \mathcal{P}(C)$ in the Kleisli category is

$$x \mapsto (\mu(C) \circ \mathcal{P}(g) \circ f)(x) = \bigcup_{y \in f(x)} g(y).$$

With the above interpretation, this becomes

$$\begin{aligned} \Big\{(a, c) : c \in \bigcup_{y \in f(a)} g(y)\Big\} &= \{(a, c) : \exists y \in f(a) : c \in g(y)\} \\ &= \{(a, c) : \exists y \in B : (a, y) \in f \wedge (y, c) \in g\}, \end{aligned}$$

so the Kleisli category $\mathcal{Set}_{\mathcal{P}}$ is indeed isomorphic to $\mathcal{Rel}$.

2. An object in the Kleisli category of the monad induced by the free-forgetful adjunction from 3.2.1 is just a set and a morphism $A \rightsquigarrow B$ is a function $A \to B_+$. This can be interpreted as a partially-defined function, where the elements of A that are mapped to the extra basepoint, are thought of as having "undefined" output. This will become more apparent later, when we study the `Maybe` monad. The composition of two partially-defined functions amounts to the maximal partially-defined function. Therefore, the Kleisli category of this monad is isomorphic to the category of sets with partially-defined functions [Rie17, 5.2.10].

3. Consider the free group monad from 3.2.2. An object in the corresponding Kleisli category is just a set. A morphism $A \rightsquigarrow B$ is a function $A \to \langle B \rangle$ and for morphisms $f : A \rightsquigarrow B$, $g : B \rightsquigarrow C$, their composition $g \circ f : A \rightsquigarrow C$ is defined as follows: For

$a \in A$, f is applied to a, resulting in a list consisting of elements of B. Then g is applied to each of the elements of the list, yielding a list of lists. Finally, the sublists are concatenated and the resulting list is the image of a under $g \circ f$.

The following theorem stands in analogy to 3.8 and shows that given a monad, the Kleisli category of that monad permits an adjunction which induces that original monad.

Theorem 3.11 ([BW00, p.88])**.** For a monad (T, μ, η) on a category $\mathcal{C}$, there is an adjunction

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left, "F_T"] & \mathcal{C}_T \arrow[l, bend left, "U_T"]
\end{tikzcd}$$

between $\mathcal{C}$ and the Kleisli category $\mathcal{C}_T$, which induces the monad (T, μ, η).

Proof. The functor $F_T : \mathcal{C} \to \mathcal{C}_T$ acts on objects as the identity and maps a morphism $f : A \to B$ in $\mathcal{C}$ to

$$A \rightsquigarrow B, \quad A \xrightarrow{f} B \xrightarrow{\eta(B)} T(B).$$

The functor $U_T : \mathcal{C}_T \to \mathcal{C}$ sends an object $A \in \mathcal{C}_T$ to $T(A) \in \mathcal{C}$ and a morphism $f : A \to T(B)$ in $\mathcal{C}_T$ to

$$T(A) \to T(B), \quad T(A) \xrightarrow{T(f)} T^2(B) \xrightarrow{\mu(B)} T(B).$$

It is easy to check that F_T and U_T are indeed functors. For instance, F_T is well-behaved with respect to composition, because for morphisms $f : A \to B$, $g : B \to C$ in $\mathcal{C}$, the diagram

$$\begin{tikzcd}
A \arrow[r, "f"] & B \arrow[r, "\eta(B)"] \arrow[dr, "g"'] & T(B) \arrow[r, "T(g)"] & T(C) \arrow[r, "T(\eta(C))"] \arrow[rr, bend right, "\mathrm{id}_{T(C)}"'] & T^2(C) \arrow[r, "\mu(C)"] & T(C) \\
 & & C \arrow[ur, "\eta(C)"] & & &
\end{tikzcd}$$

commutes by naturality of η and by the definition of a monad.
With these definitions, $F_T \dashv U_T$ become adjoint functors. Indeed, for A and B objects in $\mathcal{C}$, a natural isomorphism $\alpha(A, B) : \mathrm{Hom}_{\mathcal{C}}(A, U_T(B)) \xrightarrow{\cong} \mathrm{Hom}_{\mathcal{C}_T}(F_T(A), B)$ is the same as a natural isomorphism $\mathrm{Hom}_{\mathcal{C}}(A, T(B)) \cong \mathrm{Hom}_{\mathcal{C}_T}(A, B)$, which exists by definition.
It is left to check that the adjunction $F_T \dashv U_T$ actually induces the monad (T, μ, η). Clearly $U_T \circ F_T = T$ and the unit of the adjunction $F_T \dashv U_T$ is indeed equal to the unit η of the monad:

$$\alpha^{-1}(A, F_T(A))(\mathrm{id}_{F_T(A)}) = \eta(A) : \; A \to T(A).$$

Similarly, the counit $\epsilon : F_T \circ U_T \Rightarrow \mathrm{id}_{\mathcal{C}_T}$ has components

$$\epsilon(A) = \alpha(U_T(A), A)(\mathrm{id}_{U_T(A)}) = \mathrm{id}_{T(A)} : \; T(A) \rightsquigarrow A,$$

so the multiplication $T^2 \Rightarrow T$ of the induced monad is

$$(U_T \bullet \epsilon \bullet F_T)(A) = U_T(\mathrm{id}_{T(A)}) = \mu(A) \circ T(\mathrm{id}_{T(A)}) = \mu(A) \circ \mathrm{id}_{T^2(A)} = \mu(A).$$

□

4 Category Theory in Haskell

In this section, we establish a connection between category theory and Haskell. But first, we briefly highlight some of Haskell's key features and principles and give an introduction to the language. We then define the cartesian closed category $\mathcal{Hask}$, which allows us to describe a large part of Haskell using the language of category theory. As a prerequisite for understanding monads in Haskell, we study the `Functor` type class.

4.1 Haskell as a Functional Programming Language

Many popular programming languages like Java or C are *imperative languages.* This kind of programming languages is characterized by having an implicit state, which can be altered by constructs (i.e. commands) in the source language. This state can be roughly thought of as a collection of “global” variables; that is, variables, which are available to every part of the program. This means that other constructs in the program can potentially change these values. We say that a construct has *side-effects*, if it alters the implicit state. In contrast, *declarative languages* do not have an implicit state, so in particular, no side-effects can occur and once a value is assigned to a variable, its value cannot change [Hud89, p.361].

As a very simple example, suppose x is a global variable in a larger program, which defines the function

```
function f() {
  x := 1
}
```

This pseudocode notation means that `f` denotes a function, which takes no arguments and assigns the value 1 to the global variable x. Thus, this function has side-effects, since it changes the value of x, which can be accessed by other parts of the program.

Functions, which have no side-effects and do not depend on the implicit state of the program are called *pure.* This means that given the same input (the same arguments), they will always return the same output.

For instance, suppose the program from the previous example also defines the function

```
function g() {
  if x=0 print "x is zero."
  else   print "x is not zero."
}
```

The function `g` takes no arguments, but still depends on x; i.e. it depends on the implicit state of the program and is thus not pure. If x is set to zero somewhere in the program, and then `g` is run, it will print *x is zero.* On the other hand, if `f` runs directly before `g`, then `g` will print *x is not zero.*

A *functional programming language* is a declarative language which models computations by functions. This usually works by defining a main function in terms of other functions, which in turn make use of other functions, until at the bottom level the language's keywords are used.

While there are some languages that support parts of both imperative and functional programming, Haskell is a purely functional programming language, so it only permits the use of pure functions. This means that a function can only return its result and cannot make any “global” changes, which closely resembles the notion of a mathematical function. In particular, a variable can freely be replaced by its value and vice versa, which oftentimes allows programmers to prove that a function acts as desired. This feature of (pure) functional programming languages is called *referential transparency* [Hug89, p.98]. The analogue in Haskell [1] to the function g from the example above is

```
g :: Int -> IO ()
g x = if x == 0
      then putStr "x is zero."
      else putStr "x is not zero."
```

Here g is a function taking one argument, namely x.

4.2 A Brief Introduction to Haskell

In the following, we briefly introduce the reader to the most important concepts in Haskell that are needed to understand the following pages.

Haskell is a *statically typed* language, which means that every expression in an Haskell program has a *data type*, which is known at compile time. A data type consists of a collection of possible *values*. It is often helpful to think of data types as sets. In this interpretation, the values become the elements of the corresponding set.
Many important data types come predefined in the standard library *Prelude*, which is automatically loaded by every Haskell program, unless explicitly disabled. We list some of those data types:

Data Type	Description	Exemplary Values
`Int`	fixed-precision integers	`-10, 0, 1, 42`
`Float`	real floating-point, single precision	`-2.1, 1.0, 5.0`
`Bool`	truth values	`False, True`
`Char`	single characters	`'1', 'a', '!'`
`String`	strings	`"haskell", "", "some words"`

[Mar10, 6.1,6.4].
As a functional programming language, every Haskell source file consists of a number of functions. Therefore, it is important to know how to work with functions in Haskell. Functions take a finite amount of parameters (arguments) as input and always return exactly one value.[2] Of course, it can also happen that a function loops indefinitely or encounters an error. In the first case, the program will not terminate, until the memory capacity of the computer is reached and the program crashes. In the second case, the program will terminate immediately.

[1] All the Haskell code in this text refers to the latest official release of Haskell, compiled using the *Glasgow Haskell Compiler (GHC) 8.10.2*.

[2] Technically, functions in Haskell always take exactly one argument, but this can be conveniently ignored due to Haskell's syntax. The details will be explained later.

The Haskell syntax to apply a function `f` to the values `x`, `y` and `z` is `f x y z`. Of course, this presupposes that `f` is a function taking three arguments and that `x`, `y` and `z` have suitable data types.
For instance, consider the Haskell function `squareInt`:

```
squareInt :: Int -> Int
squareInt x = x^2
```

As mentioned, every expression has a data type. It is usually not necessary, but often helpful, to explicitly state the type of an expression. This is exactly what the first line of the source code expresses; it establishes `squareInt` as a function, which takes a value of type `Int` and returns a value of type `Int`. The second line defines the actual function: `x` is the single argument of the function, and the part at the right of the equals sign tells Haskell how to compute the return value of the function. In this case, it just squares the given integer.
In order to apply the function `squareInt` to the number `5`, we write

```
squareInt 5
```

A fundamental concept in both mathematics and Haskell is the composition of functions. Inspired by the typical notation $\circ$ in mathematics, Haskell uses the symbol `.` to denote function composition. Its usage can be demonstrated by the following Haskell code:

```
squareInt . (\x -> x + 3)
```

The expression in the bracket is called a *lambda expression*. It defines a function that expects an argument `x` and returns $\mathtt{x} + 3$. So composing this lambda expression with `squareInt` yields a function, which takes an integer, adds 3 to it, squares it, and then returns the result. For instance, the expression

```
(squareInt . (\x -> x + 3)) 2
```

which applies our newly composed function to the integer 2, is equal to 25.

Another key tool in Haskell are lists. A list is a ordered collection of values of the same data type. For instance, the expression

```
[1,2,3]
```

is a list, consisting of three integers.
Lists are widely used in Haskell; for example, we can apply a function to every element in a list by using the `map` function:

```
map squareInt [1,2,3]
```

evaluates to `[1,4,9]`.

A *type variable* is a variable which represents a type. This is useful since there are functions that make sense for many types. For example, the signature of the `map` function is

```
map :: (a -> b) -> ([a] -> [b])
```

This means that map transforms a function a -> b into a function [a] -> [b]. a and b are type variables. When invoking map as in the example before, Haskell automatically chooses the correct type for a and b; in that case both were set to Int. A value is called *polymorphic* if its type contains at least one type variable. Thus, map is polymorphic and so is the empty list [], since it has type [a] and thus belongs to every list type. A polymorphic type essentially refers to a family of types. As another example, the polymorphic function const is defined as follows:

```
const :: a -> (b -> a)
const x = \y -> x
```

In words, const takes a value x of type a and returns a function b -> a, which - regardless of its argument - always returns x.
To distinguish normal data types like Int or String from type variables, the former are required to start with a capital letter, whereas the latter must begin in lower case or with an underscore [Mar10, 1.4].

Using the last two examples, we briefly discuss how to read the type signature of functions. First, brackets in the signature can sometimes be omitted, because -> is right-associative. For example, the signature of map simplifies to

```
map :: (a -> b) -> [a] -> [b]
```

and the type signature of const becomes

```
const :: a -> b -> a
```

By currying, we can alternatively view const as a function

```
const :: (a,b) -> a
```

The data type (a,b) consists of pairs, where the first entry is of type a and the second is of type b. This allows us to view const as a function that essentially takes two arguments and simply returns the first one.
In fact, all functions in Haskell technically take exactly one argument. However, the syntax of Haskell often hides this fact and allows us to conveniently define functions as if they had multiple arguments. For instance, the definition of const given above might look laborious to an experienced Haskell programmer. The following, equivalent definition would generally be preferred, since it is more concise and avoids unnecessary lambda expressions.

```
const :: a -> b -> a
const x _ = x
```

The underscore is used to denote a variable whose value is not accessed and thus is irrelevant. It could be replaced by y or a similar variable name.
As another, more complex example, consider the following two functions:

```
1 repeatChar :: Int -> Char -> String
2 repeatChar n c = repeatCharRecursive n c ""
3
4 repeatCharRecursive :: Int -> Char -> String -> String
5 repeatCharRecursive 0 c str = str
6 repeatCharRecursive n c str = repeatCharRecursive (n-1) c (c:str)
```

The function `repeatChar` takes an integer `n` and a character `c` and returns a string repeating that character `n` times.[3] To produce its return value, it simply calls the function `repeatCharRecursive` with the original values received and the empty string as an extra argument. As the name suggests, the `repeatCharRecursive` function uses recursion. If the supplied integer is zero, it simply returns the string. This is achieved by line 5 in the definition of `repeatCharRecursive`. Haskell's *pattern matching* means that if the supplied integer `n` is not zero, then the pattern in line 5 does not match and the corresponding expression is not returned. Instead, the pattern in the next line is checked. Names starting with a lowercase letter, like `n`, `c` or `str` always match. Therefore, when `n` is non-zero, the definition in line 6 matches. There the recursive step happens and the function calls itself with the by one reduced integer, the same character, and the string `c:str`, which is the string `str`, but with the character `c` attached at the front.
For example, the expression

```
repeatChar 3 'a'
```

is evaluated as follows:

```
repeatChar 3 'a'
▷ repeatCharRecursive 3 'a' ""
 ▷ repeatCharRecursive 2 'a' "a"
  ▷ repeatCharRecursive 1 'a' "aa"
   ▷ repeatCharRecursive 0 'a' "aaa"
    ▷ "aaa"
```

Functions with multiple arguments can be *partially applied*, which refers to calling a function with less arguments than it actually requires [HF92, p.10]. This relies on the fact that all functions in Haskell are curried: By calling a function f, which requires n arguments, with the parameters $a_1, \ldots, a_k$, $k < n$, we derive a function taking $n - k$ parameters $a_{k+1}, \ldots, a_n$ and returning $f(a_1, \ldots, a_n)$. For example, the expression

```
const True
```

has data type `b -> True` and it could further be used in the following way:

```
map (const True) [1,2,3]
```

This expression evaluates to `[True,True,True]`.

[3] As described before, `repeatChar` is technically curried. It takes an integer `n` and returns a function which takes a character `c` and yields the respective string. We will continue to say that a function takes multiple arguments, well knowing that strictly speaking this is not true.

A *type constructor* takes zero or more data types to create a new data type. If the required number of data types is zero, this just amounts to an ordinary data type like `String` or `Int`. However, type constructors which take at least one data type are not data types themselves [Lip11, p.117]. For example, the list type constructor, often denoted by `[]`, needs precisely one data type `T` in order to produce a proper data type, namely the list data type containing elements of type `T`. For instance, applying the list type constructor to the type `Int` yields the data type `[Int]`.

As a final topic in our short explanation of the core features of Haskell, we look at type classes, which were originally introduced in the paper [WB89]. A *type class* is an interface that defines some behavior. If a data type is an *instance* of a type class, it implements and supports that behavior.
For example, the `Eq` class is a collection of types, for which a notion of "equal" and "unequal" exists. An instance needs to define the function `==` for checking equality or the function `/=` for checking inequality. Many types like `Int`, `Float`, `Char` etc. are instances of `Eq` [Jon95, 3.1].

4.3 The Category $\mathcal{Hask}$

Haskell first appeared in 1990 and was named after the logician Haskell B. Curry. It was designed by a committee, constituted of many academics with a computer science or mathematical background. As a result, the syntax is very close to conventional mathematical notation [Hud+07, p.12].
For example, if we model the mathematical notion of a set as a list in Haskell, then the set

$$X = \big\{(x, x^2) : x \in \{1, \ldots, 5\}\big\}$$

translates to the list

```
x = [(x,x^2) | x <- [1..5]]
```

in Haskell.
Not only is the syntax of Haskell inspired by common mathematical notation, but there are also many concepts of the language which were directly taken from category theory. To understand the influence of category theory on Haskell, we first have to define a category that describes Haskell.

It is tempting to define the following pseudo-category:

- The objects are the data types of Haskell.
- A morphism $f : a \to b$ is a function `f :: a -> b` in Haskell.

The composition is given by the `.` operator and the identity morphism for a data type `a` is `id :: a -> a`, which is the function taking a value of data type `a` and returning that same value. Equality of functions means that they return the same output for every input of the correct type.

However, the definition of equality of morphisms is problematic: A function `a -> b` is also a value of type `a -> b`. Thus, equality of two functions `a -> b` should also mean that their values agree. In the following, we construct two functions for which this does not hold.

In denotational semantics, an area of computer science which tries to formalize the meaning of programming languages, the symbol $\perp$ (called *bottom*) represents computations that never complete successfully. In particular, this includes functions which loop indefinitely or yield an error. In Haskell, the `undefined` value corresponds to bottom and evaluating it leads to an error. `undefined` is polymorphic and can be of any type.

The `seq` function in Haskell plays a special role. It is defined as follows:

$$\texttt{seq}\ x\ y = \begin{cases} \perp & \text{if } x = \perp \\ y & \text{otherwise} \end{cases} .$$

Now consider the functions

```haskell
undef1 :: a -> b
undef1 = undefined
undef2 :: a -> b
undef2 = const undefined
```

The `undef1` function is simply defined to be `undefined`, which as mentioned can be of any type, so in particular of type `a -> b`. In contrast, in the definition of `undef2`, `undefined` has type `b`. By definition of the `const` function, `undef2` ignores its argument of type `a` and evaluates `undefined`; i.e. it results in an error.

Using the `seq` function, we see that

```haskell
seq undef1 0 = undefined
seq undef2 0 = 0
```

so the value of `undef1` and `undef2` must be different. However, `undef1` and `undef2` represent the same function, since they return the same output given the same input.

One approach to solve this problem works by introducing a slightly different composition function, namely `f .! g = ((.) $! f) $! g` [Wikibook]. We will not follow this approach further.

Instead, we restrict ourselves to a subset of Haskell, such that types do not have bottom values and all functions terminate. We call the resulting category $\mathcal{Hask}$.

Due to our restrictions, $\mathcal{Hask}$ is very similar to $\mathcal{Set}$ and in particular cartesian closed [HaskellWiki]:

- The data type `()` constitutes a terminal object. There exists a unique value of type `()`, which is also denoted by `()`.

- The product of two data types `a` and `b` is the data type `(a,b)` together with the projections `fst :: (a,b) -> a` and `snd :: (a,b) -> b`. The values of `(a,b)` are pairs. `fst` and `snd` return the first or second component of the pair, respectively.

- For two data types `a` and `b`, the data type `a -> b` is the corresponding exponential object.

There also is a different idea to construct a cartesian closed category that describes Haskell. It uses the fact that Haskell is based on the *typed lambda calculus*, a formal system in logic to express computations. In [LS88, 10,11], it is demonstrated that any typed lambda calculus gives rise to a cartesian closed category. It is also shown that a cartesian closed category corresponds to a typed lambda calculus, which is called its *internal language*. The two constructions give rise to an equivalence between appropriately defined categories.
The details require knowledge about the lambda calculus and type theory; the interested reader is directed to mentioned book. It is also interesting to note that the `seq` function is not definable in the lambda calculus. This is one of the reasons why the inclusion of `seq` into Haskell was controversial [Hud+07, 10.3].

Another approach is to to use a category of cpos to model Haskell; more information on cpos can be found in [GS90].

The following table summarizes the connection between $\mathscr{Hask}$ and $\mathscr{Set}$ and illustrates common notation.

Category Theory	$\mathscr{Hask}$	$\mathscr{Set}$
object	data type `a` with values (without bottom)	sets A with elements
morphism	(terminating) function `f :: a -> b`	function $f : A \to B$
terminal object	`()`	singleton set $\{*\}$
product	`(a,b)`	$A \times B$
exponential object	data type `a -> b`	hom-set $\mathrm{Hom}(A, B)$

4.4 The `Functor` Type Class

The endofunctors $\mathscr{Hask} \to \mathscr{Hask}$ are modeled by the `Functor` type class in Haskell. The most important part of its *class declaration* looks like this:

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

This means that a type constructor `f`, which takes one type argument, is a functor, if it implements the function `fmap`. It must additionally satisfy the axioms of a functor from category theory; i.e. it must hold that

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Identities like these cannot be enforced by Haskell. Instead, the programmer has to check that they hold if a new instance of the `Functor` type class is defined. Of course, all the instances of `Functor` from the standard library satisfy the functor laws [Mar10, 13.1]. Applying `fmap` to a morphism `a -> b` is called *lifting* the morphism.[4]
Notice that the type constructor itself represents the action of the endofunctor on the objects of $\mathscr{Hask}$. The `fmap` function corresponds to the action on the morphisms of $\mathscr{Hask}$. For example, the list type constructor becomes a functor by the following definition:

[4] Recall that by right-associativity of `->`, `fmap` is just a function taking a morphism `a -> b` and returning the "lifted" morphism `f a -> f b`.

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

The second line is short notation for `fmap f list = map f list`, so a function `f :: A -> B` becomes a function `[A] -> [B]` by applying `f` to every entry in the supplied list.

5 Monads in Haskell

In this section, we put our gained knowledge to work by looking at some applications of monads in Haskell. We start by studying the type class `Monad` and after that define *extension systems*, which are an equivalent way to describe monads. We then fill the concept of monads in Haskell with life by looking at some examples. Those include the list monad, the `Maybe` monad and the `Writer` monad.

5.1 The `Monad` Type Class

We discussed that Haskell, as a purely functional programming language, makes it impossible to change the value of a variable once it is set. This might seem limiting at first. Indeed, the reader surely knows of many algorithms which make use of a regularly updated counter. Since computations depending on a (global) state are encountered frequently, it is natural to search for an elegant solution to this problem. In this case, the problem served as motivation for introducing the `State` monad into Haskell. Another significant problem is dealing with input and output, often denoted by I/O. How can asking for user input be modeled as a pure function? After all, a function which simply returns the user's input is not pure, since the return value might change when calling the function multiple times. Again, this problem was overcome with the help of monads; more precisely, by defining the *I/O monad*.

Many more applications of monads in Haskell have since been discovered and implemented. Consequently, monads form a vital part of the Haskell programming language.

Historically, Moggi was one of the first to realize that monads can be used in functional programming languages to model computations. A key principle of his work on monads was the distinction between ordinary functions and functions that perform computations. He considered monads (T, μ, η) to be computational models, where the type $T(A)$ was interpreted as the object of computations of type A [Mog88, Def. 2.1]. Since he viewed programs as functions from values to computations, the Kleisli category appeared to be more suitable for the interpretation of programs than the original category. As a consequence of this attitude, he worked with *extension systems* (which he called *Kleisli triples*). Extension systems constitute an equivalent description of monads, as we will see in 5.2. They were first defined by Manes in [Man76, Exercise 1.3.12, p.32].

The reason that Moggi preferred extension systems to conventional monads is that, while the latter are more suitable for abstract manipulation, they are less convenient for describing computations, which was the application Moggi was interested in. Using extension systems, Moggi was able to describe ideas like nondeterminism or side-effects [Mog91, Exa. 1.4].

In [Wad92b; Wad92a], Wadler applied Moggi's ideas to Haskell.

In order to understand how monads can be applied to Haskell, we have to define extension systems.

Definition 5.1. An **extension system** (or **Kleisli triple**) $(T, \eta, -^T)$ on a category $\mathcal{C}$ consists of the following:

- For every object $A \in \mathcal{C}$, there is an object $T(A) \in \mathcal{C}$ and a morphism $\eta(A) : A \to T(A)$.
- For every morphism $f : A \to T(B)$ in $\mathcal{C}$, there exists a morphism $f^T : T(A) \to T(B)$.

For every object $A, B, C \in \mathcal{C}$ and all morphisms $f : A \to T(B)$, $g : C \to T(A)$ in $\mathcal{C}$, it must hold that

$$(\eta(A))^T = \mathrm{id}_{T(A)}$$

and the diagrams

$$\begin{tikzcd} A \arrow[r, "\eta(A)"] \arrow[dr, "f"'] & T(A) \arrow[d, "f^T"] \\ & T(B), \end{tikzcd} \qquad \begin{tikzcd} T(C) \arrow[r, "g^T"] \arrow[dr, "(f^T \circ g)^T"'] & T(A) \arrow[d, "f^T"] \\ & T(B) \end{tikzcd}$$

must commute.

The significance of extension systems lies in the following theorem, which tells us that an extension system on a category is just a monad on that category and vice versa. This connection served as motivation for Manes to introduce extension systems in [Man76].

Theorem 5.2. Let $\mathcal{C}$ be a category. Then there is a bijective correspondence between the monads and the extension systems on $\mathcal{C}$:
An extension system $(T, \eta, -^T)$ induces the monad (T, μ, η) as follows:

- T is transformed into an endofunctor $\mathcal{C} \to \mathcal{C}$ by defining $T(f) = (\eta(B) \circ f)^T$ for a morphism $f : A \to B$ in $\mathcal{C}$.
- The multiplication $\mu : T^2 \Rightarrow T$ is defined to be $\mu(A) = (\mathrm{id}_{T(A)})^T$.

On the other hand, a monad (T, μ, η) gives rise to the extension system $(T, \eta, -^T)$:

- T is restricted to its action on objects.
- $-^T$ is given by $f^T = \mu(B) \circ T(f)$ for a morphism $f : A \to T(B)$ in $\mathcal{C}$.

Proof. Let $(T, \eta, -^T)$ be an extension system on $\mathcal{C}$. Clearly, it holds $T(\mathrm{id}_A) = (\eta(A))^T = \mathrm{id}_{T(A)}$ and for morphisms $f : A \to B$, $g : B \to C$ in $\mathcal{C}$, we observe

$$\begin{aligned} T(g) \circ T(f) &= (\eta(C) \circ g)^T \circ (\eta(B) \circ f)^T \\ &= \big((\eta(C) \circ g)^T \circ \eta(B) \circ f\big)^T \\ &= (\eta(C) \circ g \circ f)^T = T(g \circ f), \end{aligned}$$

where we first utilized the second diagram, and then the first one.
μ is natural, because for any morphism $f : A \to B$ in $\mathcal{C}$, it holds

$$\begin{aligned}
\mu(B) \circ T^2(f) &= \big(\mathrm{id}_{T(B)}\big)^T \circ \big(\eta(T(B)) \circ (\eta(B) \circ f)^T\big)^T \\
&= \Big(\big(\mathrm{id}_{T(B)}\big)^T \circ \eta(T(B)) \circ (\eta(B) \circ f)^T\Big)^T \\
&= \big(\mathrm{id}_{T(B)} \circ (\eta(B) \circ f)^T\big)^T \\
&= \big((\eta(B) \circ f)^T \circ \mathrm{id}_{T(A)}\big)^T \\
&= (\eta(B) \circ f)^T \circ (\mathrm{id}_{T(A)})^T = T(f) \circ \mu(A)
\end{aligned}$$

by the definition of an extension system. It is left to prove that the diagrams

$$\begin{tikzcd}
T^3 \arrow[r, Rightarrow, "T\bullet\mu"] \arrow[d, Rightarrow, "\mu\bullet T"] & T^2 \arrow[d, Rightarrow, "\mu"] \\
T^2 \arrow[r, Rightarrow, "\mu"] & T,
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, Rightarrow, "\eta\bullet T"] \arrow[dr, Rightarrow, "\mathrm{id}_T"'] & T^2 \arrow[d, Rightarrow, "\mu"] & T \arrow[l, Rightarrow, "T\bullet\eta"'] \arrow[dl, Rightarrow, "\mathrm{id}_T"] \\
& T &
\end{tikzcd}$$

commute. Indeed, for any object $A \in \mathcal{C}$, it holds

$$\begin{aligned}
\mu(A) \circ T(\mu(A)) &= (\mathrm{id}_{T(A)})^T \circ (\eta(T(A)) \circ \mu(A))^T \\
&= \big((\mathrm{id}_{T(A)})^T \circ \eta(T(A)) \circ (\mathrm{id}_{T(A)})^T\big)^T \\
&= \big(\mathrm{id}_{T(A)} \circ (\mathrm{id}_{T(A)})^T\big)^T \\
&= \big((\mathrm{id}_{T(A)})^T \circ \mathrm{id}_{T^2(A)}\big)^T \\
&= (\mathrm{id}_{T(A)})^T \circ (\mathrm{id}_{T^2(A)})^T = \mu(A) \circ \mu(T(A)),
\end{aligned}$$

so the first diagram commutes. The left triangle of the second diagram commutes by the calculation

$$\mu(A) \circ \eta(T(A)) = (\mathrm{id}_{T(A)})^T \circ \eta(T(A)) = \mathrm{id}_{T(A)}$$

and the commutativity of the right triangle follows analogously. We conclude that (T, μ, η) is a monad.
On the other hand, let (T, μ, η) be a monad on $\mathcal{C}$. Obviously, it holds $(\eta(A))^T = \mu(A) \circ T(\eta(A)) = \mathrm{id}_{T(A)}$. Moreover, the diagrams in the definition of an extension system commute: Let $f : A \to T(B)$ and $g : C \to T(A)$ be morphisms in $\mathcal{C}$. It holds

$$f^T \circ \eta(A) = \mu(B) \circ T(f) \circ \eta(A) = \mu(B) \circ \eta(T(B)) \circ f = f$$

by the naturality of η and the definition of a monad. Similarly, we see

$$\begin{aligned}
f^T \circ g^T &= \mu(B) \circ T(f) \circ \mu(A) \circ T(g) = \mu(B) \circ \mu(T(B)) \circ T^2(f) \circ T(g) \\
&= \mu(B) \circ T(\mu(B)) \circ T^2(f) \circ T(g) = \mu(B) \circ T(f^T \circ g) = \big(f^T \circ g\big)^T,
\end{aligned}$$

so $(T, \eta, -^T)$ is an extension system. It is clear that the two constructions are inverses to one and another. □

The extension system $(T, \eta, -^T)$ of a monad (T, μ, η) permits a simple interpretation of the composition in the Kleisli category $\mathcal{C}_T$; that is, the composition of two morphism $f : A \to T(B)$, $g : B \to T(C)$ in the Kleisli category is just $g^T \circ f$.
This observation entails a simple interpretation of the axioms of an extension system: The equation $(\eta(A))^T = \mathrm{id}_{T(A)}$ means that the unit η is a left identity with respect to Kleisli composition. Similarly, the diagrams

$$\begin{tikzcd}
A \arrow[r, "\eta(A)"] \arrow[dr, "f"'] & T(A) \arrow[d, "f^T"] \\
& T(B),
\end{tikzcd}
\qquad
\begin{tikzcd}
T(C) \arrow[r, "g^T"] \arrow[dr, "(f^T \circ g)^T"'] & T(A) \arrow[d, "f^T"] \\
& T(B)
\end{tikzcd}$$

express that η is a right identity with respect to Kleisli composition and that Kleisli composition is associative.

After these theoretical observations, we draw the connection to Haskell. The extension systems (i.e. the monads) are represented by the Monad type class. The relevant part of that type class looks as follows:

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

The first line states that every monad m is necessarily an instance of the Applicative type class, which consists of *applicative functors*. Applicative functors were first introduced in the paper [MP08] and are functors with extra structure; more precisely, their theoretical equivalent in category theory are lax monoidal functors with tensorial strength.
It is not surprising that every instance of Applicative must also belong to the Functor type class, so every instance of Monad is necessarily an instance Functor.
The other two lines state that every instance of Monad must define the functions >>= (called *bind*) and return. Every instance should also satisfy the *monad laws*

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

for all[5]

```haskell
k :: a -> m b,   m :: m a,   h :: b -> m c.
```

If we interpret return as the unit η and use the equation $-(>>=)f = f^T$ for >>=, then the previous axioms translate to the following (using mathematical notation for the application of a function to a value):

- $k^T(\eta(a)(a)) = k(a)$

[5] It is common to denote Haskell values by the same name as their type and we adhere to that convention.

- $(\eta(a))^T(m) = m$
- $\big(x \mapsto h^T(k(x))\big)^T(m) = h^T\big(k^T(m)\big)$.

These equations precisely correspond to those of an extension system on $\mathscr{H}ask$.

If one prefers the conventional definition of a monad, one can use the `join` function, which resembles the multiplication μ. Its definition

```
join :: (Monad m) => m (m a) -> m a
join x = x >>= id
```

states the type signature and provides a default implementation, whose correctness is an immediate consequence of 5.2.

We mentioned that the Kleisli category plays an important role when modeling computations by monads. The Kleisli composition (left-to-right) is embodied by the function[6]

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \x -> f x >>= g
```

The default implementation in the second line is motivated by our previous observation that the Kleisli composition of $f : A \to T(B)$ and $g : B \to T(C)$ is given by $g^T \circ f$.

In summary, the monads in the `Monad` type class are the extension system on $\mathscr{H}ask$. The following table gives an overview on how notions from category theory translate to Haskell:

Category Theory	Haskell
extension $-^T : A \to T(B) \rightsquigarrow T(A) \to T(B)$	`(>>=) :: m a -> (a -> m b) -> m b`
unit $\eta(A) : A \to T(A)$	`return :: a -> m a`
multiplication $\mu(A) : T^2(A) \to T(A)$	`join :: m (m a) -> m a`

Kleisli composition	`(>=>) :: (a -> m b) ->`
	`(b -> m c) -> (a -> m c)`

Since monads are an integral part of Haskell, it is not surprising that many more functions for working with monads have been defined. There even exists a special Haskell syntax, called *do-notation*, in order to make working with monads in Haskell more convenient. Since we are mainly interested in the theoretical uses of monads, this feature of the language will not be covered here.
Instead, we turn our attention to concrete examples of monads in Haskell and rediscover some familiar monads.

5.2 The List Monad

The purpose of the list monad is to represent *nondeterministic computations*. We already saw that lists are functors in Haskell, where `fmap = map`. In fact, lists are also monads, due to the following implementation:

[6] Alternatively, the function `<=<` can be used if one preferes right-to-left composition.

```
instance Monad [] where
  xs >>= f = [y | x <- xs, y <- f x]
  return x = [x]
```

Here the second line defines `>>=` as follows: `xs >>= f` is defined to be the list comprehension `[y | x <- xs, y <- f x]`. This means that the resulting list consists of the elements of `f x`, where `x` is drawn from the list `xs`. The third line establishes `return` for the list monad: Given a value `x`, it returns the singleton list `[x]` consisting only of that value.

Since a list is nothing but a formal word, this monad is just the free monoid monad from 3.2.3. Indeed, the units agree, and the concatenation $\mu(X) : [[X]] \to [X]$ induces $-^T$ and thus `>>=`: For $f : A \to T(B)$, it holds $f^T = \mu(B) \circ T(f)$ by 5.2, and this translates to

$$\texttt{xs >>= f} \;=\; \texttt{f}^T\texttt{(xs)} \;=\; \mu(B)(\texttt{[f x | x <- xs]}) \;=\; \texttt{[y | x <- xs, y <- f x]}.$$

We calculated the Eilenberg-Moore category of this monad in 3.7.3, it was just the category of monoids $\mathscr{Mon}$.

5.3 The `Maybe` Monad

In Haskell, `Maybe` is a type constructor that takes one type argument to produce a data type. The values of the data type `Maybe a` are of the form `Just a` or `Nothing`, where `Nothing` is a polymorphic value. For example, the values `Just 0`, `Just 5`, `Nothing` all share the same type, namely `Maybe Int`. If we interpret a type `a` as a set A, then `Maybe a` amounts to the set $A \cup \{*\}$, where $*$ is an arbitrary element not contained in A. This element $*$ corresponds to the `Nothing` value.

`Maybe` becomes a functor by the following declaration:

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

This means that given a function `f :: a -> b`, it is lifted to a function `Maybe a -> Maybe b` as follows: If it gets a `Nothing` value, it returns `Nothing`. Otherwise, it applies the function as in the third line. This functor might seem familiar to the reader. Indeed, this is the endofunctor $(-)_+$ from 3.2.1, translated to Haskell. In fact, the `Maybe` monad is just the monad arising from the free-forgetful adjunction between $\mathscr{Set}$ and $\mathscr{Set}_*$, explained in the same example. This is reflected in the monad declaration of `Maybe`:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= _ = Nothing
  return x = Just x
```

In 3.7.1, we showed that the Eilenberg-Moore category of this monad is isomorphic to $\mathscr{Set}_*$. The Kleisli category of this monad is isomorphic to the category of sets with partially-defined functions by 3.10.2.

This reveals the main use of the `Maybe` monad in functional programming: It is used to compose functions that can fail. For example, we might write a function that calculates a single real root of a given polynomial of degree 2. Then `Maybe Float` is a suitable return type for our function, since there are polynomials of degree 2 that have no real roots. In that case, the state of failure is represented by the `Nothing` value.

5.4 The `Writer` Monad

The `Writer` monad in Haskell corresponds to 3.5.1, which is a special case of the action monad (see 3.3). Notice that this monad also arises from a suitable free-forgetful adjunction, as mentioned in 3.2.6. As in both examples, let M be a monoid.
Since the Haskell definition of `Writer` is rather complicated and in particular depends on the definition of the more general `WriterT` type, we only sketch its use in Haskell: When identifying $\mathscr{Hask}$ with a subcategory of $\mathscr{Set}$, a Kleisli morphism of the writer monad becomes a map $X \to Y \times M$. This can be interpreted as a process that not only computes an element in Y (i.e. in Haskell terms a value of the corresponding type), but also produces an element in M. For example, the monoid could be the data type `String` with concatenation of strings as the multiplication of the monoid and the empty list as the identity element. In that case, the `Writer` monad can be used to work with functions, while “logging” information about the functions that are called [Per20, Exa. 5.1.14].
We calculated the Eilenberg-Moore category of this monad in 3.7.4, it turned out to be isomorphic to the category ${}_M\mathscr{Set}$.

We covered three important monads in Haskell, but there are many more. For example, there is the previously mentioned `State` monad or the `Reader` monad.
We finish our list of concrete monads in Haskell and conclude by looking at strong and enriched monads and their connection to Haskell.

5.5 Strong and Enriched Monads

We saw the connection between the extension operation $-^T$ and Haskell's bind operation `>>=`, namely that $-(>>=)f = f^T$. We now want to find an alternative description for `>>=`. This requires some definitions.

Definition 5.3. A monoidal category $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ is called **symmetric**, if it can be equipped with a natural transformation $\beta(A, B) : A \otimes B \to B \otimes A$ (called *braiding*), such that for all objects $A, B, C \in \mathcal{C}$, the diagram

$$\begin{tikzcd}
(A \otimes B) \otimes C \arrow[r, "\alpha(A,B,C)"] \arrow[d, "\beta(A,B)\otimes \mathrm{id}_C"] & A \otimes (B \otimes C) \arrow[r, "\beta(A,B\otimes C)"] & (B \otimes C) \otimes A \arrow[d, "\alpha(B,C,A)"] \\
(B \otimes A) \otimes C \arrow[r, "\alpha(B,A,C)"] & B \otimes (A \otimes C) \arrow[r, "\mathrm{id}_B \otimes \beta(A,C)"] & B \otimes (C \otimes A)
\end{tikzcd}$$

commutes and that

$$\beta(B, A) \circ \beta(A, B) = \mathrm{id}_{A \otimes B}.$$

Definition 5.4. A symmetric monoidal category $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ is called **closed**, if for every object $B \in \mathcal{C}$, the functor $- \otimes B : \mathcal{C} \to \mathcal{C}$ has a right adjoint functor $[B, -] : \mathcal{C} \to \mathcal{C}$.

For two objects $B, C \in \mathcal{C}$, the object $[B, C]$ is called *internal hom* of B and C. We call the counit $\mathrm{eval}(A, B) : [A, B] \otimes A \to B$ *evaluation.*

One can show that a closed symmetric monoidal category allows the construction of a functor $[-, -] : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{C}$. This functor is similar to the well-known hom-functor $\mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{S}et$ and is therefore called *internal hom-functor.*

Example 5.5. Every cartesian monoidal category (see 2.2.1) is symmetric. The main reason for this is the essential uniqueness of the product. If a cartesian monoidal category is additionally closed, it is called *cartesian closed category.*

We now introduce strong functors.

Definition 5.6. Let $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ be a monoidal category.
A **strong functor** (or a **functor with tensorial strength**) on $\mathcal{C}$ is an endofunctor $F : \mathcal{C} \to \mathcal{C}$, together with a natural transformation (called *tensorial strength*)

$$t(A, B) : A \otimes F(B) \to F(A \otimes B),$$

such that the diagrams

$$\begin{tikzcd}
(A \otimes B) \otimes F(C) \arrow[rr, "t(A\otimes B, C)"] \arrow[d, "\alpha(A,B,F(C))"] & & F((A \otimes B) \otimes C) \arrow[d, "F(\alpha(A,B,C))"] \\
A \otimes (B \otimes F(C)) \arrow[r, "\mathrm{id}_A \otimes t(B,C)"] & A \otimes F(B \otimes C) \arrow[r, "t(A, B\otimes C)"] & F(A \otimes (B \otimes C)),
\end{tikzcd}$$

$$\begin{tikzcd}
1 \otimes F(A) \arrow[r, "t(1,A)"] \arrow[dr, "\lambda(F(A))"'] & F(1 \otimes A) \arrow[d, "F(\lambda(A))"] \\
& F(A)
\end{tikzcd}$$

commute for all objects $A, B, C \in \mathcal{C}$.

Definition 5.7. A **strong monad** on a monoidal category $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ is a monad (T, μ, η) together with a natural transformation

$$t(A, B) : A \otimes T(B) \to T(A \otimes B),$$

such that (T, t) is a strong functor and that the diagrams

$$\begin{tikzcd}
A \otimes B \arrow[d, "\mathrm{id}_A \otimes \eta(B)"'] \arrow[dr, "\eta(A\otimes B)"] & \\
A \otimes T(B) \arrow[r, "t(A,B)"] & T(A \otimes B)
\end{tikzcd}$$

$$\begin{tikzcd}
A \otimes T^2(B) \arrow[r, "t(A,T(B))"] \arrow[d, "\mathrm{id}_A \otimes \mu(B)"] & T(A \otimes T(B)) \arrow[r, "T(t(A,B))"] & T^2(A \otimes B) \arrow[d, "\mu(A \otimes B)"] \\
A \otimes T(B) \arrow[rr, "t(A,B)"] & & T(A \otimes B)
\end{tikzcd}$$

commute.

The diagrams express the compatibility of the monad structure with the tensorial strength. Next, we look at some notions from enriched category theory.

Definition 5.8. Let $(\mathcal{V}, \otimes, 1, \alpha, \lambda, \rho)$ be a monoidal category. A (small) **$\mathcal{V}$-enriched category** (or **$\mathcal{V}$-category**) $\mathcal{C}$ consists of

- a set of objects $\mathrm{Ob}(\mathcal{C})$ (called *objects*);
- for any two objects $A, B \in \mathcal{C}$ an object $\mathcal{C}(A, B) \in \mathcal{V}$ (called *hom-object*);
- for any three objects $A, B, C \in \mathcal{C}$ a morphism $\circ(A, B, C) : \mathcal{C}(B, C) \otimes \mathcal{C}(A, B) \to \mathcal{C}(A, C)$ in $\mathcal{V}$ (called *composition morphism*);
- for each object $A \in \mathcal{C}$ a morphism $j(A) : 1 \to \mathcal{C}(A, A)$ (called *identity*).

Furthermore, for all objects $A, B, C, D \in \mathcal{C}$, the following two diagrams must commute:

$$\begin{tikzcd}
(\mathcal{C}(C,D) \otimes \mathcal{C}(B,C)) \otimes \mathcal{C}(A,B) \arrow[rr, "\alpha"] \arrow[d, "\circ(B,C,D) \otimes \mathrm{id}_{\mathcal{C}(A,B)}"] & & \mathcal{C}(C,D) \otimes (\mathcal{C}(B,C) \otimes \mathcal{C}(A,B)) \arrow[d, "\mathrm{id}_{\mathcal{C}(C,D)} \otimes \circ(A,B,C)"'] \\
\mathcal{C}(B,D) \otimes \mathcal{C}(A,B) \arrow[r, "\circ(A,B,D)"'] & \mathcal{C}(A,D) & \mathcal{C}(C,D) \otimes \mathcal{C}(A,C), \arrow[l, "\circ(A,C,D)"]
\end{tikzcd}$$

$$\begin{tikzcd}
\mathcal{C}(B,B) \otimes \mathcal{C}(A,B) \arrow[r, "\circ(A,B,B)"] & \mathcal{C}(A,B) & \mathcal{C}(A,B) \otimes \mathcal{C}(A,A) \arrow[l, "\circ(A,A,B)"'] \\
1 \otimes \mathcal{C}(A,B) \arrow[u, "j(B) \otimes \mathrm{id}_{\mathcal{C}(A,B)}"] \arrow[ur, "\lambda(\mathcal{C}(A,B))"'] & & \mathcal{C}(A,B) \otimes 1. \arrow[u, "\mathrm{id}_{\mathcal{C}(A,B)} \otimes j(A)"'] \arrow[ul, "\rho(\mathcal{C}(A,B))"]
\end{tikzcd}$$

In the first diagram, α denotes the appropriate associator $\alpha(\mathcal{C}(C, D), \mathcal{C}(B, C), \mathcal{C}(A, B))$.

The diagrams are inspired by the axioms of an ordinary category: The first diagram expresses that composition in $\mathcal{C}$ is associative, while the second states that $j(A)$ acts as a unit.

Example 5.9. 1. A $(\mathcal{Set}, \times)$-enriched category just amounts to a locally small ordinary category.

2. A $(\mathcal{Cat}, \times)$-enriched category is a locally small strict 2-category. [Kel05, p.8].

Having defined enriched categories, it is only natural to next consider enriched functors.

Definition 5.10. Let $\mathcal{C}$ and $\mathcal{D}$ be two $(\mathcal{V}, \otimes, 1, \alpha, \lambda, \rho)$-enriched categories. A **$\mathcal{V}$-enriched functor** (or **$\mathcal{V}$-functor**) $F : \mathcal{C} \to \mathcal{D}$ consists of a function

$$F : \mathrm{Ob}(\mathcal{C}) \to \mathrm{Ob}(\mathcal{D})$$

and for each pair $A, B \in \mathrm{Ob}(\mathcal{C})$ a morphism

$$F(A, B) : \mathcal{C}(A, B) \to \mathcal{D}(F(A), F(B))$$

in $\mathcal{V}$, such that the diagrams

$$\begin{tikzcd}
\mathcal{C}(B,C) \otimes \mathcal{C}(A,B) \arrow[r, "\circ(A,B,C)"] \arrow[d, "F(B,C)\otimes F(A,B)"] & \mathcal{C}(A,C) \arrow[d, "F(A,C)"] \\
\mathcal{D}(F(B),F(C)) \otimes \mathcal{D}(F(A),F(B)) \arrow[r, "\circ(F(A),F(B),F(C))"] & \mathcal{D}(F(A),F(C)),
\end{tikzcd}$$

$$\begin{tikzcd}
1 \arrow[d, "j(A)"'] \arrow[dr, "j(F(A))"] & \\
\mathcal{C}(A,A) \arrow[r, "F(A,A)"] & \mathcal{D}(F(A),F(A))
\end{tikzcd}$$

commute.

The commutativity of the two diagrams means that F preserves composition and units.

Example 5.11. 1. The $(\mathcal{Set}, \times)$-enriched functors are just ordinary functors.

2. Any $(\mathcal{Cat}, \times)$-enriched functor constitutes a 2-functor.

The following lemma shows that symmetric monoidal closed categories are always self-enriched.

Lemma 5.12 ([Kel05, p.15])**.** Let $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ be a symmetric monoidal closed category. Then a $\mathcal{C}$-enriched category can be constructed as follows:

- The objects are those of $\mathcal{C}$.
- The hom-object of two objects $A, B \in \mathcal{C}$ is given by $[A, B]$.
- For three objects $A, B, C \in \mathcal{C}$, the composition morphism corresponds to the composition

  $$([B,C] \otimes [A,B]) \otimes A \xrightarrow{\alpha} [B,C] \otimes ([A,B] \otimes A) \xrightarrow{\mathrm{id}\otimes\mathrm{eval}} [B,C] \otimes B \xrightarrow{\mathrm{eval}} C$$

  under the tensor-hom adjunction.
- For an object $A \in \mathcal{C}$, the identity corresponds to $\lambda(A) : 1 \otimes A \to A$ under the tensor-hom adjunction.

Proof. It is left to the reader to check that the necessary diagrams commute. □

The reason for our interest in enriched category theory lies in the following theorem. It is a special case of [Koc72, Thm. 1.3] and adapted from [NLab, 5].

Theorem 5.13. Let $(\mathcal{C}, \otimes, 1, \alpha, \lambda, \rho)$ be a symmetric monoidal closed category. There exists a bijection between strong functors $\mathcal{C} \to \mathcal{C}$ and $\mathcal{C}$-enriched functors $\mathcal{C} \to \mathcal{C}$.

Proof. A $\mathcal{C}$-enriched endofunctor $\mathcal{C} \to \mathcal{C}$ corresponds to a natural transformation

$$[A, B] \to [T(A), T(B)]$$

making the diagrams in the definition of a $\mathcal{V}$-enriched functor commute. By the tensor-hom adjunction, such a map is equivalent to

$$t'(A, B) : [A, B] \otimes T(A) \to T(B).$$

In the following two diagrams, we omit the arguments for the natural transformations. If (T, t) is a strong endofunctor $\mathcal{C} \to \mathcal{C}$, then we can define $t'(A, B)$ to be

$$[A, B] \otimes T(A) \xrightarrow{\quad t \quad} T([A, B] \otimes A) \xrightarrow{T(\text{eval})} T(B).$$

On the other hand, a natural transformation (extranatural in A) $t : [A, B] \otimes T(A) \to T(B)$, induces a strength via

$$A \otimes T(B) \xrightarrow{\eta \otimes \mathrm{id}_{T(B)}} [B, A \otimes B] \otimes T(B) \xrightarrow{\quad t' \quad} T(A \otimes B),$$

where η denotes the unit of the tensor-hom adjunction.
It is not hard to check that these two constructions are indeed inverses to one and another. □

Since every strong monad is in particular a strong endofunctor, it is not surprising that the previous theorem can be improved to show a bijection between strong monads and $\mathcal{C}$-enriched monads. In particular, this holds if $\mathcal{C}$ is a cartesian closed category by 5.5.
In mathematical notation, `(>>=)` is a function $T(A) \to (A \to T(B)) \to T(B)$. By currying, this is equivalent to a function $\phi(A, B) : (A \to T(B)) \to (T(A) \to T(B))$. This looks very similar to the extension operation $-^T$ of an extension system, which assigns to every morphism $A \to T(B)$ a morphism $T(A) \to T(B)$. Motivated by this, we make the following ad-hoc definition:

Definition 5.14 (sketch). A **strong extension system** (T, η, ϕ) consists of the following:

- For every object $A \in \mathcal{C}$, there is an object $T(A) \in \mathcal{C}$ and a morphism $\eta(A) : A \to T(A)$.
- For all objects $A, B \in \mathcal{C}$, there exists a morphism $\phi(A, B) : [A, T(B)] \to [T(A), T(B)]$.

Furthermore, it must satisfy some suitable diagrams, similar to those of an extension system.

We can now describe the equivalence between strong monads and strong extension systems, which is in analogy to the equivalence between monads and extension systems from 5.2.

Let $\mathcal{C}$ be a cartesian closed category. Given a strong monad (T, μ, η, t) on $\mathcal{C}$, we define ϕ via the composition

$$[A, T(B)] \xrightarrow{\qquad} [T(A), T^2(B)] \xrightarrow{[\mathrm{id}_{T(A)}, \mu(B)]} [T(A), T(B)],$$

where the first arrow denotes the natural transformation given by the enrichment of T by 5.13.

On the other hand, a strong extension system (T, η, ϕ) gives rise to the following enrichment for T:

$$[A, B] \xrightarrow{[\mathrm{id}_A, \eta(B)]} [A, T(B)] \xrightarrow{\phi(A,B)} [T(A), T(B)].$$

Applying this to the cartesian closed category $\mathcal{H}ask$, we get a description of monads that closely resembles the type class `Monad` with its functions `return` and `>>=`. In particular, all monads represented by the `Monad` type class are strong.

References

[AHS04] Jiří Adámek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories. The Joy of Cats*. Citeseer, 2004.

[Awo10] Steve Awodey. *Category Theory*. 2nd ed. Oxford Logic Guides 52. Oxford ; New York: Oxford University Press, 2010. ISBN: 978-0-19-958736-0 978-0-19-923718-0.

[BJK05] Francis Borceux, George Janelidze, and Gregory Maxwell Kelly. "Internal Object Actions". In: *Commentationes Mathematicae Universitatis Carolinae* 46.2 (2005), pp. 235–255.

[BJT97] Hans-Joachim Baues, Mamuka Jibladze, and Andy Tonks. "Cohomology of Monoids in Monoidal Categories". In: *Contemporary Mathematics* 202 (1997), pp. 137–166.

[Bra14] Martin Brandenburg. "Tensor Categorical Foundations of Algebraic Geometry". In: *arXiv:1410.1716 [math]* (Oct. 2014). arXiv: `1410.1716 [math]`.

[Bra17] Martin Brandenburg. *Einführung in die Kategorientheorie: Mit ausführlichen Erklärungen und zahlreichen Beispielen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. ISBN: 978-3-662-53520-2 978-3-662-53521-9. URL: `http://link.springer.com/10.1007/978-3-662-53521-9` (visited on 10/01/2020).

[BW00] Michael Barr and Charles Wells. *Toposes, Triples, and Theories*. Springer-Verlag, 2000.

[CJ11] Dion Coumans and Bart Jacobs. "Scalars, Monads, and Categories". In: *arXiv:1003.0585 [math]* (Nov. 2011). arXiv: `1003.0585 [math]`.

[EK66] Samuel Eilenberg and G. Max Kelly. "Closed Categories". In: *Proceedings of the Conference on Categorical Algebra*. Ed. by S. Eilenberg et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1966, pp. 421–562. ISBN: 978-3-642-99904-8 978-3-642-99902-4. URL: `http://link.springer.com/10.1007/978-3-642-99902-4_22` (visited on 10/01/2020).

[EM01] M. Mehdi Ebrahimi and Mojgan Mahmoudi. "The Category of M-Sets". In: *Italian journal of pure and applied mathematics* (2001), pp. 123–132.

[EM65] Samuel Eilenberg and John C. Moore. "Adjoint Functors and Triples". In: *Illinois Journal of Mathematics* 9.3 (1965), pp. 381–398.

[GS90] Carl A. Gunter and Dana S. Scott. "Semantic Domains". In: *Formal Models and Semantics*. Elsevier, 1990, pp. 633–674.

[HaskellWiki] HaskellWiki. *Hask - HaskellWiki*. URL: `https://wiki.haskell.org/index.php?title=Hask%5C&oldid=52908` (visited on 10/07/2020).

[HF92] Paul Hudak and Joseph H. Fasel. "A Gentle Introduction to Haskell". In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–52.

[Hub61] Peter J. Huber. “Homotopy Theory in General Categories”. In: *Mathematische Annalen* 144.5 (Oct. 1961), pp. 361–385. ISSN: 0025-5831, 1432-1807. URL: `http://link.springer.com/10.1007/BF01396534` (visited on 10/01/2020).

[Hud+07] Paul Hudak et al. “A History of Haskell: Being Lazy with Class”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. 2007, pp. 1–55.

[Hud89] Paul Hudak. “Conception, Evolution, and Application of Functional Programming Languages”. In: *ACM Computing Surveys* 21.3 (Sept. 1989), pp. 359–411. ISSN: 0360-0300, 1557-7341. URL: `https://dl.acm.org/doi/10.1145/72551.72554` (visited on 10/01/2020).

[Hug89] John Hughes. “Why Functional Programming Matters”. In: *The computer journal* 32.2 (1989), pp. 98–107.

[Jon95] Mark P. Jones. “Functional Programming with Overloading and Higher-Order Polymorphism”. In: *Advanced Functional Programming*. Ed. by Gerhard Goos et al. Vol. 925. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 97–136. ISBN: 978-3-540-59451-2 978-3-540-49270-2. URL: `http://link.springer.com/10.1007/3-540-59451-5_4` (visited on 10/01/2020).

[Kel05] G. M. Kelly. “Basic Concepts of Enriched Category Theory”. In: *Reprints in Theory and Applications of Categories* 10 (2005), pp. vi+137. URL: `https://mathscinet.ams.org/mathscinet-getitem?mr=2177301` (visited on 10/13/2020).

[Kle65] Heinrich Kleisli. “Every Standard Construction Is Induced by a Pair of Adjoint Functors”. In: *Proceedings of the American Mathematical Society* 16.3 (1965), pp. 544–546.

[Kna06] Anthony W. Knapp. *Basic Algebra*. Cornerstones. Boston, MA: Birkhäuser Boston, 2006. ISBN: 978-0-8176-3248-9 978-0-8176-4529-8. URL: `http://link.springer.com/10.1007/978-0-8176-4529-8` (visited on 09/28/2020).

[Koc72] Anders Kock. “Strong Functors and Monoidal Monads”. In: *Archiv der Mathematik* 23.1 (1972), pp. 113–120.

[Lan02] Serge Lang. *Algebra*. Ed. by S. Axler, F. W. Gehring, and K. A. Ribet. Vol. 211. Graduate Texts in Mathematics. New York, NY: Springer New York, 2002. ISBN: 978-1-4612-6551-1 978-1-4613-0041-0. URL: `http://link.springer.com/10.1007/978-1-4613-0041-0` (visited on 10/01/2020).

[Law06] F. William Lawvere. “Adjointness in Foundations with the Author’s Commentary”. In: *Reprints in Theory and Applications of Categories* 16 (2006), pp. 1–16.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* no starch press, 2011. ISBN: 978-1-59327-283-8.

[LS88] Joachim Lambek and Philip J. Scott. *Introduction to Higher-Order Categorical Logic.* Vol. 7. Cambridge University Press, 1988.

[Mac63] Saunders Mac Lane. “Natural Associativity and Commutativity”. In: *Rice Institute Pamphlet-Rice University Studies* 49.4 (1963).

[Mac65] Saunders MacLane. “Categorical Algebra”. In: *Bulletin of the American Mathematical Society* 71.1 (1965), pp. 40–106.

[Mac98] Saunders Mac Lane. *Categories for the Working Mathematician.* Vol. 5. Springer Science & Business Media, 1998.

[Man76] Ernest G. Manes. *Algebraic Theories.* Ed. by P. R. Halmos. Vol. 26. Graduate Texts in Mathematics. New York, NY: Springer New York, 1976. ISBN: 978-1-4612-9862-5 978-1-4612-9860-1. URL: `http://link.springer.com/10.1007/978-1-4612-9860-1` (visited on 10/08/2020).

[Mar10] Simon Marlow. *Haskell 2010 Language Report.* Tech. rep. 2010. URL: `https://www.haskell.org/onlinereport/haskell2010` (visited on 10/01/2020).

[Mog88] Eugenio Moggi. *Computational Lambda-Calculus and Monads.* Tech. rep. University of Edinburgh, Department of Computer Science, 1988.

[Mog91] Eugenio Moggi. “Notions of Computation and Monads”. In: *Information and computation* 93.1 (1991), pp. 55–92.

[MP08] Conor McBride and Ross Paterson. “Applicative Programming with Effects”. In: *Journal of functional programming* 18.1 (2008), pp. 1–13.

[NLab] *Strong Monad - nLab.* URL: `https://ncatlab.org/nlab/show/strong+monad` (visited on 10/13/2020).

[Per20] Paolo Perrone. “Notes on Category Theory with Examples from Basic Mathematics”. In: *arXiv:1912.10642 [cs, math]* (Aug. 2020). arXiv: `1912.10642 [cs, math]`.

[Rie17] Emily Riehl. *Category Theory in Context.* Courier Dover Publications, 2017.

[RJ14] Exequiel Rivas and Mauro Jaskelioff. “Notions of Computation as Monoids”. In: *arXiv:1406.4823 [cs, math]* (May 2014). arXiv: `1406.4823 [cs, math]`.

[Sea13] Gavin J. Seal. “Tensors, Monads and Actions”. In: *arXiv:1205.0101 [math]* (June 2013). arXiv: `1205.0101 [math]`.

[Wad92a] Philip Wadler. “Comprehending Monads”. In: *Mathematical Structures in Computer Science* 2 (1992), pp. 461–493.

[Wad92b] Philip Wadler. “The Essence of Functional Programming”. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 1992, pp. 1–14.

[WB89] Philip Wadler and Stephen Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 1989, pp. 60–76.

[Wikibook] Wikibooks. *Haskell/Category Theory - Wikibooks, The Free Textbook Project.* URL: `https://en.wikibooks.org/w/index.php?title=Haskell/Category_theory%5C&oldid=3580981` (visited on 10/01/2020).