A review of Cartesian closed categories, string diagrams, monads, and optics.

Jad Issa

Abstract

Contents

1 Basics of category theory **3**

1.1 Categories and functors 3
1.2 Natural transformations 4
1.3 Duality 4
1.4 Adjunctions 5
1.4.1 Definition through hom-functors 5
1.4.2 Definition through units and counits 5
1.4.3 Adjunctions in 2-categories 8
1.5 Representable functors 8
1.6 Yoneda's lemma 9
1.7 Cones, cocones, limits, colimits 10
1.8 Special limits and colimits 10
1.9 General construction of limits and colimits 11
1.10 Interaction with adjunctions 12

2 Monoidal categories **13**

3 Cartesian closed categories and their endofunctors **14**

3.1 Lambda calculus 15
3.2 Category of endofunctors, monads 16
3.3 Kleisli category 17
3.3.1 Kleisli adjuction 17
3.3.2 Kleisli construction of a monad out of an adjunction 18
3.3.3 Application on the free/forgetful monoid adjunction 19
3.4 Properties of monads over Cartesian closed categories 19

4 String diagrams **20**

4.1 String diagrammatic representation of monoidal laws 22
4.2 More types of string diagrams 23

5 Wedges and ends, cowedges and coends. **24**

6 Optics **25**
6.1 Lenses . . . 25
6.2 Prisms . . . 26
6.3 Optics . . . 26
6.4 Actions of categories, or actegories . . . 26
6.5 Mixed optics . . . 27

A List of interesting monads in computer science **27**
A.1 Reader . . . 27
A.2 Writer . . . 27
A.3 IO . . . 28
A.4 Continuation . . . 28
A.5 Parser . . . 29

1 Basics of category theory

Category theory allows one to take a general view of mathematics wholly, providing uniform definitions for a wide variety of concepts in different mathematical fields, and proving theorems abstract enough to be applied to a large section of mathematical theories while being powerful enough to be significant in most of these theories. The basic power of category theory comes from the focus on how mathematical objects interact with other objects of a similar kind (groups with groups, topological spaces with topological spaces) instead of focusing on the details of the implementation of these objects into set theory.

1.1 Categories and functors

Definition 1.1 (Category)**.** A category is a collection of objects and arrows or morphisms that point from one object to another as well as a composition operator which composes two arrows $f : A \to B$ and $g : B \to C$ into $g \circ f : A \to C$ with some structural properties:

i. Composition is associative.

ii. Every object A admits an arrow id_A which functions as an identity for composition.

A category generally cannot be fit into sets, hence the use of 'collection' in the definition. Some categories have collections of objects or arrows that are proper classes. The categories for which the collection of arrows between any two objects is a set are called **locally small** in this case, the set of morphisms between any two objects A and B is called a hom-set and is written as $\hom_{\mathscr{C}}(A, B)$ omitting $\mathscr{C}$ when it is obvious. Categories where the collection of all arrows (and thus of all objects since any object admits at least one distinct endomorphism) is a set is called a **small** category. One example of a category which is not small, but is locally small, is the category **Set** of sets whose objects are sets and whose arrows are functions between sets, with their usual composition. The collection of functions between two sets is indeed a set itself, but the collection of all sets is a proper class.

Definition 1.2 (Functor)**.** If $\mathscr{C}$ and $\mathscr{D}$ are two categories, a functor $F : \mathscr{C} \to \mathscr{D}$ is a map from the objects of $\mathscr{C}$ into the objects of $\mathscr{D}$ and from the morphisms of $\mathscr{C}$ into the morphisms of $\mathscr{D}$ such that a morphism $f : A \to B$ from A to B in $\mathscr{C}$ is mapped to a morphism $Ff : FA \to FB$ from the image of A to the image of B.

Functors are supposed to satisfy some intuitive coherence conditions with the composition structure of the categories as well:

i. For an object A of $\mathscr{C}$, $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$.

ii. For arrows $f : A \to B$ and $g : B \to C$ in $\mathscr{C}$, $F(gf) = F(g)F(f)$.

The concept of a category, that is, objects with morphisms between then satisfying these basic properties generalizes the concepts of homomorphisms between many kinds of mathematical objects, but can be even more general.

For example, morphisms of sets are best considered as simple functions, giving us the category **Set**. On the other hand, while one can make from groups a category with morphisms being any function, it is not very useful, and the more intuitive approach would be to restrict morphisms in the category to morphism (homomorphisms) of groups. This can be done because the identity function is a homomorphism and the composition of two homomorphisms is also a homomorphism. We obtain then the category **Grp** of groups.

Two notable examples of functors can be given between **Set** and **Grp**.

First, there is a trivial functor U : **Grp** $\to$ **Set** called a forgetful functor which 'forgets' that the objects of **Grp** are groups and maps a group $(G, \cdot)$ to its underlying set G. The

homomorphisms of groups are mapped to themselves as functions also 'forgetting' that they respected any group structure. While this functor seems useless, it is related to another much more important functor, $F : \mathbf{Set} \to \mathbf{Grp}$, the free group functor which maps a set S to the free group over S and maps any function from S to another set R into a homomorphism between the corresponding free groups, this is done by 'dismantling' formal products and applying the function on each of the factors of the product separately.

Sometimes, morphisms are not functions at all. For example, for an preorder R on a set X, we can define a category with the elements of X as objects and morphisms representing the $\leq$ relation. Such categories are called poset categories and have certain special properties that are inherited from the order. For example, hom-sets are at most a singletons.

As we can see, functors are morphisms of categories. Hence, we can define the category **Cat** of all *small* (to avoid Russel's paradox!) categories with objects being small categories and arrows being functors between them.

1.2 Natural transformations

There is also a concept of homomorphisms between functors: a natural transformation. Functors from a fixed category $\mathscr{C}$ to fixed category $\mathscr{D}$ along with natural transformations between them form yet another category, the category of functors from $\mathscr{C}$ to $\mathscr{D}$. Such categories will be important later on, especially in the case where $\mathscr{C} = \mathscr{D}$ and we are dealing with the category of **endofunctors** of $\mathscr{C}$.

Definition 1.3 (Natural transformation)**.** If $\mathscr{C}, \mathscr{D}$ are categories and $F, G : \mathscr{C} \to \mathscr{D}$ are functors between them, a natural transformation $\eta : F \to G$ is a family of morphisms of $\mathscr{D}$. For each object A of $\mathscr{C}$, η has a 'component' η_A which is a morphism in $\mathscr{D}$ from $F(A)$ to $G(A)$.

We say the transformation is natural because it satisfies a 'naturality' property. Naturality is best represented by 'diagrams' which are said to 'commute', meaning that all possible paths in the diagram are equal. In this case, for a morphism $f : A \to B$ in $\mathscr{C}$, η is required to make the following diagram commute.

$$\begin{array}{ccc} F(A) & \xrightarrow{\eta_A} & G(A) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(B) & \xrightarrow[\eta_B]{} & G(B) \end{array}$$

This condition should not be surprising: just as group morphisms are expected to preserve the group structure, functor morphisms are expected to preserve the functorial structure, that is, how the functor acts on morphisms of the category $\mathscr{C}$.

Invertible natural transformation are called natural isomorphism and they play a big role in the parts to come.

1.3 Duality

Definitions, theorems, and proofs in category theory never 'look inside' of objects meaning that we never talk about what the objects and the morphisms actually represent. All those definitions, theorems, and proofs remain valid if we simply flip all the arrows in the definitions, theorems, and proofs.

Here's an example of a dual definition:

Definition 1.4 (Initial object)**.** An object A in a category $\mathscr{C}$ is initial if for any object $B \in \mathscr{C}$, there is a unique morphism from A to B.

Definition 1.5 (Terminal object)**.** An object A in a category $\mathscr{C}$ is terminal if for any object $B \in \mathscr{C}$, there is a unique morphism from B to A.

Often, dual definitions are denoted by prefixing 'co-' to the name, so a terminal object can also be called a co-initial object.

Terminal objects are necessarily unique up to isomorphism, that is, up to invertible morphisms. Indeed, if T and T' are two terminal objects, then by the terminality of T, there is a unique map f from T' to T, and by the terminality of T', there is a unique map g from T to T'. Those maps can then be composed to form endomorphisms $g \circ f$ of T and $f \circ g$ of T'. However, by the fact that both T and T' are terminal, the only endomorphisms possible for T and T' are the identity endomorphisms. Hence, $f \circ g = \mathrm{id}_{T'}$ and $g \circ f = \mathrm{id}_T$, and T and T' are isomorphic.

The fact that an arrow which is an isomorphism remains an isomorphism if we flip it reflects the fact that the dual of a terminal object, that is, an initial object, is also unique up to isomorphism.

1.4 Adjunctions

1.4.1 Definition through hom-functors

Often, functors $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ are almost inverses, or in some sense, inverses in 'spirit'. For example, if $\mathscr{C}$ is the category **Grp** of groups and $\mathscr{D}$ is the category **Set** of sets. There is a functor $U : \mathbf{Grp} \to \mathbf{Set}$ called the forgetful functor which maps a group G to the underlying set of G and a group homomorphism f to itself seen only as a function of the underlying sets. There is another functor $F : \mathbf{Set} \to \mathbf{Grp}$ called the free functor which maps a set X to the free group on X. While F and U are clearly not inverses (for example $U(F(X))$ is bigger than X and if G isn't a free group, $F(U(G))$ is not even isomorphic to G), there is a sense in which they do inverse operations. One makes of a group the most natural related set, and the other makes of a set the most naturally related group. The formal way to describe this relation is through *adjunctions* of functors.

Definition 1.6 (Adjoint functors)**.** Given categories $\mathscr{C}$ and $\mathscr{D}$ and functors $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$, we say that F is *left-adjoint* to G (and G *right-adjoint* to F) when there is an natural isomorphism between the functors $\mathrm{Hom}(F(-), -)$ and $\mathrm{Hom}(-, G(-))$ as functors from $\mathscr{C} \times \mathscr{D}$ to **Set** (for $\mathscr{C}$ and $\mathscr{D}$ locally small). An adjunction is denote $F \dashv G$.

It's common to see this written as, for $A \in \mathscr{C}$ and $B \in \mathscr{D}$, there is a bijection which is natural in A and B.

$$\mathrm{Hom}(F(A), B) \cong \mathrm{Hom}(A, G(B)).$$

1.4.2 Definition through units and counits

Another way to see adjunctions which puts the focus on the idea that the functors involved are almost semi-inverses is the following.

Definition 1.7 (Adjoint functors)**.** Given categories $\mathscr{C}$ and $\mathscr{D}$ and functors $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$, $F \dashv G$ if there are natural transformations:

$$\eta : \mathrm{id}_{\mathscr{C}} \to GF$$
$$\varepsilon : FG \to \mathrm{id}_{\mathscr{D}}$$

that satisfy the triangle laws:

$$\begin{array}{ccc} F & \xrightarrow{F\eta} & FGF \\ & \overset{\mathrm{id}_F}{\searrow\!\!\searrow} & \downarrow{\scriptstyle \varepsilon F} \\ & & F \end{array} \qquad \begin{array}{ccc} G & \xrightarrow{\eta G} & GFG \\ & \overset{\mathrm{id}_G}{\searrow\!\!\searrow} & \downarrow{\scriptstyle G\varepsilon} \\ & & G \end{array}$$

Under these notations, η is called the unit of the adjunction and ε the counit.

If F and G were actually inverses, then η and ε would be isomorphisms as well. The unit/counit definition will generally be the preferred definition in this text, but before we use it, we need to show that these two definitions are indeed equivalent.

Proof. Assuming first the hom-functor definition, let $\phi_{A,B} : \mathrm{Hom}(F(A), B) \to \mathrm{Hom}(A, G(B))$ be the given natural isomorphism. We define $\eta_A = \phi_{A,F(A)}(\mathrm{id}_{F(A)})$ and we prove that η is natural.

In fact, the naturality of ϕ gives the following commutative diagram for $f : A \to A'$ in $\mathscr{C}$.

$$\begin{array}{ccccc}
\mathrm{Hom}(F(A), F(A)) & \xrightarrow{(Ff\circ -)} & \mathrm{Hom}(F(A), F(A')) & \xleftarrow{(-\circ Ff)} & \mathrm{Hom}(F(A'), F(A')) \\
\downarrow{\scriptstyle \phi_{A,F(A)}} & & \downarrow{\scriptstyle \phi_{A,F(A')}} & & \downarrow{\scriptstyle \phi_{A',F(A')}} \\
\mathrm{Hom}(A, GF(A)) & \xrightarrow[(GFf\circ -)]{} & \mathrm{Hom}(A, GF(A')) & \xleftarrow[(-\circ f)]{} & \mathrm{Hom}(A', GF(A'))
\end{array}$$

Notice that in this diagram, $Ff \circ \mathrm{id}_A = Ff = \mathrm{id}_{A'} \circ Ff$. Therefore, $\phi_{A',F(A')}(\mathrm{id}_{F(A')}) \circ f = GF(f) \circ \phi_{A,F(A)}(\mathrm{id}_{F(A)})$. In other words, $\eta_{A'} \circ f = GF(f) \circ \eta_A$, which is the naturality condition for η.

$$\begin{array}{ccc}
A & \xrightarrow{f} & B \\
\downarrow{\scriptstyle \eta_A} & & \downarrow{\scriptstyle \eta_B} \\
GF(A) & \xrightarrow[GF(f)]{} & GF(B)
\end{array}$$

Similarity, define $\varepsilon_B = \phi^{-1}_{G(B),B}(\mathrm{id}_{G(B)})$ and prove its naturality with the following diagrams.

$$\begin{array}{ccccc}
\mathrm{Hom}(G(B), G(B)) & \xrightarrow{(Gf\circ -)} & \mathrm{Hom}(G(B), G(B')) & \xleftarrow{(-\circ Gf)} & \mathrm{Hom}(G(B'), G(B')) \\
\downarrow{\scriptstyle \phi^{-1}_{G(B),B}} & & \downarrow{\scriptstyle \phi^{-1}_{G(B'),B}} & & \downarrow{\scriptstyle \phi^{-1}_{G(B'),B'}} \\
\mathrm{Hom}(FG(B), B) & \xrightarrow[(f\circ -)]{} & \mathrm{Hom}(FG(B), B') & \xleftarrow[(-\circ FGf)]{} & \mathrm{Hom}(FG(B'), B')
\end{array}$$

$$\begin{array}{ccc}
FG(A) & \xrightarrow{FG(f)} & FG(B) \\
\downarrow{\scriptstyle \varepsilon_A} & & \downarrow{\scriptstyle \varepsilon_B} \\
A & \xrightarrow[f]{} & B
\end{array}$$

We still need to show the triangle laws which we will leave to prove during our proof of the converse

Conversely, suppose we have the unit and counit satisfying the triangle laws. Define a function $\phi_{A,B} : \mathrm{Hom}(F(A), B) \to \mathrm{Hom}(A, G(B))$ by $\phi(f) = G(f) \circ \eta_A$ and $\psi_{A,B} : \mathrm{Hom}(A, G(B)) \to \mathrm{Hom}(F(A), B)$ by $\psi(g) = \varepsilon_B \circ F(g)$

$$\begin{aligned}
\psi_{A,B}(\phi_{A,B}(f)) &= \varepsilon_B \circ F(G(f) \circ \eta_A) && (1)\\
&= \varepsilon_B \circ FG(f) \circ F(\eta_A) && (2)\\
&= f \circ \varepsilon_{F(A)} \circ F(\eta_A) && (3)\\
&= f && (4)
\end{aligned}$$

$$
\begin{aligned}
\phi_{A,B}(\psi_{A,B}(g)) &= G(\varepsilon_B \circ F(g)) \circ \eta_A && (5)\\
&= G(\varepsilon_B) \circ GF(g) \circ \eta_A && (6)\\
&= G(\varepsilon_B) \circ \eta_{F(B)} \circ g && (7)\\
&= g && (8)
\end{aligned}
$$

Steps 1 and 5 follow from the formulae given for ϕ and ψ, the steps 2 and 6 follow from the functoriality of F and G respectively, and the steps 3 and 7 follow from the naturality of ε and η respectively. Lastly, given the formulae, the functoriality of F and G and the naturality of ε and η, the calculations show that ψ and ϕ are inverses if and only if the triangle laws are satisfied (steps 4 and 8).

The direction unit/counit implies hom-functor has been proven, and what remains for the direction hom-functor implies unit/counit is to show the formulae for ϕ and $\psi = \phi^{-1}$.

In fact, those follow from the commutativity of the following two diagrams at the identities $\mathrm{id}_{F(A)}$ and $\mathrm{id}_{G(B)}$ respectively.

$$
\begin{array}{ccc}
\mathrm{Hom}(F(A), F(A)) & \xrightarrow{f\circ -} & \mathrm{Hom}(F(A), B)\\
\downarrow{\scriptstyle \phi_{A,F(A)}} & & \downarrow{\scriptstyle \phi_{A,B}}\\
\mathrm{Hom}(A, GF(A)) & \xrightarrow[G(f)\circ -]{} & \mathrm{Hom}(A, G(B))
\end{array}
\qquad
\begin{array}{ccc}
\mathrm{Hom}(G(B), G(B)) & \xrightarrow{-\circ g} & \mathrm{Hom}(A, G(B))\\
\downarrow{\scriptstyle \psi_{G(B),B}} & & \downarrow{\scriptstyle \psi_{A,B}}\\
\mathrm{Hom}(FG(B), B) & \xrightarrow[-\circ F(g)]{} & \mathrm{Hom}(F(A), B)
\end{array}
$$

$\square$

Example 1.1 (Free/forgetful adjunctions)**.** Some of the most common (and simple) adjunctions are the free/forgetful adjunctions for a given algebraic structure. It's also a good example of how adjoints of simple or even trivial functors can be much more complex and useful. We will look at the example of the forgetful functor $U : \mathbf{Grp} \to \mathbf{Set}$ which maps a group G to the underlying set of G (that is, forgets the group structure of G). A morphism is **Grp**, that is, a group homomorphism, is mapped by U to itself seen as a function of sets.

The forgetful functor admits a left adjoint $F : \mathbf{Set} \to \mathbf{Grp}$ which is actually the free functor mapping a set X to the free group $F(X)$ on X given by formal words in the alphabet of $\{x, x^{-1} : x \in X\}$ quotiented by the relation $xx^{-1} \sim 1$. Any function $f : X \to Y$ induces a unique homomorphism $Ff : FX \to FY$ extending f defined at the generating set X of FX. $Ff : y_1 \cdots y_n \to \tilde{f}(y_1) \cdots \tilde{f}(y_n)$ where $\forall i, y_i\{x, x^{-1} : x \in X\}$ and if $x \in X$,

$$
\begin{aligned}
\tilde{f} : x &\to f(x)\\
x^{-1} &\to f(x)^{-1}
\end{aligned}
$$

We will prove that $F \dashv U$ using the unit/counit definition as this provides more insight into the adjunction by constructing two natural transformations η, ε.

$$
\begin{aligned}
\eta : X &\to UF\\
x &\to x \in F(X) \text{ as a set}\\
\varepsilon : FU(G) &\to G)\\
g_1^{\pm_1 1} \cdots g_n^{\pm_n 1} &\to g_1^{\pm_1 1} \cdots g_n^{\pm_n 1} \text{evaluated in } G
\end{aligned}
$$

We verify the triangle laws where we denote, for clarity, the formal singleton word x by (x).

$$
\begin{array}{ccc}
U(G) & \xrightarrow{\eta_{U(G)}} & UFU(G) \\
& {}_{\mathrm{id}_{U(G)}}\diagdown\!\!\diagdown & \big\downarrow U\varepsilon_G \\
& & U(G)
\end{array}
\qquad
\begin{array}{ccc}
g & \xmapsto{\eta_{U(G)}} & (g) \\
& {}_{\mathrm{id}_{U(G)}}\diagdown\!\!\diagdown & \big\downarrow U\varepsilon_G \\
& & g
\end{array}
$$

$$
\begin{array}{ccc}
F(X) & \xrightarrow{F\eta_X} & FUF(X) \\
& {}_{\mathrm{id}_{F(X)}}\diagdown\!\!\diagdown & \big\downarrow \varepsilon_{F(X)} \\
& & F(X)
\end{array}
\qquad
\begin{array}{ccc}
x_1^{\pm_1 1}\cdots x_n^{\pm_n 1} & \xrightarrow{F\eta_X} & (x_1)^{\pm_1 1}\cdots (x_n)^{\pm_n 1} \\
& {}_{\mathrm{id}_{F(X)}}\diagdown\!\!\diagdown & \big\downarrow \varepsilon_{F(X)} \\
& & x_1^{\pm_1 1}\cdots x_n^{\pm_n 1}
\end{array}
$$

Both of these diagrams express the fact that the evaluation of (x) is x itself, but in two different 'axes'. We will see later that this expresses the left and right unit laws for the monad $UF : \mathbf{Set} \to \mathbf{Set}$.

Often, such 'forgetful' functors have left adjoints which are the 'free' functors for a given structure. Rarely, those forgetful functors can also have right adjoints. This is the case for topological spaces which has the discrete topology as left adjoint and the indiscrete topology as right adjoint.

1.4.3 Adjunctions in 2-categories

It turns out that the concept of adjunctions can be generalized to more than functors to more general morphisms in a category. The necessary condition for the generality to work is the possibility to form categories of morphisms like we do for functors; hence the definition below.

Definition 1.8 (2-category (simplified))**.** A 2-category $\mathscr{C}$ consists of a collection of objects, and for each pair of objects, $A, B \in \mathscr{C}$, there is a hom-category $\mathrm{Hom}(A, B)$ such that for each triplet A, B, C of objects, we have a composition functor $\circ : \mathrm{Hom}(B, C) \times \mathrm{Hom}(A, B) \to \mathrm{Hom}(A, C)$, and each category $\mathrm{Hom}(A, A)$ admits an identity object id_A such that the composition with this object on the right (resp. left) is the identity functor on $\mathrm{Hom}(A, B)$ (resp. $\mathrm{Hom}(B, A)$). The composition functor must also be associative.

Note that this definition is simplified and relatively inelegant, but a proper definition can only be given with an understanding of monoidal categories in mind as it involves the associator and unitors of the monoidal category **Cat**.

The idea is that in this category, there are not only morphisms between objects, but morphisms between morphisms as well, called 2-morphisms. Categories, functors, and natural transformations form a 2-category, and 2-categories are generalizations of that structure.

In the context of 2-categories, we can define adjoint 1-morphisms $F \dashv G$ where $F : C \to D$ as having a unit 2-morphism $\eta : \mathrm{id}_C \to GF$ and a counit 2-morphism $\varepsilon : FG \to \mathrm{id}_D$ satisfying the triangle laws. Then, adjoint functors are simply adjoint (1-)morphisms in the 2-category **Cat**.

1.5 Representable functors

Often, it may be difficult to work in general categories because morphisms can be tricky to pinpoint and the internal structure (if any) of objects is typically inaccessible. It is, however, easier if we can somehow see the category within **Set**. To make this reasonable, we will stick to locally small categories (that is, categories where the hom-functor is defined into **Set**.) It's often possible to define very many interesting functors from a locally small category $\mathscr{C}$ into **Set**. Sometimes the functors are covariant $F : \mathscr{C} \to \mathbf{Set}$, sometimes they are contra-variant $F : \mathscr{C}^{\mathrm{op}} \to \mathbf{Set}$; contravariant functors into **Set** are called presheaves. For example, one interesting presheaf on **Top** is the functor $F : \mathbf{Top}^{\mathrm{op}} \to \mathbf{Set}$, which, to a topological space

X, maps the underlying topology (the set of open sets of F) and to a continuous function $f : X \to Y$, maps Ff the function on open subsets of Y mapping V into its open pre-image $f^{-1}(V)$.

If $\mathscr{C}$ is locally small, there are two particular special kinds of covariant and contravariant functors that can be defined on $\mathscr{C}$, the hom-functors out of and into a fixed object. Let A be an object of $\mathscr{C}$. Then the (covariant) functor $h_A = \mathrm{Hom}(A, -)$ is **Set**-valued, and the contravariant functor $h^A = \mathrm{Hom}(-, A)$ is a presheaf. If a given **Set**-valued functor F is isomorphic (in the category of functors $\mathscr{C} \to \mathbf{Set}$) to h_A for some $A \in \mathscr{C}$, we say that F is **representable**, and represented by A and the natural isomorphism found $h_A \to F$. If F is contravariant, it is representable if it is isomorphic to h^A for some $A \in \mathscr{C}$.

1.6 Yoneda's lemma

Following the discussion in 1.5, we find a very important result: if $h^A \cong h^B$, then $A \cong B$. This encapsulates the idea that, if two objects appear the same ($h^A \cong h^B$) in a category, then they are the same ($A \cong B$), up to isomorphism.

The result is a corollary of a more general result: the Yoneda lemma.

Lemma 1.1 (Yoneda)**.** *Let $\mathscr{C}$ be a locally small category and $F : \mathscr{C} \to \mathbf{Set}$. Then for any $A \in \mathscr{C}$,*

$$\mathrm{Nat}(h_A, F) \cong F(A).$$

Proof. We will prove the dual.

Let $\eta \in \mathrm{Nat}(h^A, F)$, then the following naturality diagram commutes for any $f : A \to B$ in $\mathscr{C}$.

$$\begin{array}{ccc} \mathrm{Hom}(A, A) & \xrightarrow{(f\circ -)} & \mathrm{Hom}(A, B) \\ \downarrow{\scriptstyle \eta_A} & & \downarrow{\scriptstyle \eta_B} \\ F(A) & \xrightarrow{F(f)} & F(B) \end{array}$$

This diagram is in **Set**, so we are dealing with sets and functions between sets. By chasing the element $\mathrm{id}_A \in \mathrm{Hom}(A, A)$ in the diagram, we obtain the following.

$$\eta_B(f) = F(f)(\eta_A(\mathrm{id}_A)).$$

In other words, η is completely defined by its value at $\eta_A(\mathrm{id}_A)$, so that the function $\eta \mapsto \eta_A(\mathrm{id}_A)$ is injective. Furthermore, for any $a \in F(A)$, it's possible to define $\eta_A(f) = F(f)(a)$, and then define al of η using the formula above to obtain a natural transformation such that $\eta_A(\mathrm{id}_A) = a$. In other words, we also have surjectivity.

□

A dual proof gives $\mathrm{Nat}(h^A, F) \cong F(A)$ when F is contravariant.

Corollary 1.1. *If $h^A \cong h^B$, then $A \cong B$.*

Proof. We have $\mathrm{Nat}(h^A, h^B) \cong h^B(A) = \mathrm{Hom}(A, B)$, which makes h^- a covariant full and faithful functor. Then, if $\phi : h^A \to h^B$ is an isomorphism, then there exists $f : A \to B$ such that $\phi = F(f)$ and $g : B \to A$ such that $\phi^{-1} = F(g)$ by the fullness of F. But then, $F(gf) = F(g)F(f) = \mathrm{id}_{h^A}$. By the faithfulness of F, $gf = \mathrm{id}_A$. Similarity, $F(fg) = \mathrm{id}_{h^B} \implies fg = \mathrm{id}_B$, so $A \cong B$. □

Remark 1.1. Of course, by looking at $\mathscr{C}^{\mathrm{op}}$, which is also locally small, we can prove that $h_A \cong h_B \implies A \cong B$ as well.

1.7 Cones, cocones, limits, colimits

Definition 1.9 (Cone)**.** If $\mathscr{C}$ is a category and F a functor from another category $\mathscr{I}$ to $\mathscr{C}$, then a *cone* η from $A \in \mathscr{C}$ to F is a natural transformation from the constant functor $A : \mathscr{I} \to \mathscr{C}$ to F.

The functor F is often called a *diagram* and the category $\mathscr{I}$ is the *shape* of the diagram. Since a cone is a natural transformation from the constant functor A, it's equivalent to a family of morphisms η_α from A to objects $F(\alpha)$ for all objects $\alpha \in \mathscr{I}$ which satisfies a naturality principle: for $f : \alpha \to \beta$ in $\mathscr{I}$, the following diagram commutes:

$$\begin{array}{ccc} & A & \\ {\scriptstyle \eta_\alpha} \swarrow & & \searrow {\scriptstyle \eta_\beta} \\ F(\alpha) & \xrightarrow{Ff} & F(\beta) \end{array}$$

Therefore, a cone is characterized by its vertex (A) and the arrows from that vertex to each of the objects in the image of F. Cones into F form a category where objects are cones and morphisms between a cone of vertex A and morphisms η_α for $\alpha \in \mathscr{I}$ and a cone of vertex B and morphisms ν_α for $\alpha \in \mathscr{I}$ consists of a morphism f from A to B such that for each $\alpha \in \mathscr{I}$, the diagram below commutes.

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle \eta_\alpha} \searrow & & \swarrow {\scriptstyle \nu_\alpha} \\ & F(\alpha) & \end{array}$$

In other words, the cone on A factors through the cone on B.

Definition 1.10 (Limit)**.** A limit of a diagram $F : \mathscr{I} \to \mathscr{C}$ is a terminal object in the category of cones into F.

Being a terminal object, a limit is necessarily unique up to isomorphism. In fact, terminal objects are themselves limits of a diagram with no arrows. Similar notions of cocone and colimit follow by duality, and colimits are, just like limits, unique up to isomorphism.

1.8 Special limits and colimits

Certain special cases of limits and colimits have their own names because of how important they are independent of the more general notion of limits and colimits. We present a couple here.

Definition 1.11 (Discrete category)**.** For any set X, there is a category whose objects are elements of X and whose only morphisms are the identity morphisms for each element. This is the discrete category on X.

Definition 1.12 (Product)**.** If $\mathscr{C}$ is a category and $(A_i)_{i \in I}$ is a family of objects of $\mathscr{C}$ indexed by I, then the product $\Pi_{i \in I} A_i$, if it exists, is the limit of the diagram $i \mapsto A_i$ from the discrete category on I into $\mathscr{C}$. In other words, the product is a limit that disregards arrows.

Example 1.2 (Product group)**.** In **Grp**, the categorical product of G and H is the product group $G \times H$ with the usual multiplication operation.

Proof. We have projection group morphisms $\pi_1 : G \times H \to G$ and $\pi_2 : G \times H \to H$ and any pair of group morphisms $f : X \to G$ and $g : X \to H$ define a unique group morphism, namely, $f \times g : X \to G \times H$ through which f and g factor with the projections. $\square$

More generally, product algebraic structures correspond to categorical products.

Example 1.3 (Product topology)**.** Consider $\mathbb{N}$ copies of the real line $\mathbb{R}$. There are at least two distinct ways to define a topology on the infinite product $\mathbb{R}^{\mathbb{N}}$: the box topology which we will denote by $\tau_{\Box}$, and the product topology which we will denote by τ_{π}. The box topology is the topology with basis any product of open sets U_i. On the other hand, the product topology has basis products of open sets U_i where only finitely many of the U_i are not $\mathbb{R}$ itself. It is said, in most introductory topology textbooks, that the box topology 'simply doesn't behave nicely enough' and that the product topology behaves much better. The reason for that is that the product topology is the categorical product in **Top** while the box topology is just another cone, which is generally **not** the universal cone corresponding to the product.

In topology, this is expressed slightly differently: both the product and the box topologies make the projections continuous (are cones) but the product topology is the coarsest (universal) one for which projections are continuous. In the general case, the product topology is strictly coarser than the box topology, which means that the identity function taken from X with the product topology to X with the box topology, which is the only candidate for the unique map making the universality diagram for 'the box topology is the categorical product' commute, is actually not continuous, that is, does not exist in **Top**. Hence, the box topology cannot, in general, be the categorical product, and categorical properties implied by being the categorical product do not apply to the box topology, hence, 'it behaves badly.'

Definition 1.13 (Equalizer)**.** If $\mathscr{C}$ is a category and A, B are objects of $\mathscr{C}$ with morphisms $f, g \in \mathscr{C}(A, B)$, the equalizer of f and g is the limit of the diagram $A \underset{g}{\overset{f}{\rightrightarrows}} B$ in $\mathscr{C}$.

Example 1.4 (Kernel)**.** In **Grp**, if G, H are groups, and $f : G \to H$ is a group homomorphism, then the equalizer of f and the trivial homomorphism $1 : G \to H; g \mapsto 1_H$ is (isomorphic to) the kernel of f. In fact, $\operatorname{Ker} f$ and the injection into G is a cone commuting f and 1, since $f(k) = 1_H = 1(k)$ for all $k \in K$, and it is universal since if K', i is another cone, then $f(i(k')) = 1_H$ so that i factors through $\operatorname{Ker} f$. By the uniqueness of limits, any other group satisfying the limit conditions is isomorphic to $\operatorname{Ker} f$.

Notions of *Coproduct* and *Coequalizer* follow by duality, and they do agree with usual notions of coproduct (direct sum, disjoint union, LCM, etc...) and coequalizer (cokernel, for example).

1.9 General construction of limits and colimits

It turns out that finding products and equalizers is enough to find any limit.

Theorem 1.1 (Construction of limits from products and equalizers)**.** *If $\mathscr{C}$ is a category admitting all products and all equalizers, then $\mathscr{C}$ also admits all limits. Let $F : \mathscr{I} \to \mathscr{C}$ be a diagram, and consider the product $\prod_{i \in \mathscr{I}^o} F(i)$ of the objects of the diagram F with the corresponding projections $p_j : \prod_{i \in \mathscr{I}^o} F(i) \to F(j)$. Then, the limit of F is obtained as the equalizer of the following two maps.*

$$\prod_{i \in \mathscr{I}^o} F(i) \underset{\prod_{\substack{f:i\to j \\ i,j \in \mathscr{I}^o}} p_j}{\overset{\prod_{\substack{f:i\to j \\ i,j \in \mathscr{I}^o}} F(f)\circ p_i}{\rightrightarrows}} \prod_{\substack{f:i\to j \\ i,j \in \mathscr{I}^o}} F(j)$$

Example 1.5 (Pullback)**.** A pullback in a category $\mathscr{C}$ is the limit of a diagram of the form

$$\begin{array}{ccc} & & A \\ & & \downarrow f \\ B & \xrightarrow[g]{} & C \end{array}$$

Let's position ourselves in the category **Set** and where we know how to construct products (Cartesian products) and equalizers. The equalizer of $f : X \to Y$ and $g : X \to Y$ in **Set** is the subset of X defined by $\{x : x \in X, f(x) = g(x)\}$.

The pullback will be a set D with functions $p_1 : P \to A$ and $p_2 : P \to B$ making the following diagram commute.

$$\begin{array}{ccc} D & \xrightarrow{p_1} & A \\ {\scriptstyle p_2}\downarrow & & \downarrow{\scriptstyle f} \\ B & \xrightarrow[g]{} & C \end{array}$$

We begin by constructing the products mentioned in the theorem.

$$A \times B \times C \underset{(\pi_C \times \pi_C)}{\overset{(f \circ \pi_A) \times (g \circ \pi_B)}{\rightrightarrows}} C \times C$$

Next, we find the equalizer of these two maps. This will be the subset of $A \times B \times C$ with elements (a, b, c) with $f(a) = c$ and $g(b) = c$. In other words, it's isomorphic to the subset of $A \times B$ of pairs (a, b) with $f(a) = g(b)$.

1.10 Interaction with adjunctions

Limits and colimits provide yet another reason to care about adjunctions of functors: functors having certain adjoints preserve limits/colimits.

Theorem 1.2 (Continuity of right adjoints)**.** *If $\mathscr{C}, \mathscr{D}$ are categories with functors $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ that are adjoint $F \dashv G$. And if $L : \mathscr{I} \to \mathscr{D}$ has a limit $\lim L \in \mathscr{C}$, then GL has a limit and $G(\lim L) = \lim GL$.*

Theorem 1.3 (Cocontinuity of left adjoints)**.** *Dually, if $\mathscr{C}, \mathscr{D}$ are categories with functors $F : \mathscr{C} \to \mathscr{D}$ and $G : \mathscr{D} \to \mathscr{C}$ that are adjoint $F \dashv G$. And if $L : \mathscr{I} \to \mathscr{C}$ has a colimit $\operatorname{colim} L \in \mathscr{C}$, then FL has a colimit and $F(\operatorname{colim} L) = \operatorname{colim} FL$.*

Example 1.6 (Preservation of products for groups)**.** Since the forgetful functor $U : \mathbf{Grp} \to \mathbf{Set}$ is right adjoint to the free group functor $F : \mathbf{Grp} \to \mathbf{Set}$, it preserves limits. In particular, it preserves products meaning that if G and H are groups and K is their product, then the underlying set $U(K)$ of K is the product of the underlying sets of G and H. In other words, $K = G \times H$ as a set. The fact that free/forgetful adjunctions are so common for set-based algebraic structures explains, purely categorically, why so often the product algebraic structure is constructed from the Cartesian product of the underlying sets.

Example 1.7 ($U : \mathbf{Grp} \to \mathbf{Set}$ has no right adjoint)**.** Let G and H are groups, then the set $U(G) \sqcup U(H)$ which is the coproduct of $U(G)$ and $U(H)$ does not (generally) define a group with injective group homomorphisms from G and H. In fact, the coproduct of G and H is also the product $G \times H$. In the case of G the trivial group and $H = \mathbb{Z}_4$, $G \times H = G \oplus H$ is of order 4, but $U(G) \sqcup U(H)$ has cardinality 5. This means that U does not preserve colimits; therefore, it is not left adjoint to any functor (a.k.a. U does not admit a right adjoint).

Example 1.8 (Products and coproducts in **Top**)**.** The category **Top** of topological spaces and continuous maps admits a forgetful functor into **Set** which has both left and right adjoints. We expect that functor to then preserve both limits and colimits. This is indeed the case demonstrated by the following examples: if X, Y are topological spaces, the (categorical) product space of X and Y is indeed the Cartesian product $X \times Y$ endowed with the corresponding topology. The categorical coproduct of X and Y consists of the space that 'places X and Y

next to each other, disjoint', that is, it has underlying set the disjoint union (coproduct in **Set**) of X and Y and the topology is the topology of X and the topology of Y embedded as a clopen partition of the coproduct space.

The fact that the forgetful functor from topological spaces has both left and right adjoints explains the compatibility of many set operations with the topologies involved (e.g. the disjoint union of topological spaces is a topological space encompassing the spaces as subspaces, while the disjoint union of groups is not generally a group containing the original groups as subgroups.)

2 Monoidal categories

A monoidal category is a category with some form of extra 'monoid' structure on the objects. That is, there is a way to 'multiply' objects of the category which is associative, and this multiplication has some 'identity'. Monoidal categories are very important in computer science and we will divert our attention to them now.

The key to properly defining a monoidal category $\mathscr{C}$ is to properly define multiplication and identity. Multiplication should be a sort of binary map on the objects of a category. A good way to define that while respecting the categoric structure is to make multiplication is a bi-functor into $\mathscr{C}$, that is, a functor from the product category $\mathscr{C} \times \mathscr{C}$ into $\mathscr{C}$. More explicitly,

Definition 2.1 (Product category)**.** If $\mathscr{C}$ and $\mathscr{D}$ are categories, then there is a category $\mathscr{C} \times \mathscr{D}$ whose objects are pairs (c, d) of objects $c \in \mathscr{C}$ and $d \in \mathscr{D}$ and whose morphisms are pairs (f, g) of morphisms f in $\mathscr{C}$ and g in $\mathscr{D}$.

We will denote this multiplication as $\otimes$ and the identity for this multiplication will be an object of $\mathscr{C}$ which we will denote as 1.

Of course, this does not yet resemble a monoid as no laws are required to be respected yet. Naively, we could impose that $1 \otimes C = C \otimes 1 = C$ for all $C \in \mathscr{C}$ and that $A \otimes (B \otimes C) = (A \otimes B) \otimes C$ for all $A, B, C \in \mathscr{C}$. However, this is usually too strict since it's not even satisfied in the category of sets with the usual Cartesian product.[1] Such a structure will be called a **strict monoidal category**, but we will need to look at more general notions; we will have to replace equality with (natural) isomorphism. This naturally leads us to impose the following natural isomorphisms.

Left unitor We require the existence of a natural isomorphism $\lambda : (1 \otimes (-)) \to (-)$ called the left unitor of the monoidal category. To clarify the notation above, $(1 \otimes (-)) : \mathscr{C} \to \mathscr{C}$ is the functor which maps an object $A \in \mathscr{C}$ to $1 \otimes A$ and a morphism $f : A \to B$ to $\mathrm{id}_1 \otimes f : 1 \otimes A \to 1 \otimes B$, and $(-)$ is the identity endofunctor on $\mathscr{C}$.

Right unitor The same applies to multiplication on the right. We have a natural isomorphism: $\rho : ((-) \otimes 1) \to (-)$.

Associator Imposing associativity requires the existence of the *associator* natural isomorphism $\alpha : ((-) \otimes (-)) \otimes (-) \to (-) \otimes ((-) \otimes (-))$ with the intuitive order respected; in other words, α has for components isomorphisms $(A \otimes B) \otimes C \to A \otimes (B \otimes C)$.

We would like to be able to give a unique value to an expression of the form $A_1 \otimes A_2 \otimes \cdots \otimes A_n$. There are many ways to associate the multiplications, all of them are, of course, isomorphic; however, once we are dealing with more than 3 objects, there are many different ways to compose associators to obtain, a priori, distinct isomorphisms between the same two objects. Distinct isomorphisms between the same objects pose a problem because, for instance, if the objects are sets or set-like, we are unable to uniquely identify an element of one with an element of another,

[1] For example, $\mathbb{N} \times (\mathbb{N} \times \mathbb{N}) \neq (\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$ as, for example, $(0, (0, 0)) \in (\mathbb{N} \times (\mathbb{N} \times \mathbb{N}))$ but $(0, (0, 0)) \notin ((\mathbb{N} \times \mathbb{N}) \times \mathbb{N})$. The two sets are, however, indeed isomorphic by the isomorphism $(x, (y, z)) \mapsto ((x, y), z)$ (among others).

leaving us confused as to which isomorphism we should chose at any given moment. Therefore, we impose the commutativity of some diagrams. We call those conditions of commutativity 'coherence conditions'.

$$\begin{array}{ccc} (A \otimes B) \otimes 1 & \xrightarrow{\alpha_{A,B,1}} & A \otimes (B \otimes 1) \\ \downarrow{\scriptstyle \rho_{A\otimes B}} & \swarrow{\scriptstyle \mathrm{id}_A \otimes \rho_B} & \\ A \otimes B & & \end{array} \qquad \begin{array}{ccc} (1 \otimes A) \otimes B & \xrightarrow{\alpha_{1,A,B}} & 1 \otimes (A \otimes B) \\ & \searrow{\scriptstyle \lambda_A \otimes \mathrm{id}_B} & \downarrow{\scriptstyle \lambda_{A\otimes B}} \\ & & A \otimes B \end{array}$$

$$\begin{array}{ccc} & ((A \otimes B) \otimes C) \otimes D & \\ \swarrow{\scriptstyle \alpha_{A,B,C} \otimes \mathrm{id}_D} & & \searrow{\scriptstyle \alpha_{A\otimes B,C,D}} \\ (A \otimes (B \otimes C)) \otimes D & & (A \otimes B) \otimes (C \otimes D) \\ \downarrow{\scriptstyle \alpha_{A,B\otimes C,D}} & & \downarrow{\scriptstyle \alpha_{A,B,C\otimes D}} \\ A \otimes ((B \otimes C) \otimes D) & \xrightarrow{\mathrm{id}_A \otimes \alpha_{B,C,D}} & A \otimes (B \otimes (C \otimes D)) \end{array}$$

Many mathematical structures appear under the framework of monoidal categories, most notably, they appear as special objects of the category which are called monoids.

Definition 2.2 (Monoid)**.** Given a monoidal category $(\mathscr{C}, \otimes, 1)$, a monad M is an object of $\mathscr{C}$ with a 'multiplication' morphism $\mu : M \otimes M \to M$ and a unit morphism $\eta : 1 \to M$. Multiplication is associative with identity the unit. In other words, the following diagrams commute.

$$\begin{array}{ccc} M \otimes (M \otimes M) & \xrightarrow{\alpha_{M,M,M}} & (M \otimes M) \otimes M \\ \downarrow{\scriptstyle \mathrm{id}_M \otimes \mu} & & \downarrow{\scriptstyle \mu \otimes \mathrm{id}_M} \\ M \otimes M & & M \otimes M \\ & \searrow{\scriptstyle \mu} \quad M \quad \swarrow{\scriptstyle \mu} & \end{array} \qquad \begin{array}{ccccc} 1 \otimes M & \xrightarrow{\eta \otimes \mathrm{id}_M} & M \otimes M & \xleftarrow{\mathrm{id}_M \otimes \eta} & M \otimes 1 \\ & \searrow{\scriptstyle \lambda_M} & M & \swarrow{\scriptstyle \rho_M} & \end{array}$$

Monoids are objects of the category that have an extra operation. Rings, for example, are Abelian groups with an extra operation; they are, in fact, monoids in the category $(\mathbf{Ab}, \otimes, 1)$ of Abelian groups with their tensor product (as $\mathbb{Z}$-modules). Let's see how that works in detail.

If R is a monoid in $\mathbf{Ab}$, then R is first an Abelian group. Then, there is a homomorphism of groups $\mu : R \otimes R \to R$. This means that if $a, b, c \in R$, then $\mu(a, (b + c)) = \mu((a, b) + (a, c))$ by the tensor product, and $\mu((a, b) + (a, c)) = \mu(a, b) + \mu(a, c)$. This proves distributivity on the left. Distributivity on the right is obtained in a similar fashion. The associativity of μ is provided in the definition of a monoid. Finally, the morphism $\eta : 1 \to M$ simply corresponds to an element $1_M \in M$ such that $\mu(1_M, x) = \mu(x, 1_M) = x$.

3 Cartesian closed categories and their endofunctors

Definition 3.1 (Cartesian category)**.** A Cartesian category $\mathscr{C}$ is a monoidal category $(\mathscr{C}, \otimes, 1)$ where $A \otimes B$ is the product – in the sense of limits – of A and B.

In order for the definition to make sense, a Cartesian category must necessarily have all finite products. Furthermore, it must be a *symmetric* monoidal category, that is, $A \otimes B \cong B \otimes A$ for all objects $A, B \in \mathscr{C}$. The reason is that $A \otimes B$ is the categorical product of A and B which is also the categorical product of B and A, so it is isomorphic to $B \otimes A$.

Example 3.1. The category $(\mathbf{Set}, \times, \{0\})$ is a Cartesian category because $A \times B$ is indeed the categorical product of A and B. It satisfies the universal property: for all $C \in \mathscr{C}$ having two morphisms $f : C \to A$ and $g : C \to B$, there exists a unique morphism $u_{f,g} : C \to A \times B$ such that the diagram below commutes.

$$\begin{array}{ccc} & C & \\ {\scriptstyle f} \swarrow & \downarrow {\scriptstyle u_{f,g}} & \searrow {\scriptstyle g} \\ & A \times B & \\ \swarrow {\scriptstyle \pi^1} & & {\scriptstyle \pi^2} \searrow \\ A & & B \end{array}$$

Definition 3.2 (Closed symmetric monoidal category)**.** In the context of a symmetric monoidal category $\mathscr{C}$, $\mathscr{C}$ is said to be closed when, for all $A \in \mathscr{C}$, the functor $- \otimes A : \mathscr{C} \to \mathscr{C}$ has a right adjoint denoted $[A, -] : \mathscr{C} \to \mathscr{C}$.

A better intuition for what a closed symmetric monoidal category is is that it has 'exponential' objects, or 'internal hom-objects', objects of the category $\mathscr{C}$ which behave, in a certain sense like the set $\mathrm{Hom}(A, B)$. That sense is made precise by the adjunction above.

$$\mathrm{Hom}_{\mathscr{C}}(A \otimes B, C) \overset{f_{A,B,C}}{\cong} \mathrm{Hom}_{\mathscr{C}}(A, [B, C]).$$

The bijection $f_{A,B,C}$ is called currying (homage to Haskell Curry) and its inverse is called uncurrying.

Definition 3.3 (Cartesian closed category)**.** A Cartesian closed category is Cartesian and closed as a symmetric monoidal category.

Example 3.2 (**Set**)**. Set** is Cartesian closed with products being Cartesian products of sets and the exponential object $[A, B]$ being the set of functions from A to B.

Example 3.3 (**Hask**)**.** Another example is the category **Hask**. The name **Hask** comes from the programming language Haskell, which is itself named after Haskell Curry. Haskell is a pure functional programming language meaning that programmed functions work the way mathematical functions work. Whenever a function is called on the same argument, it produce the same result, and it always produces a result. On the contrary 'functions' in other language can fail or produce different values depending on contexts (user input, state of storage disks, network fetches, etc...). Functions in Haskell are always typed, that is, they operate on pre-specified types of data. A type is somewhat analogous to a set. For example, the addition operator can be defined to have the type $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Another example function is $\mathrm{concatenate} : [A] \times [A] \to [A]$ which concatenates two lists with elements of some type A.

The objects of **Hask** are types, and a morphism from A to B is a Haskell function $f : A \to B$. **Hask** is a Cartesian closed category: it has product types $A \times B$ and function types $A \to B$.

3.1 Lambda calculus

Cartesian closed categories have enough structure to be interpreted a lot like the category of types in a functional programming language. More precisely, Cartesian closed categories can give meaningful interpretations to lambda calculus, meaning that there are ways to see objects as types of certain elements and morphisms as functions of these elements. We say that (simply typed) lambda calculus is the *internal language* of Cartesian closed categories.

To see that, let's first describe lambda calculus formally and in detail.

3.2 Category of endofunctors, monads

In the latter part of this document, we will be interested in the Cartesian closed category **Hask**. In **Hask**, endofunctors are very important as they represent well-behaved type constructors: they allow us to construct new types of out existing types and allows to extend functions on the new types.

Theorem 3.1. *The category of endofunctors of any category $\mathscr{C}$ is a strict monoidal category with tensor the composition of functors and unit the identity functor on $\mathscr{C}$.*

The category of endofunctors of a category is very important and the concept of monoids in that category proves especially interesting, so interesting that it has its own name: a monad.

Definition 3.4 (Monad)**.** A monad on $\mathscr{C}$ is a monoid in the category of endofunctors of $\mathscr{C}$.

While the definition is phrased simply, it's important to unwrap the details to observe the complexity of the notion. Being a monoid, a monad is first and foremost an object of the category $[\mathscr{C}, \mathscr{C}]$, that is, a monad T is an endofunctor of $\mathscr{C}$. Additionally, it has a multiplication morphism $\mu : T \otimes T \to T$, that is, there is a natural transformation μ from $T \otimes T = T \circ T = T^2$ to T. There is also some identity morphism from the unit of $[\mathscr{C}, \mathscr{C}]$, that is, there is a natural transformation $\eta : \mathrm{id}_{\mathscr{C}} \to T$. The components of the natural transformations give us some extra insight here.

From now on, we will only be looking at the category **Hask** of Haskell types and Haskell functions between them. A monad T on **Hask** is then an endofunctor of **Hask** which, for every type A, provides two functions: $\eta_A : A \to T(A)$ called `return` and $\mu_A : T(T(A)) \to T(A)$ called `join`. Being a monoid, a monad has to make some diagrams commute, namely, the unit should function as a unit and multiplication should be associative. The diagrams associated are made easier by the fact that the category of endofunctors is strictly monoidal.

$$\begin{array}{ccccc} id_{\mathscr{C}} \circ T & = & T & = & T \circ id_{\mathscr{C}} \\ \downarrow{\scriptstyle \eta T} & & \downarrow{\scriptstyle id_T} & & \downarrow{\scriptstyle T\eta} \\ T \circ T & \xrightarrow[\mu]{} & T & \xleftarrow[\mu]{} & T \circ T \end{array}
\qquad
\begin{array}{ccccc} (T \circ T) \circ T & & = & & T \circ (T \circ T) \\ \downarrow{\scriptstyle \mu T} & & & & \downarrow{\scriptstyle T\mu} \\ T \circ T & & & & T \circ T \\ & \searrow^{\mu} & & {}^{\mu}\swarrow & \\ & & T & & \end{array}$$

Those diagrams are sometimes called 'monad laws'.

An often useful analogy for monads in programming is that of a 'context' in which data lives. Under this analogy, `return` 'returns' a value, that is, wraps it in the context, and `join` 'joins' two contexts into one. An example monad we will look at is the `Maybe` monad.

The `Maybe` monad maps types A to MA whose elements are of the form `Nothing` or `Just a` with a of type A. We will simplify the notation by denoting `Nothing` as $\emptyset$ and `Just a` as $\eta(a)$ (η will be the return of the monad). Functions $f : A \to B$ are mapped to functions $Mf : MA \to MB$ using the following construction:

$$(Mf)(a') = \begin{cases} \emptyset & a' = \emptyset \\ \eta(f(a)) & a' = \eta(a) \end{cases}.$$

The `Maybe` monad helps represent operations that can fail. For example, the division operation on reals would ideally be of type $/ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. However, division by 0 can fail, so we can instead give it the type $/ : \mathbb{R} \times \mathbb{R} \to M\mathbb{R}$, and define division as such:

$$a/b = \begin{cases} \eta(a \div b) & b \neq 0 \\ \emptyset & b = 0 \end{cases}.$$

A problem quickly arises from the fact that the co-domain is a different type that the domain, as it's impossible to compose these functions. This is where the monadic structure of `Maybe` comes in handy. With M being a functor, $f : A \to MB$ can produce $Mf : MA \to MMB$. Composing Mf with μ_B gives us $\mu_B \circ Mf : MA \to MB$. In other words, the monadic structure of M allows us to extend any possibly failing function on A to a possibly failing function on MA. The monadic laws ensure that when the computation of MA fails (we have $\emptyset$), so does the composition with Mf. In other words, $(Mf)(\emptyset) = \emptyset$. The function that maps $f : A \to MB$ to $\mu_B \circ Mf$ is called `bind`, and is generally denoted as the operator $>>=: MA \to (A \to MB) \to MB$. The notation comes from the analogy of monads being boxes and environments, $x >>= f$ 'forces' x which is wrapped in a box into f which only accepts unwrapped values.

In more mathematical terms, `bind` gives a way to compose $f : A \to MB$ with $g : B \to MC$. $g \circ_M f = (>>= g) \circ f$. For each type A there is a unique function that behaves like an identity for this composition, denoted as $\circ_M$, namely $\eta_A : A \to MA$.

Example 3.4 (Division by 0)**.** Suppose we would like to use the function above to compute `((54/25)/0)/3`. We would at first want to calculate 54/25. Since the division is successful, we will obtain $\eta(54 \div 25)$, so far so good. Next, we want to (try to) divide that by 0. But it is not possible to use the type $M\mathbb{N}$ with the division as defined. Therefore, we will use the `bind` operation: `(54/25) >>= (λ x. x/0)` where λ is used to denote a function $x \mapsto x/0$. Next, the result of that is supposed to be divided by 3, so we repeat the same operation and compute `((54/25) >>= (λ x. x/0)) >>= (λ x. x/3)`.

Certain programming languages that employ monads attempt to make this composition cleaner by introducing special notations for it. In Haskell, we can use the `do` notation to hide the issues of `bind` behind a notation of 'extraction' out of a monad `<-`.

```
do:
  a <- 54/25
  b <- 54/0
  b/3
```

3.3 Kleisli category

Overall, this composition, along with η_A for each A, gives rise to a category denoted as $\mathscr{C}_T$ where $\mathscr{C}$ is the underlying category and T is the monad in question and called the Kleisli category of $\mathscr{C}$ associated with T. $\mathscr{C}_T$ has the same objects as $\mathscr{C}$, but a morphism $\tilde{f} : A \to B$ in $\mathscr{C}_T$ corresponds to a morphism $f : A \to TB$. This construction allows us to define a functor $F : \mathscr{C} \to \mathscr{C}_T$ which is the identity of objects and which, for each morphism $f : A \to B$ associates the morphism $F(f) = \eta_B \circ f$ in $\mathscr{C}_T$

In a sense, the Kleisli category is just $\mathscr{C}$ but where functions always wrap their values in the environment T after they're done computing them. We might Since T is a monad, it also makes sense to ask for wrapping values **before** computing them, that is, making the domain of arrows be in the image of T. We can then define another functor $G : \mathscr{C}_T \to \mathscr{C}$ which will do this pre-wrapping.

$$
\begin{aligned}
GA &= TA \\
Gf &= \mu_A \circ Tf.
\end{aligned}
$$

3.3.1 Kleisli adjuction

Notice then that F and G factor the monad T through the category $\mathscr{C}_T$ as $GF = T$. Furthermore, the existence of $\eta : \mathrm{id}_{\mathscr{C}} \to GF$ hints at an adjunction with unit η and counit some

natural transformation $\varepsilon : FG \to id_{\mathscr{C}_T}$. Let's attempt to construct ε. For each type A in the Kleisli category, $\varepsilon_A : FGA \to A$ is a morphism in $\mathscr{C}_T$, that is, it corresponds to a morphism $\varepsilon_A : TA \to TA$ in $\mathscr{C}$. We can chose this morphism to be the identity morphism of TA, so that $\varepsilon_A = \mu_A \circ \eta_{TA}$, so $\varepsilon = \mu\eta T$.

We can finally verify the triangle identities for the adjunction.

$$\begin{array}{ccc} F & \xrightarrow{F\eta} & FGF \\ & \overset{\mathrm{id}_F}{\searrow\!\!\searrow} & \big\downarrow \varepsilon F \\ & & F \end{array} \quad \text{in } \mathscr{C}_T \text{ arises from} \quad \begin{array}{ccc} A & \xrightarrow{\eta_{T(A)}\eta_A} & T(T(A)) \\ & \overset{\eta_A}{\searrow} & \big\downarrow \mu_A \\ & & T(A) \end{array} \quad \text{in } \mathscr{C}.$$

$$\begin{array}{ccc} G & \xrightarrow{\eta G} & GFG \\ & \overset{\mathrm{id}_G}{\searrow\!\!\searrow} & \big\downarrow G\varepsilon \\ & & G \end{array} \quad \text{in } \mathscr{C} \text{ arises from} \quad \begin{array}{ccc} T(A) & \xrightarrow{\eta_{T(A)}} & T(T(A)) \\ & \overset{\mathrm{id}_{T(A)}}{\searrow\!\!\searrow} & \big\downarrow \mu_A \\ & & T(A) \end{array} \quad \text{in } \mathscr{C}.$$

3.3.2 Kleisli construction of a monad out of an adjunction

In fact, it turns out that every adjunction has an associated monad. Let $F \dashv G$ be an adjunction. Then $GF : \mathscr{C} \to \mathscr{C}$ is a monad of $\mathscr{C}$. The unit of GF is the unit of the adjunction $\eta : \mathrm{id}_C \to GF$, and the multiplication $\mu : GFGF \to GF$ is defined as $\mu = G\varepsilon F$.

$$\begin{array}{c} \mathscr{C} \\ F \big\downarrow \big\uparrow G \\ \mathscr{D} \end{array}$$

The verification of the monad laws gives some extra insight on the construction, so we will do that in detail.

$$\begin{array}{ccccc} id_{\mathscr{C}} \circ T & = & T & = & T \circ id_{\mathscr{C}} \\ \eta T \big\downarrow & & \big\downarrow id_T & & \big\downarrow T\eta \\ T \circ T & \xrightarrow[\mu]{} & T & \xleftarrow[\mu]{} & T \circ T \end{array} \qquad \begin{array}{ccc} (T \circ T) \circ T & = & T \circ (T \circ T) \\ \mu T \big\downarrow & & \big\downarrow T\mu \\ T \circ & & T \circ T \\ & \overset{\mu}{\searrow} \quad T \quad \overset{\mu}{\swarrow} & \end{array}$$

translate to

$$\begin{array}{ccccc} id_{\mathscr{C}} GF & = & GF & = & GF id_{\mathscr{C}} \\ \eta GF \big\downarrow & & \big\downarrow id_{GF} & & \big\downarrow GF\eta \\ GFGF & \xrightarrow[G\varepsilon F]{} & GF & \xleftarrow[G\varepsilon F]{} & GFGF \end{array} \qquad \begin{array}{ccc} (GFGF)GF & = & GF(GFGF) \\ G\varepsilon FGF \big\downarrow & & \big\downarrow GFG\varepsilon F \\ GFGF & & GFGF \\ & \overset{G\varepsilon F}{\searrow} \quad GF \quad \overset{G\varepsilon F}{\swarrow} & \end{array}$$

The diagram for the unit splits in two:

$$\begin{array}{ccc} GF & & \\ \eta GF \big\downarrow & \overset{\mathrm{id}_{GF}}{\searrow\!\!\searrow} & \\ GFGF & \xrightarrow[G\varepsilon F]{} & GF \end{array} \quad \text{and} \quad \begin{array}{ccc} & & GF \\ & \overset{\mathrm{id}_{GF}}{\swarrow\!\!\swarrow} & \big\downarrow GF\eta \\ GF & \xleftarrow[G\varepsilon F]{} & GFGF \end{array}$$

Which follow from the triangle identities for the adjunction $F \dashv G$.

And the diagram for associativity is obtained by composing the diagram for the naturality of ε on itself with G on the left and F on the right (rotate your head 45° to see it better).

$$\begin{array}{ccc} FGFG(A) & \xrightarrow{FG(\varepsilon_A)} & FG(A) \\ {\scriptstyle \varepsilon_{FG(A)}}\downarrow & & \downarrow{\scriptstyle \varepsilon_A} \\ FG(A) & \xrightarrow[\varepsilon_A]{} & A \end{array}$$

3.3.3 Application on the free/forgetful monoid adjunction

Return to the notations of 3.3.2 with $\mathscr{C} = \mathbf{Set}$ the category of sets and their functions, $\mathscr{D}$ the category of monoids and their homomorphisms, F the free monoid functor, and G the forgetful functor. The unit of this adjunction is $\eta : \mathrm{id}_{\mathscr{C}} \to GF$ which, for each set X provides the function $\eta_X : x \mapsto [x]$ the formal single-letter word. And the counit $\varepsilon : FG \to \mathrm{id}_D$ has components at a monoid M the evaluation homomorphism into M from the free monoid obtained by constructing formal words from the underlying set of M.

The constructed monad $T : \mathscr{C} \to \mathscr{C}$ is the functor mapping a set X to the set of formal words with alphabet X. The multiplication $\mu_X : T(T(X)) \to T(X)$ sends a formal word of formal words of X which we will write as $x = [[x_{1,1}, \cdots, x_{1,m_1}], \cdots, [x_{n,1}, \cdots x_{n,m_n}]]$ to $G\varepsilon_{F(X)}(x) = \varepsilon_{T(X)}(x) = [x_{1,1}, \cdots, x_{1,m_1}, \cdots x_{n,m_n}]$, and the unit $\eta_X : X \to T(X)$ simply maps $x \mapsto [x]$.

The monad unit laws can be deduced because $\mu([[x]]) = [x]$. The associativity law requires that, given a 3-dimensional word in the set X, flattening is associative on axes: flatting axis 1 and 2 then flatting that with axis 3 is the same as flatting axis 2 and 3, then flatting 1 with the result (notice the order of axes is respected). This is, of course, true.

The free monoid adjunction gave us what in Haskell is called the List monad. It naturally gave rise to flattening, mapping functions on elements, returning singletons, and the very useful `flatMap` of type `[A] -> (A -> [B]) -> [B]` which is simply the `bind` of the monad and which maps $[x_1, x_2, \cdots, x_n]$ to the concatenation of $f(x_1), \cdots, f(x_n)$.

Similarly, the `Maybe` monad can be obtained from the free forgetful adjunction where, the free functor maps a type A to the type $A+1$, the disjoint union of A and a fixed singleton type 1 and the forgetful functor simply forgets the fact that $A+1$ is a disjoint union with 1.

3.4 Properties of monads over Cartesian closed categories

Note: this section remains unfinished

Definition 3.5 (Lax monoidal functor)**.** If $(\mathscr{C}, \otimes, I_C)$ and $(\mathscr{D}, \bullet, I_D)$ are monoidal categories, a lax monoidal functor $F : \mathscr{C} \to \mathscr{D}$ is a functor with a natural transformation $\varphi_{A,B} : FA \bullet FB \to F(A \otimes B)$ and a morphism $\psi : I_{\mathscr{D}} \to FI_{\mathscr{C}}$ satisfying the following coherence conditions.

$$\begin{array}{ccc} (FA \bullet FB) \bullet FC & \xrightarrow{\alpha_{\mathscr{D};FA,FB,FC}} & FA \bullet (FB \bullet FC) \\ {\scriptstyle \varphi_{A,B} \bullet \mathrm{id}_{FC}}\downarrow & & \downarrow{\scriptstyle \mathrm{id}_{FA}\varphi_{B,C}} \\ F(A \otimes B) \bullet FC & & FA \bullet F(B \otimes C) \\ {\scriptstyle \varphi_{A\otimes B,C}}\downarrow & & \downarrow{\scriptstyle \varphi_{A,B\otimes C}} \\ F((A \otimes B) \otimes C) & \xrightarrow[F\alpha_{\mathscr{C};A,B,C}]{} & F(A \otimes (B \otimes C)) \end{array}$$

$$\begin{array}{ccc} FA \bullet I_{\mathscr{D}} & \xrightarrow{\mathrm{id}_{FA} \circ \psi} & FA \bullet FI_{\mathscr{C}} \\ \rho_{\mathscr{D};FA} \downarrow & & \downarrow \varphi_{A,I_{\mathscr{C}}} \\ FA & \xleftarrow[F\rho_{\mathscr{C};A}]{} & F(A \otimes I_{\mathscr{C}}) \end{array} \qquad \begin{array}{ccc} I_{\mathscr{D}} \bullet FA & \xrightarrow{\psi \circ \mathrm{id}_{FA}} & FI_{\mathscr{C}} \bullet FA \\ \lambda_{\mathscr{D};FA} \downarrow & & \downarrow \varphi_{I_{\mathscr{C}},A} \\ FA & \xleftarrow[F\lambda_{\mathscr{C};A}]{} & F(A \otimes I_{\mathscr{C}}) \end{array}$$

Remark 3.1. (Laxity and strictness)

- If $F : \mathscr{C}^{\mathrm{op}} \to \mathscr{D}^{\mathrm{op}}$ is a lax monoidal functor on the opposite categories, it is said to be **colax monoidal** from $\mathscr{C}$ to $\mathscr{D}$.
- If ϕ and ψ are isomorphisms, F is said to be **pseudo-monoidal**.
- If ϕ and ψ are identity morphisms, that is $FA \bullet FB = F(A \otimes B)$ and $FI_{\mathscr{C}} = I_{\mathscr{D}}$, F is a **strict monoidal**.

Definition 3.6. (Applicative functor) Let $\mathscr{C}$ be a Cartesian closed category and F be an endofunctor of $\mathscr{C}$. F is said to be *applicative* if it admits a natural transformation $\eta : \mathrm{id}_{\mathscr{C}} \to F$ and a natural transformation $\phi_{A,B} : F(A \to B) \to (F(A) \to F(B))$ (as exponentials) such that certain coherence conditions are met.

Notice that the existence of η and ϕ defines a morphism $\phi_{A,B} \circ \eta_{A \to B} : (A \to B) \to (F(A) \to F(B))$. However, F is a functor, and so it naturally has an action on morphisms, so it provides a morphism $F_{A,B} : (A \to B) \to (F(A) \to F(B))$ which is just the morphism map of F at $\mathrm{Hom}(A, B)$. We require that these two maps be the same.

$$\begin{array}{ccc} (A \to B) & \xrightarrow{\eta_{A \to B}} & F(A \to B) \\ & \searrow^{F_{A,B}} & \downarrow \phi_{A,B} \\ & & (F(A) \to F(B)) \end{array}$$

In particular, this means that $\phi_{A,A}(\eta_{A \to A}(\mathrm{id}_A)) = \mathrm{id}_{F(A)}$. Fhis is known as the

Theorem 3.2. *(Applicativity of monads) Let $\mathscr{C}$ be a Cartesian closed category, and T be a monad over $\mathscr{C}$. Then, T is* ***applicative****, that is, there is a natural transformation $\phi : T(A \to B) \to (T(A) \to T(B))$.*

4 String diagrams

In categories without a monoidal structure, there is a unique canonical composition of morphisms, namely, horizontal composition of $f : A \to B$ and $g : B \to C$ into $g \circ f = f; g : A \to C$. That composition is monoidoidal, that is, it's associative and has units for each object, in other words, it's like a monoid, but 'horizontally categorified'. Horizontal categorisation is, roughly, the process of generalizing an algebraic structure to a structure on categories by seeing the original algebraic structure as a category with one object. The rule for naming such categories is to add an 'oid' postfix to the name of the structure. A monoidoid is simply a category.

The use of the term 'monoidoidal' explains the connection between categories and monoids, and allows us to introduce simpler, cleaner notation for morphism composition in categories. Compositions are simply (well constructed) words in the alphabet being the set of morphisms in the category. So, instead of writing $(f; g); h$, we may simply write fgh which is also equal to $f; (g; h)$. This common notation simplifies matters of associativity and unitality.

In a (strictly) monoidal category, there is another way to compose morphisms: vertical composition. If $f : A \to B$ and $g : C \to D$, $f \otimes g : A \otimes C \to B \otimes D$. This composition is

also (strictly) associative and (strictly) unital, which gives rise to the fun name of 'monoidal monoidoid' for monoidal categories reflecting the double monoidoidal structure.[2]

This double monoidal structure inspires string diagrams, which are essentially two-dimensional words: horizontal composition is written horizontally and vertical composition vertically.

For example, the following string diagram represents the morphism $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

Technically speaking, this diagram is built inductively either as $f \otimes (g \otimes h)$ or as $(f \otimes g) \otimes h$. We represent this parenthesizing graphically as a dotted rectangle, but because the morphisms represented are morphisms of a strict monoidal category, the two constructions coincide, so there is no need to keep track of the construction steps, and the dotted rectangles are optional.

Just as an empty word represents the identity morphism, an empty box represents the identity morphism in string diagrams. Here's an example of the identity on an object A.

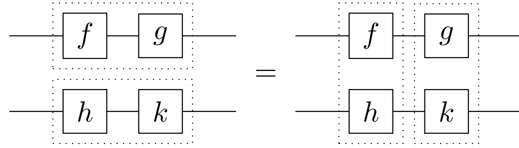
Similarly, the identity object of the category is denoted by an empty line. So the identity morphism on the identity object can be represented by an empty dotted box, or by a completely empty diagram.

This empty box notation explains the intuitive unitality laws for vertical composition.

Similar unitality and associativity laws apply to horizontal composition.

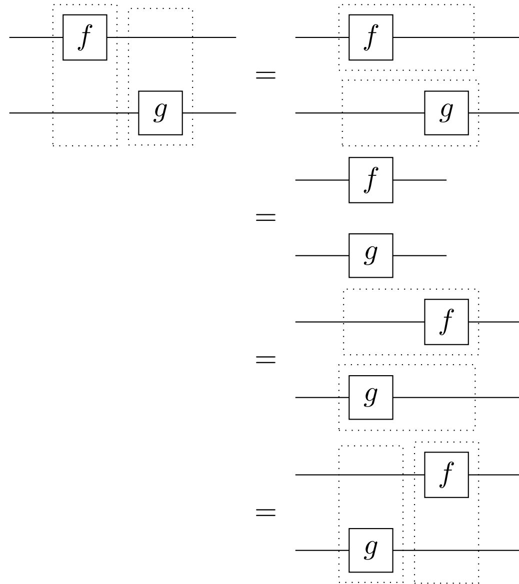
However, another form of associativity appears by combining the two operations, namely, the possibility of interchanging the two kinds of composition. This is one of the equations that's starting to appear a little simpler using string diagrams; however, we will see that they have much more use than that.

[2]https://ncatlab.org/nlab/show/monoidal+category

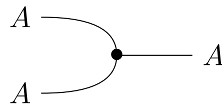


In text, this is the equation $(f; g) \otimes (h; k) = (f \otimes h); (g \otimes k)$.

The equations given allow us to safely ignore some intuitive transformations of the drawing of a diagram. For example,



Sometimes, instead of writing the name of a morphism in a box, we may use a special symbol for it, like a solid or empty dot for convenience, especially when it involves composite objects as domain or co-domain. Here is an example of an operator $f : A \otimes A \to A$ denoted as a solid disk.



4.1 String diagrammatic representation of monoidal laws

In a monoidal category $\mathscr{C}$, we recall that a monoid M is an object of $\mathscr{C}$ with morphisms $\mu : M \otimes M \to M$ and $\eta : \mathrm{id}_{\mathscr{C}} \to M$ such that μ is associative and η acts as the (left and right) identity for μ. If $\mathscr{C}$ is strictly monoidal, we can represent η and μ using string diagrams. In which case, the associativity and unitality laws become simple equations of string diagrams. If we represent μ as a solid dot and η as an empty dot, we can write the associativity and unitality laws as follows.

(9)

(10)

(11)

As we have defined them earlier, monads over a category $\mathscr{C}$ are monoids in the strictly monoidal category $[\mathscr{C},\mathscr{C}]$ of endofunctors of $\mathscr{C}$. $[\mathscr{C},\mathscr{C}]$ being strictly monoidal, we can use equations of string diagrams 9, 10, and 11 to express the monadic laws of associativity and unitality.

4.2 More types of string diagrams

String diagrams have found very many applications because of their versatility and flexibility in denoting complex morphisms with intuitive drawings. We will give some examples here.

Example 4.1. (Symmetric monoidal category) A symmetric monoidal category is a monoidal category where we can commute products. Formally, it's monoidal category $\mathscr{C}$ with a natural isomorphism $\sigma_{X,Y} : X \otimes Y \to Y \otimes X$ satisfying $\sigma_{X,Y}; \sigma_{Y,X} = \mathrm{id}_{X,Y} = \mathrm{id}_X \otimes \mathrm{id}_Y$ (and coherent with associators and unitors).

If the category is strictly monoidal, we can represent $\sigma_{X,Y}$ as a swap of wires.

X Y

Y X

The equation from above then translates to untwisting a double swap.

X X = X Y

Y Y Y X

Example 4.2. (Braided monoidal category) The swapping operation from above is an isomorphism $X \otimes Y \to Y \otimes X$ with extra properties (the untwisting). We could instead ask for fewer properties. By doing that, we get braided monoidal categories. Braided (strict) monoidal categories are strict monoidal categories with a natural isomorphism $\beta_{X,Y} : X \otimes Y \to Y \otimes X$ which may or may not satisfy the condition $\beta_{X,Y}; \beta_{Y,X} = \mathrm{id}_{X \otimes Y}$ from symmetric monoidal categories. A braiding $\beta_{X,Y}$ is also denoted as a swap of lines but which respects which line went first, i.e. it is a 3-dimensional passing of lines one in front of the other.

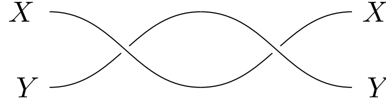
$\beta_{X,Y} =$ X Y

Y X

$\beta^{-1}_{X,Y} =$ Y X

X Y

We see with these diagrammatic representations that two string diagrams represent the same morphism when, seen in 3 dimensions, the strings can slide over one another. However a braid $\beta_{X,Y};\beta_{Y,X}$ cannot visually be disentangled without cutting the strings (i.e. adding an extra equation). So the braids may provide non-trivial endomorphisms of $X \otimes Y$.



Due to the topological significance of the string diagrams in braided monoidal categories, it's common to find applications involving swaps that 'leave trace'. For example, in topological quantum computing, the string diagram above can be seen as a pair of paths in 3-dimensions, the left to right dimension being time, and the other two dimensions (down-up and in-out) being dimensions of space. When X and Y are seen as 'anyons' which are quasi-particles appearing only in 2-dimensional systems, this braiding cannot be disentangled as it would require the paths to intersect, which is not possible for this type of particles according to the Fermi principle which states that it's impossible for two of these particles to have the same quantum state. Switching, say, anticlockwise, the positions of two anyons twice results in a different state than if they were not switched. This is in contrast with the usual 3-dimensional particles which are of two types: fermions and bosons. For fermions like electrons, this braiding (switching the positions counter-clockwise) applied twice leaves you with the original quantum state; therefore, if the category concerned Fermion objects, the braided monoidal category would be a symmetric monoidal category with this braiding. And in the case of bosons like photons, swapping X and Y leaves the system in the same quantum state (bosons do not abide by the Fermi principle and can be in the same quantum state, that is, lines can intersect). Categorically, this means that the braiding is the identity and the category is strictly symmetric. Topological quantum computing encodes computations as string diagrams consisting of different braidings and relies on anyons to make sure that the braidings do actually provide a non-trivial structure, and on the Fermi principle to ensure the stability of the computation by forbidding the disentanglement of braids.

5 Wedges and ends, cowedges and coends.

In this section, we will be interested in some special kinds of limits of functors of the form $F : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$. When $\mathscr{D} = \mathbf{Set}$, which is often the case, we call F a *profunctor*. A prime example of profunctors is the hom functor. We are in particular interested in the behavior of F on the diagonal. F is a functor on each component, but because it is a contravariant in the first and covariant in the second, $A \to F(A, A)$ does not define a functor, so it's not possible to define limits and colimits of it because they require the use of the notion of natural transformations. However, we can define a more general notion: dinatural transformations.

Definition 5.1 (Dinatural transformation)**.** If $F, G : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ are functors, a dinatural transformation $\alpha : F \xrightarrow{\bullet} G$ is a family of morphisms α_A for $A : \mathscr{C}$ such that, for all $A, B : \mathscr{C}$, the following diagram commutes.

$$
\begin{array}{ccccccc}
 & & F(A,A) & \xrightarrow{\alpha_A} & G(A,A) & & \\
 & \overset{F(f,\mathrm{id}_A)}{\nearrow} & & & & \overset{G(\mathrm{id}_A,f)}{\searrow} & \\
F(B,A) & & & & & & G(A,B) \\
 & \underset{F(\mathrm{id}_B,f)}{\searrow} & & & & \underset{G(f,\mathrm{id}_B)}{\nearrow} & \\
 & & F(B,B) & \xrightarrow[\alpha_B]{} & G(B,B) & &
\end{array}
$$

Definition 5.2 (Wedge, cowedge)**.** If $F : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ is a functor, a wedge of F is an object $A : \mathscr{D}$ and dinatural transformation $\alpha : A \overset{\bullet}{\to} F$. Dually, a cowedge admits a dinatural transformation $\alpha : F \overset{\bullet}{\to} A$.

If $f : A \to B$ and B is the summit of a wedge of F with dinatural transformation α and such that $\alpha \circ f$ is a dinatural transformation, f can be seen as a morphism of wedges from the wedge on A to the wedge on B. We can thus form the category of wedges.

Definition 5.3 (End, coend)**.** An end is a universal wedge, that is an initial object in the category of wedges. Dually, a coend is a universal cowedge, that is, a terminal object in the category of cowedges.

An end of $F : \mathscr{C}^{\mathrm{op}} \times \mathscr{C} \to \mathscr{D}$ is denoted $\int_{C:\mathscr{C}} F(C,C)$ and a coend is denoted $\int^{C:\mathscr{C}} F(C,C)$.

6 Optics

Optics are inspired from computer science and provide ways to inspect and modify parts of a data structure without having to change the remainder of the structure. The concept of mixed optics in category theory generalises that a lot.

6.1 Lenses

Lenses originated as a way to split a complex record data structure into a focus and a context. Here is an example. Suppose we would like to write a program that browses Wikipedia. Somewhere in our programming process, we are likely to need to split URLs of the form `https://en.wikipedia.org/wiki/Category_theory` into the context `https://en.wikipedia.org/wiki/` and the focus `Category_theory` in such a way that we are able to replace the focus with some other focus `Functor` (another Wikipedia page), and reconstruct our URL to obtain `https://en.wikipedia.org/wiki/Functor`.

We will need two functions to do the task: a *view* function which takes in the string URL and returns a pair of strings consisting of the context and the focus, and another *update* function which takes in a pair consisting of the context and a new value for the focus, and reconstructs the complete URL. In other words, we need functions $l : \mathrm{String} \to \mathrm{String} \times \mathrm{String}$ and $r : \mathrm{String} \times \mathrm{String} \to \mathrm{String}$. More generally, we need functions $l : S \to M \times A$ and $r : M \times B \to T$ where the idea is that we change the focus A to the focus B, and thus change the overarching structure S to the structure T, but we keep the same context M.

To formally express the fact that M remains untouched, we need to somehow make sure to separate M from A and B. This means then that any map on M can be applied when M is paired to the focus A or to the focus B without changing the result T, so the replacement of A by B is not dependent on M and M just "floats around" to accompany the focus so that we are able to reconstruct the initial data structure. This means that we need to quotient the set of pairs (M, l, r) by the relation: (M, l_1, r_1) (N, l_2, r_2) iff there exists $l : S \to M \times A$, $r : N \times B \to T$, and $f : M \to N$ such that

$$(l_1, r_1) = ((f \times \mathrm{id}_A) \circ l, r)$$
$$(l_2, r_2) = (l, r \circ (f \times \mathrm{id}_B))$$

This set is precisely $\int^{M:\mathscr{C}} \mathscr{C}(S, M \times A) \times \mathscr{C}(M \times B, T)$. We call elements of this set *lenses* from (S, T) to (A, B).

6.2 Prisms

Prisms are obtained from lenses by replacing the product with a sum (coproduct). In other words, the set of prisms from (S, T) to (A, B) is the set $\int^{M:\mathscr{C}} \mathscr{C}(S, M + A) \times \mathscr{C}(M + B, T)$. Prisms are especially useful in situations where it's not always possible to extract an A from an S. For example, in parsing a string into an integer, if indeed the string represents an integer, we can safely return an integer; however, if the string, for example, contains letters, we cannot parse it into an integer, we can instead return the string itself, or some error, perhaps. We can thus define parsing function $l : \mathrm{String} \to \mathrm{String} + \mathrm{Int}$. To complete the parsing, we can also define a printing function, possibly for something other than integers $r : \mathrm{String}+\mathrm{Float} \to \mathrm{String}$. The pair (l, r) forms a prism which focuses on the integer in a string, if possible, and is able to follow that by a replacement, if we have an integer, by a float (perhaps we may choose it to be the same number written with .0 at the end).

6.3 Optics

Optics are generalizations of lenses and prisms. An optic from (S, T) to (A, B) is an element of the set $\int^{M:\mathscr{C}} \mathscr{C}(S, M \otimes A) \times \mathscr{C}(M \otimes B, T)$ where a changing definition of the monoidal operation on $\mathscr{C}$ changes the kind of optic we're dealing with. In the case of a Cartesian monoidal category $\mathscr{C}$, $\otimes$ is the categorical product, so optics are lenses. If $\mathscr{C}$ has all finite coproducts, then by defining a tensor product $A \otimes B = A + B$ on $\mathscr{C}$, we obtain prisms. For most of computer science, this is all that's necessary because we are always, or almost always working within the same category, the category of types and functions between them.

6.4 Actions of categories, or actegories

We would like, at that point, to generalize $M \otimes A$ and $M \otimes B$ to have, possibly, M, A, and B belonging to distinct categories, $\mathscr{M}$, $\mathscr{C}$, and $\mathscr{D}$. Of course, if M and A belong to different categories, it doesn't make sense to talk about their tensor product. Instead, as long as M belongs to a monoidal category $\mathscr{M}$, we can define $M \bullet A$, the action of M on A in a fashion very similar to the action of classical monoids (i.e. monoidal sets) on sets.

In other words, we start with a monoidal functor $\phi : (\mathscr{M}, \otimes, 1) \to ([\mathscr{C} : \mathscr{C}], \circ, \mathrm{id}_{\mathscr{C}})$. Depending on the author, this can have the strictness and laxity that the author desires; here we will assume we are dealing with pseudo-monoidal functors. Using ϕ, we can define an action of the monoidal category $\mathscr{M}$ on $\mathscr{C}$ as follows for an object A and a morphism f in $\mathscr{C}$.

$$M \bullet A = \phi(M)(A)$$
$$M \bullet f = \phi(M)(f)$$

It's also worth nothing that since ϕ is a functor, it has values on morphisms, so if $f : M \to N$ in $\mathscr{M}$, $\phi(f) = f \bullet - : M \bullet - \to N \bullet -$ is a natural transformation in $\mathscr{C}$.

The fact that ϕ is a pseudo-monoidal gives us generalizations of the basic properties of actions of monoids on sets.

$$I_{\mathscr{M}} \bullet - \cong \mathrm{id}_{\mathscr{C}}$$
$$(M \otimes N) \bullet - \cong M \bullet (N \bullet -)$$

We say that $\mathscr{C}$ is an $\mathscr{M}$-actegory to mean that it is a category with an action from $\mathscr{M}$.

6.5 Mixed optics

Given a monoidal category $\mathscr{M}$ and two $\mathscr{M}$-actegories $\mathscr{C}$ and $\mathscr{D}$, the category of $(\mathscr{C}, \mathscr{D})$-mixed optics has objects (A, B) with $A : \mathscr{C}$ and $B : \mathscr{D}$ and morphisms $(S, T) \to (A, B)$ elements of the set $\int^{M:\mathscr{M}} \mathscr{C}(S, M \bullet A) \times \mathscr{D}(M * B, T)$.

If $\mathscr{M} = \mathscr{C} = \mathscr{D}$, the tensor product in $\mathscr{M}$ is itself an action on $\mathscr{M}$, just like a group acts on itself by multiplication. Therefore, the optics mentioned in 6.5 are nothing more than special cases of mixed optics.

A List of interesting monads in computer science

The monads in this section are chosen as endofunctors of **Hask** and the examples will be in the Haskell language, when necessary.

A.1 Reader

The reader monad allows one to encode computations that are dependent on a certain context, or a certain value to be 'read'. This can, for example, be used to pass global constants or configuration settings for a program without having to reference them explicitly every time.

If the desired environment type for the reader monad is A, then the monad is defined as the endofunctor $-^A$, mapping X to the type of functions out of A into X, so that all the values treated of type X in this context would actually be dependent on the global context of type A.

The unit (return) of the reader monad returns a constant value, independent of the context A. That is, $\eta_X : X \to X^A; x \mapsto (- \mapsto x)$, and the multiplication consists of realizing that the type asks for the global context twice when it only needs it once: $\mu_X : (X^A)^A \to X^A$ which for a function $(a \mapsto (b \mapsto x(b)))$ produces the function $a \mapsto x(a)$ which only reads the (same) context once.

A.2 Writer

The writer monad is, in a sense, dual to the reader monad. Here, the values don't depend on any context, but the computations can produce 'things' that we need to keep, but that don't affect the computation. The prime example of that is logging. It's often the case that we want to keep logs of what the program has done while it's running so that we can diagnose it later for errors, or for statistical purposes, or for legal purposes; however, we would not like to have to carry the logs into every single computation we do explicitly partly for practicality partly because they do not affect the computations at all.

The writer monad is also dependent on the type of 'written' data. However, there are conditions: the 'writing' type must have a monoidal structure. This is because the most important example of write monads are log monads which need to append text to another piece of already written log text and that may simply produce no log sometimes (empty string). If $(M, \cdot, 1)$ is the monoidal type of the written data, then the writer monad for M is defined as the functor $- \times M$ with the following join and return transformations.

$$
\begin{aligned}
\eta_X : X &\to X \times M \\
x &\mapsto (x, 1) \\
\mu_X : (X \times M) \times M &\to X \times M \\
((x, m), m') &\mapsto (x, m \cdot m')
\end{aligned}
$$

A.3 IO

Perhaps one of the most impressive uses of monads in programming is their ability to encode impure computer functions which can read and write data in memory, on disk, over a network, etc... into pure functions that do none of that. The secret is to find implement the 'dirty' work of input and output separately in some abstract definition of an endofunctor IO. $\mathrm{IO}(A)$ is a type representing values of type A produced after some input/output operation. For example, a element of type $\mathrm{IO}(\mathbb{N})$ could be 'read user input and return it interpreted as an integer', or if 1 is the type with a single element $*$, elements of $\mathrm{IO}(1)$ are IO operations that don't produce any value (or that we don't care about keeping the values they possibly produce). They can also be combinations of input and output; for example, we may want to print a prompt on the screen (output), then read a response from the user (input) and finally produce some value parametrized by the user's response.

The exact implementations are irrelevant to the programmer because they are completely abstracted away by the IO type.

The return of the IO monad $\eta_A : A \to \mathrm{IO}(A)$ maps a value a to a computation which does not actually do any I/O computations and always return a. And the join of the IO monad $\mu_A : \mathrm{IO}(\mathrm{IO}(A)) \to \mathrm{IO}(A)$ maps a sequences I/O operations. An element of $\mathrm{IO}(\mathrm{IO}(A))$ consists of an I/O operation, which, after it's executed, return another I/O operation to execute which itself returns a value of type A. μ_A is then the function that combines the two operations sequentially into one larger operation.

In summary, the IO monad is monadic precisely because of the monoidal structure of I/O operations: there is a do-nothing operation and a way to sequentially combine operations.

A.4 Continuation

A somewhat common pattern in functional programming is the 'Continuation-Passing Style' or CPS. In CPS, a function might not necessarily do all the work to produce a value, but once it's done its own job, can be composed (on the left) by a 'continuation' function which does the rest of the work from then on. CPS allows for finer control of the order of computations and can allow for optimization of program efficiency. The main take away is that the continuation of the computation is passed onto the previous chunk, so the 'past' computation can control how the 'future' continuation computations are to be done.

To define the monad, we have to fix a final result type R. Then, instead of using values of type A, we use continuations of the computation after A. We define $\mathrm{Cont}(A) = (A \to R) \to R$. Instead of using $a : A$, we use functions that, given a continuation after A, that is, given a function $A \to R$, can complete the computation and produce R.

The simplest form of that is to use an element of A to produce R from $A \to R$. This provides us with the unit of the monad: $\eta_A : A \to ((A \to R) \to R)$ which maps $a : A$ to $(f \mapsto f(a))$. The multiplication is a bit more complex notationally:

$$
\begin{aligned}
\mu_A : &((((A \to R) \to R) \to R) \to R) \to ((A \to R) \to R) \\
&f \mapsto (g \mapsto (f(h \mapsto h(g))))
\end{aligned}
$$

What all of this means in English is that applying the composition $\text{Cont}(\text{Cont}(A))$ to a function $\text{Cont}(A) \to R$ allows us to continue the computation after we have reached $\text{Cont}(A)$, so it encodes reaching $\text{Cont}(A)$ along with having control over how to do the next operations. But reaching $\text{Cont}(A)$ itself encodes reaching A and information on how to proceed after reaching A. So if we reach A, and are provided with a continuation function from A to R, we are able to 'rewind' doing the inner continuation first to produce a continuation function $\text{Cont}(A) \to R$ which can then be used with the outer continuation to finalise the computation and reach R.

This again exploits the monoidal structure of operations.

A.5 Parser

Another frequently used construction is the parser monad. Parsing text means discovering structure in a string of characters. For example, an appropriate parser for addition of integers could interpret the string of characters `"21 + 45"` as meaning the number 21, the + operator, and the number 45, in that order. A function that would parse `"21 + 45"` is then expected to have an input type `String`, that is, a string of characters, and an output type some structure that can encode integers, and operands, and their relation. Often, this is called an abstract syntax tree and is indeed a tree, but parsing is also usually done in many smaller steps. The critical situation with parsing making it an impure function is essentially the fact that the type of the output is not easily determined (is what we're reading an integer, a floating number, an operator, or something else?) and sometimes parsing is simply impossible if it doesn't fit any syntax we define. For example, if we define a language for addition, and then we decide to use operators unknown to this language, for example, the multiplication operator `*`, it's impossible to parse the string of characters. Furthermore, suppose we encounter a string `"21 + 45 * 7"`. We will be able to parse all the way until the integer 45, at which point we will fail to parse `*` and have to stop the parsing with a 'syntax error' message. So we need some way to detect and propagate errors, while keeping track of the parsing results and the remaining string to be parsed.

We define then the functor $P : \textbf{Hask} \to \textbf{Hask}$ which, to a type A maps the type $S \to M(A \times S)$. Here, S is the type `String`, and M is the `Maybe` monad discussed previously. The explanation is that a parser whose job is to find a value of type A in a string, will have to read in a string, and if it's able to find a value of type A, return that value along with the remaining string for further parsing. The unit and multiplication of the monad can be given as follows.

$$
\begin{aligned}
\eta_A :& A \to S \to M(A \times S) \\
& a \mapsto (s \mapsto (a, s)) \\
\mu_A :& S \to M(S \to M(A \times S) \times S) \to (S \to M(A \times S)) \\
& f \mapsto (s \mapsto (f(s) >\!\!>\!\!=_M ((p, s') \mapsto p(s'))))
\end{aligned}
$$

where $>\!\!>\!\!=_M$ is the bind for the `Maybe` monad.

The effect of the `Maybe` bind is to propagate syntax errors to the next potential parsers so that we stop parsing the rest of the string if we reach a syntax error at any point during the parsing. If all goes well, and we are able to parse, the join (multiplication) of the parser monad asks us to read the string and obtain a parser out of it which can then be applied to the rest of the string to finally parse an A.