1 Preamble

1.1 Natural transformation

We can start with the simple following relationship

$$\begin{tikzcd}
\mathcal{A} \arrow[r, shift left, "F"] \arrow[r, shift right, "G"'] & \mathcal{B}
\end{tikzcd}$$

Here we have two categories $\mathcal{A}$ and $\mathcal{B}$ and two functors $F, G : \mathcal{A} \to \mathcal{B}$. As a reminder a functor $\mathcal{F}$ is a mapping or simply a function between categories that maps object to object and morphism to morphism while preserving the structure of the category.
A question which someone might have when thinking about two different functors is, if we can have mappings between objects in a category i.e. morphisms, and if we can have mappings between categories i.e. functors, can we have mappings between functors?
An important commonality among the notion of maps is that they are structure preserving. When we speak of a *natural transformation* in category theory, we are speaking of a mapping between functors that preserves structure. Before we define a natural transformation, lets briefly touch on how functors are structure preserving.

1.1.1 Functors preserve structure

Formally a functor $F : \mathcal{A} \to \mathcal{B}$ is a mapping that ...

- associates to each object $X \in \mathcal{A}$ an object $F(X) \in \mathcal{B}$
- associates to each morphism $f : X \to Y$ in $\mathcal{A}$ a morphism $F(f) : F(X) \to F(Y)$ in $\mathcal{B}$

such that the following two properties hold:

- *Identity* - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every object X in $\mathcal{A}$
- *Composition* - $F(g \circ f) = F(g) \circ F(f)$ for all morphisms $f : X \to Y$ and $g : Y \to Z$ in $\mathcal{A}$

In other words the functor F must ensure the following diagram commutes.

Commute means that any path through the diagram with the same start and end point gives the same result. For example in the diagram below, we can start at X and go to Z directly via $f \circ g$ or we can go from X to Y via f and then from Y to Z via g. Both paths give the same result. And we can see that the same holds for the functor F.

$$\begin{tikzcd}
F(X) \arrow[rr, "f \circ g"] \arrow[dr, "F(f)"] & & F(Z) \\
& F(Y) \arrow[ur, "F(g)"] & \\
& Y \arrow[u, "F"] \arrow[dr, "g"] & \\
X \arrow[uuu, dashed] \arrow[ur, "f"] \arrow[rr, "f \circ g"] & & Z \arrow[uuu, dashed]
\end{tikzcd}
\quad \equiv \quad
\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "F"] & Y \arrow[d, "F"] \\
F_X \arrow[r, "F_f"] & F_Y
\end{tikzcd}$$

1.1.2 A transformation between functors

We can start to define a natural transformations by considering a mapping between two functors F and G, well call it α. Commonly we denote α as follows

$$\begin{tikzcd}
\mathcal{A} \arrow[r, bend left, "F"{name=U}] \arrow[r, bend right, "G"'{name=D}] & \mathcal{B} \arrow[Rightarrow, from=U, to=D, "\alpha"]
\end{tikzcd}$$

A question which arises then is what does α mean in more precise terms. Which brings us to the definition of a natural transformation. A *natural transformation* α from a functor $F : \mathcal{A} \to \mathcal{B}$ to a functor $G : \mathcal{A} \to \mathcal{B}$ is given by

- for every object X in $\mathcal{A}$ a morphism $\alpha_X : F(X) \to G(X)$ in $\mathcal{B}$
- such that for every morphism $f : X \to Y$ in $\mathcal{A}$ the following diagram - called the *naturality square* - commutes

$$\begin{tikzcd}
F_X \arrow[r, "\alpha_X"] \arrow[d, "F_f"] & G_X \arrow[d, "G_f"] \\
F_Y \arrow[r, "\alpha_Y"] & G_Y
\end{tikzcd}$$

Which is to say that we have

$$G(f) \circ \alpha_X = \alpha_Y \circ F(f)$$

1.1.3 Properties of natural transformations

A question which might appear for the case of natural transformations is do they have properties similar to those we see functors preserve for objects and morphisms? The answer is yes. A natural transformation $\alpha : F \to G$ has the following properties:

- *Identity* - There is an identity natural transformation $1_F : F \to F$.
- *Composition* - Given two natural transformations $\alpha : F \to G$ and $\beta : G \to H$ we can define their *vertical* composition for each $X \in \mathcal{C}$ by

  $$(\beta \circ \alpha)_X = \beta_X \circ \alpha_X$$

  Equivelantly we can define the following commutative diagram

$$\begin{tikzcd}
F_X \arrow[r, "\alpha_X"] \arrow[d] \arrow[rr, bend left, dashed, "(\beta\circ\alpha)_X"] & G_X \arrow[r, "\beta_X"] \arrow[d] & H_X \arrow[d] \\
F_Y \arrow[r, "\alpha_Y"] \arrow[rr, bend right, dashed, "(\beta\circ\alpha)_Y"'] & G_Y \arrow[r, "\beta_Y"] & H_Y
\end{tikzcd}
\quad \equiv \quad
\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "F"{name=F}] \arrow[r, "G"{name=G, description}] \arrow[r, bend right=50, "H"'{name=H}] & \mathcal{D} \arrow[Rightarrow, from=F, to=G, "\alpha"] \arrow[Rightarrow, from=G, to=H, "\beta"]
\end{tikzcd}$$

> An interesting thing to observe when we talk about natural transformations is that composition comes in two flavors. Vertical composition quite naturally arises from just thinking of composing natural transformations. But since functors themselves can be composed we also have to consider *horizontal* composition.

Since we are talking about natural transformations and composition we'll consider two pairs of functors with a natural transformation between each pair.

$$F, G : \mathcal{C} \to \mathcal{D} \quad \eta : F \to G \tag{1}$$

$$J, K : \mathcal{D} \to \mathcal{E} \quad \epsilon : J \to K \tag{2}$$

We can define the horizontal composition of η and ϵ as follows

$$\eta \star \epsilon : \underbrace{J \circ F}_{(\mathcal{C}\to\mathcal{D}\to\mathcal{E})} \to \underbrace{K \circ G}_{(\mathcal{C}\to\mathcal{D}\to\mathcal{E})}$$

With the following commutative diagram

$$\begin{tikzcd}
(J \circ F)_X \arrow[r, "J(\eta_X)"] \arrow[d, "(J\circ F)_f"'] & (J \circ G)_Y \arrow[r, "\epsilon_{G(X)}"] \arrow[d, "(J\circ G)_f"] & (K \circ G)_Y \arrow[d, "(K\circ G)_f"] \\
(J \circ F)_Y \arrow[r, "J(\eta_Y)"'] & (J \circ G)_Y \arrow[r, "\epsilon_{G(Y)}"'] & (K \circ G)_Y
\end{tikzcd}$$

Or the equivalent component wise definition for each $X \in \mathcal{C}$

$$(\eta \star \epsilon)_X = \epsilon_{G(X)} \circ J(\eta_X) = K(\eta_X) \circ \epsilon_{F(X)}$$

If you are a bit confused where these compositions are coming from then this should make it clearer how we constructed the above diagram.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\Downarrow \eta"] & \mathcal{D} \arrow[r, bend left, "J"] \arrow[r, bend right, "K"'] \arrow[r, phantom, "\Downarrow \epsilon"] & \mathcal{E}
\end{tikzcd}$$

If we for example look at the slice

$$(J \circ F)_X \xrightarrow{J(\eta_X)} (J \circ G)_X$$

It should be clear that the only path to get between the compositions is via the natural transformation η.

2 Monads

Formally we say that a monad is a triple (T, η, μ) where T is a functor and η and μ are natural transformations. More specifically

- $T : \mathcal{C} \to \mathcal{C}$ is a functor from a category to itself, also known as an *endofunctor*
- $\eta : 1_{\mathcal{C}} \to T$ is the natural transformation from the identity functor to T
- $\mu : T^2 \to T$ is the natural transformation from the double application of T to T

A nice way to understand what the two natural transformations are doing is that

- η *Unit* - this function takes an ordinary value and puts it into a monadic context. It is often referred to as *unit* or *return* in programming languages.
- μ *Multiplication* - this function represents the notation for flattening or merging two layers of a monadic structure into one. It is often referred to as *join* in programming languages.

There are two (or three) conditions that must hold for a monad. These are called the *monad laws* or more formally the *coherence conditions*. We can also start talking about haskell monads here.

- *Left Identity* - Left identity expresses that if we take a value, put it in a default context with η and then flatten it with μ, we get back the original value. Formally we write this as teh following composition

  $$\mu \circ \eta T = 1_T$$

  The analogous Haskell code would be

  ```haskell
  return a >>= f == f a -- return then bind with f is the same as just applying f to a
  ```

- *Right Identity* - Right identity expresses that if we have a value in a monadic context and we flatten it with μ after applying η, we get back the original value. Formally we write this as the following composition

  $$\mu \circ T\eta = 1_T$$

  The analogous Haskell code would be

```haskell
m >>= return == m -- bind with return is the same as just m
```

We can represent both the left and right identity laws with the following commutative diagram

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

- *Associativity* - Associativity expresses that when we have n layers of monadic context, it doesn't matter how we flatten them, the result will be the same. Formally we write this as the following composition

  $$\mu \circ \mu T = \mu \circ T\mu$$

  The analogous Haskell code would be

```haskell
(m >>= f) >>= g == m >>= (\x -> f x >>= g) -- order of binding does not matter
```

We can represent the associativity law with the following commutative diagram

$$\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

2.0.1 Kleisli category

A valid question you might have asked yourself with just the basic laws that come with monads is that they don't seem to be very indicative of something actually happening to a Monad. Which is to say that while the laws do give us the monadic structure, they don't really by themselves reveal the semantics of a monad in practice.

This is where the notion of a Kleisli category comes in. Formally a Kleisli category $\mathcal{C}_T$ for a monad (T, η, μ) on a category $\mathcal{C}$ is defined as follows:

- The objects of $\mathcal{C}_T$ are the same as the objects of $\mathcal{C}$, i.e.

  $$\text{Obj}(\mathcal{C}_T) = \text{Obj}(\mathcal{C})$$

- A morphism $f : X \to Y$ in $\mathcal{C}_T$ is a morphism $f : X \to T(Y)$ in $\mathcal{C}$ or expressed using hom-sets

  $$\text{Hom}_{\mathcal{C}_T}(X, Y) = \text{Hom}_{\mathcal{C}}(X, T(Y))$$

- Identity morphisms are given by $\eta_X : X \to T(X)$ for each object X in $\mathcal{C}$
- Composition of morphisms $f : X \to T(Y)$ and $g : Y \to T(Z)$ in $\mathcal{C}_T$ is given by the following composition

  $$g \circ_T f = \mu_Z \circ T(g) \circ f : X \to T(Y) \to T^2(Z) \to T(Z)$$

- The associativity and identity laws for composition in $\mathcal{C}_T$ follow from the monad laws for (T, η, μ)

A link we can draw here to Haskell is that a morphism in the Kleisli category is precisely the `pure` function. Which takes in a value of some type `a` and returns this value in a monadic context.

```haskell
pure :: a -> f a
```

Some examples

```
>>> pure 1 :: Maybe Int
Just 1
>>> pure 'z' :: [Char]
"z"
>>> pure (pure ":3") :: Maybe [String]
Just [":3"]
```

2.0.2 Bind as $\eta_X \circ_T f$

The Kleisli category gives us a way to understand how monads can be used to model computations. Specifically lets take a look at how the bind operator in Haskell can be understood using the Kleisli category. First we can look at the code for it

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

In plain english the operator takes a monadic value of type $m\ a$ and a function that takes a normal value of type a and returns a monadic value of type $m\ b$. The result is a monadic value of type $m\ b$.

What you might notice is that the type signature of the bind operator, especially the function part $(a \to m\ b)$, looks very similar to a morphism in the Kleisli category. Specifically the bind operator can be understood as us composing the identity morphism η_X with a morphism $f : X \to T(Y)$ in the Kleisli category. This is to say that we can express the bind operator as follows

$$\eta_X \circ_T f = \mu_Y \circ T(f) \circ \eta_X : X \to T(X) \to T^2(Y) \to T(Y)$$

Expanding the different parts of the composition we get

$$\underbrace{X \xrightarrow{\eta_X}}_{\texttt{lhs: m a}} \underbrace{T(X) \xrightarrow{T(f)} T^2(Y)}_{\texttt{rhs: a -> m b}} \underbrace{\xrightarrow{\mu_Y} T(Y)}_{\texttt{out: m b}}$$

So we can see that the bind operator is essentially just the composition of the identity morphism, which is to say the monad itself and a morphism in the Kleisli category. The nice part with this is we can apply the same notion to other monadic operators.
For completeness sake lets also look at the *then* operator where we don't care about the result of the first computation. The code for it is as follows

```
(>>) :: Monad m => m a -> m b -> m b
```

We use this operator in instances where we only care about the effects of the first computation, a basic example would be prompting for some input.

```
main :: IO ()
main = putStrLn "Enter ur name"
  >> getLine                          -- dont care about putStrLn result
  >>= \name ->                        -- do care about getLine result
      putStrLn ("Hello, " ++ name ++ "!")
```

2.0.3 Kleisli Composition as $g \circ_T f$

The aptly named Kleisli composition operator should hopefully be a quite intuitive now. Lets first take a look at the code for it again

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

We can already spot the two morphisms $f : X \to T(Y)$ and $g : Y \to T(Z)$ in the type signature. The operator takes two functions, each representing a morphism in the Kleisli category, and returns a new function that represents the composition of the two morphisms. We can break this down the same way as we did for the bind operator.

$$\underbrace{X \xrightarrow{f}}_{\texttt{lhs: (a -> m b)}} \underbrace{T(Y) \xrightarrow{T(g)} T^2(Z)}_{\texttt{rhs: (b -> m c)}} \underbrace{\xrightarrow{\mu_Z} T(Z)}_{\texttt{out: (a -> m c)}}$$

> Something important to note here is that an intrinsic property of monads is that when we compose them via Kleisli composition, we naturally apply some function to the internal state, but importantly the monad also executes the flattening operation μ.

Because we have the associativity law for monads, we also have associativity for Kleisli composition. This means that the order in which we compose Kleisli morphisms (monadic functions) does not matter. In Haskell this property is give via the Right-to-Left Kleisli composition operator

```haskell
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
```

2.1 Examples of monads

2.1.1 Maybe monad

So like with all monads we can understand them on a high level as having some representative function η and some manipulator function μ. Which satisfy the monad laws by how they are defined. Formally the maybe monad is defined as follows

- The functor $T : \mathbf{Set} \to \mathbf{Set}$ is defined on objects as

$$T(X) = (\,-\,)_* = X \cup \{\star\}$$

> (-) is a common notation used in category theory to just denote essentially a placeholder. So here we are saying that the functor takes a set X and maps it to the set X unioned with the singleton set containing the element $\star$. The element $\star$ is used to represent the absence of a value, or in Haskell terms `Nothing`.

- The representative natural transformation $\eta : 1_{\mathbf{Set}} \to T$ is defined component wise for each set X as

$$\eta_X : X \to T_X = x \mapsto x$$

- The flattening tranformation $\mu : T^2 \to T$ can be defined piecewise for each set X as

$$\mu_X : T^2(X) \to T(X) = \begin{cases} x & x \in X \\ \star & x = \star_1 \text{ or } x = \star_2 \end{cases}$$

> We can come to the above definition by considering what $T^2(X)$ is.
>
> $$\begin{aligned} T^2(X) &= T(T(X)) \\ &= T(X \cup \{\star\}) \\ &= (X \cup \{\star\}) \cup \{\star\} \\ &= X \cup \{\star_1, \star_2\} \end{aligned}$$
>
> The idea here being that after applying the functor twice we end up with a value that is either in X or is one of the two $\star$ values. The flattening function then just maps any value in X to itself and any of the $\star$ values to a single $\star$ value as they represent the same thing.

2.1.2 Writer monad

The writer monad is a monad that allows us to attach a log or some additional information to a computation. In simple terms its just a tuple of a value used for computations and some additional information. Formally the writer monad is defined as follows

- The functor $T : \mathbf{Set} \to \mathbf{Set}$ is defined on objects as

  $$T(X) = X \times M$$

  Where M is a monoid with an associative binary operation $\circ : M \times M \to M$ and an identity element $1 \in M$, we denote this as the triple $(M, \circ, 1)$.

  > A monoid is nothing more then a set equipped with an associative binary operation and an identity element. A common example of a monoid is the set of integers with addition as the binary operation and 0 as the identity element. Another example is the set of strings with concatenation as the binary operation and the empty string as the identity element.

- The representative natural transformation $\eta : 1_{\mathbf{Set}} \to T$ is defined component wise for each set X as

  $$\eta_X : X \to T(X) = x \mapsto (x, 1)$$

- The flattening transformation $\mu : T^2 \to T$ can be defined component wise for each set X as

  $$\mu_X : T^2(X) \to T(X) = ((x, m_1), m_2) \mapsto (x, m_1 \circ m_2)$$

  > Something to note here is that the flattening function μ is using the binary operation of the monoid. This is different from the case of the Maybe monad where our flattening function essentially was defined by the implicit fact that
  >
  > $$(X \cup \{\star\}) \cup \{\star\} \cong X \cup \{\star\}$$

Lets try and understand this a bit more in practice, first lets look at the type signature of the Haskell writer monad.

```haskell
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
  writer :: (a, w) -> m a
  tell   :: w -> m ()
  listen :: m a -> m (a, w)
  pass   :: m (a, w -> w) -> m a
```

So we can see the writer monad is parameterized by a type w which defines the *Monoid* describing the semantics of the 'log' store and the type m which is the monadic type constructor. The functional dependency `| m -> w` means that for a given monadic type constructor m there is only one possible monoid type w. Going through the different functions we start with `writer`. A question you might have here is how can we construct the MonadWriter if the only function inherent to this particular monad is

$$\texttt{return a} = \eta_X : X \to T(X) = x \mapsto (x, 1)$$

The answer lies in the fact that we can construct the writer method as a composition of η and `tell`. The function `tell` takes a value of type w i.e. a log Monoid value and returns a monadic value of type m (). Expressing this more formally we can define tell as

$$\texttt{tell} : M \to T(1) = m \mapsto ((), m)$$

So if we wish to define `writer` we must define it as the following composition

```haskell
writer (a, w) = tell w >> return a -- (rhs >> lhs) == (rhs >>= \_ -> lhs)
```

> Here we can see that we first use `tell` to create a monadic value with the log m and then we use the bind operator to sequence the computation with `return x` which is just our representative function η, so it puts the value x into the monadic context.

Something important to observe and remember here is that once again we actually have 3 things happening

1. Our tell function is creating a monadic value with the log m
2. Our return function is putting the value x into the monadic context
3. The bind operator is sequencing the two computations and applying the flattening function μ

The key point to remember that when we compose monadic functions there is always some implicit behavior happening in the background. In the case of the writer monad this implicit behavior is that the logs are being combined using the binary operation of the monoid.

2.1.3 State monad

The state monad is essentially just the analogue of a conventional function - i.e. something which takes an input, and returns an output - but with the addition of the fact that we also have some state that is ‘threaded through’ the computation. Formally the state monad is defined as follows

- The functor $T : \mathbf{Set} \to \mathbf{Set}$ is defined on objects as

$$T(X) : (X \times S)^S \cong S \to (S \times X) = s \mapsto (s, x)$$

- The transformation $\eta : 1_{\mathbf{Set}} \to T$ - which represents the stateful computation which just gives back the pure input value without modifying the state - is defined component wise for each set X as

$$\eta_X : X \to T(X) = x \mapsto (s \mapsto (x, s))$$

2.1.4 Currying adjunction

An important concept relevant for helping understand the state monad is the idea there is a certain level of "sameness" between functions that take multiple arguments and functions that take a single argument and return another function, which takes argument. Formally we can phrase it as follows

$$\mathbf{Set}(X \times S, Y) \cong \mathbf{Set}(X, Y^S) \equiv \mathrm{Hom}_{\mathbf{Set}}(X, Y^S) \cong \mathrm{Hom}_{\mathbf{Set}}(X \times S, Y)$$

If you prefer a diagrammatic representation we can express it as follows

$$\begin{tikzcd}
\mathbf{Set} \arrow[r, bend left, "- \times S"] & \mathbf{Set} \arrow[l, bend left, "(-)^S"]
\end{tikzcd}$$

> Intuitively the idea is that if I have a function $f(a, b) : A, B \to C$ I can ‘put off’ entering the second argument and just return a function $g(b) : B \to C$ i.e. the curried variant. It should be clear that both f and g are essentially the same function, just expressed in different ways.

What we would say formally here is given two categories $\mathcal{C}$ and $\mathcal{D}$ along with two functors $L : \mathcal{C} \to \mathcal{D}$ and $R : \mathcal{D} \to \mathcal{C}$ we say that L is left adjoint to R and R is right adjoint to L if there is a natural isomorphism $\alpha_{c,d}$ between hom-sets defined as follows

$$\alpha_{c,d} : \mathrm{Hom}_{\mathcal{C}}(c, R(d)) \cong \mathrm{Hom}_{\mathcal{D}}(L(c), d) \tag{3}$$

To denote an adjunction between two functors we write

$$L \dashv R$$

or in diagrammatic form

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

Additionally given a morphism $f : c \to R(d) \in \mathcal{C}$, its image $g = \alpha_{c,d}(f) : L(c) \to d \in \mathcal{D}$ is called the *mate* of f and vice versa. There are two special morphisms who’s mates are called the *unit* and *counit* of the adjunction. These morphisms are defined as follows

- The *unit* $\eta : 1_C \to R \circ L$ (component wise $\eta_c : c \to R(L(c))$) is the mate of the identity morphism $1_{L(c)} : L(c) \to L(c)$

> So here the unit η_c is just $\alpha_{c,L(c)}(1_{L(c)})$ i.e. the mate of the identity morphism
>
> $$1_{L(c)} : L(c) \to L(c) = (\epsilon \circ L) \circ (L \circ \eta) = 1_L$$
>
> Intuitively you can think of the unit as taking an object and putting it into the context of the adjunction via the functor L and then bringing it back to the original category via the functor R.

- The *counit* $\epsilon : L \circ R \to 1_D$ (component wise $\epsilon_d : L(R(d)) \to d$) is the mate of the identity morphism $1_{R(d)} : R(d) \to R(d)$

Similarly here the counit ϵ_d is just $\alpha^{-1}_{R(d),d}(1_{R(d)})$ i.e. the mate of the identity morphism $1_{R(d)}$.

$$1_{R(d)} : R(d) \to R(d) = (R \circ \epsilon) \circ (\eta \circ R) = 1_R$$

Here the idea is that if you forget the structure of $d \in \mathcal{D}$ and bring it back to $\mathcal{C}$ via the functor R and then bring it back to $\mathcal{D}$ via the functor L, you get a canonical simplication back to d.

Finally for something to be an adjunction the unit and counit must satisfy the following two triangle identities

$$\begin{tikzcd}
L \arrow[r, "L\circ\eta"] \arrow[dr, "id"'] & L \circ R \circ L \arrow[d, "\epsilon\circ L"] \\
 & L
\end{tikzcd}
\qquad
\begin{tikzcd}
R \arrow[r, "\eta\circ R"] \arrow[dr, "id"'] & R \circ L \circ R \arrow[d, "R\circ\epsilon"] \\
 & R
\end{tikzcd}$$

Going back to our example of currying we can see that the functor $- \times S$ is left adjoint to the functor $(-)^S$. Formally

$$- \times S \dashv (-)^S$$

With the unit and counit defined as follows

- The unit $\eta : 1_{\mathbf{Set}} \to (-)^S \circ (- \times S)$ is defined component wise for each set X as

$$\eta_X : X \to (X \times S)^S = x \mapsto (s \mapsto (x, s))$$

- The counit $\epsilon : (- \times S) \circ (-)^S \to 1_{\mathbf{Set}}$ is defined component wise for each set Y as

$$\epsilon_Y : (Y^S \times S) \to Y = (f, s) \mapsto f(s)$$

The triangle identities for this particular adjunction can be expressed as follows

- *Left triangle identity*

$$(R \circ \epsilon) \circ (\eta \circ R) = 1_R$$

  For each set Y this means that the following diagram commutes

$$Y^S \xrightarrow{\eta_{Y^S}} (Y^S \times S)^S \xrightarrow{R_{\epsilon_Y}} Y^S$$

  The identity 1_{Y^S} is just the function that takes a function $f : S \to Y$ and returns the same function $f : S \to Y$.

To see why this is the case we can expand the two functions in the composition. First we have

$$\eta_{Y^S} : Y^S \to (Y^S \times S)^S = f \mapsto (s \mapsto (f, s))$$

Then we have

$$R_{\epsilon_Y} : (Y^S \times S)^S \to Y^S = g \mapsto (s \mapsto \epsilon_Y(g(s))) = g \mapsto (s \mapsto f)$$

So if we take a function $f : S \to Y$ and apply the composition we get

$$f \xrightarrow{\eta_{Y^S}} (s \mapsto (f, s)) \xrightarrow{R_{\epsilon_Y}} (s \mapsto f) = f$$

The most straightforward intuition here being η_{Y^S} is just currying the function f and R_{ϵ_Y} is just uncurrying it.

- *Right triangle identity*

$$(\epsilon \circ L) \circ (L \circ \eta) = 1_L$$

  For each set X this means that the following diagram commutes

$$X \times S \xrightarrow{L_{\eta_X}} (X \times S)^S \times S \xrightarrow{\epsilon_{X \times S}} X \times S$$

  The identity $1_{X \times S}$ is just the function that takes a pair (x, s) and returns the same pair (x, s).

To see why this is the case we can expand the two functions in the composition. First we have

$$L_{\eta_X} : X \times S \to (X \times S)^S \times S = (x, s) \mapsto (\underbrace{(s' \mapsto (x, s'))}_{g}, s)$$

Then we have

$$\epsilon_{X \times S} : (X \times S)^S \times S \to X \times S = (g, s) \mapsto g(s) = \underbrace{(s' \mapsto (x, s'))(s) = (x, s)}_{\equiv(\lambda s'.(x,s'))(s)=(x,s)}$$

So if we take a pair (x, s) and apply the composition we get

$$(x, s) \xrightarrow{L_{\eta_X}} ((s' \mapsto (x, s')), s) \xrightarrow{\epsilon_{X \times S}} (x, s)$$

Similarly to the left triangle identity the most straightforward intuition here being L_{η_X} is just currying the identity $1_{X \times S}$ and $\epsilon_{X \times S}$ is just uncurrying it.

2.1.5 State monad from the currying adjunction

An interesting fact when you have an adjunction $L \dashv R$ is that you automatically get a monad $R \circ L$ (and a comonad $L \circ R$). Formally we would say that the adjunction $L \dashv R$ induces the monad $R \circ L$. We can state this as the following proposition.
Given a pair of adjoint functors

$$L \dashv R \quad \text{where} \quad L : \mathcal{C} \to \mathcal{D}, R : \mathcal{D} \to \mathcal{C}$$

And the unit and counit of the adjunction

$$\eta : 1_{\mathcal{C}} \to R \circ L \quad \text{and} \quad \epsilon : L \circ R \to 1_{\mathcal{D}}$$

Or equivelantly component wise for each object $c \in \mathcal{C}$ and $d \in \mathcal{D}$

$$\eta_c : c \to R(L(c)) \quad \text{and} \quad \epsilon_d : L(R(d)) \to d$$

Then $(R \circ L, \eta, \mu)$ is a monad on $\mathcal{C}$ where the unit η and multiplcation μ are defined as follows

- The unit $\eta : 1_{\mathcal{C}} \to R \circ L$ is the unit of the adjunction
- The multiplication $\mu : (R \circ L)^2 \to R \circ L$ is defined by the following composition

$$\mu = R(\epsilon_L) : (R \circ L)^2 \to R \circ L \quad \epsilon_L : L \circ R \circ L \to L$$

To prove that this is indeed a monad as a reminder we have to be able to show that the following diagrams commute

$$\begin{tikzcd}
(L \circ R)^3 \arrow[r, "(L\circ R)\circ\mu"] \arrow[d, "(L\circ R)\circ\eta"'] & (L \circ R)^2 \arrow[d, "\mu"] \\
(L \circ R)^2 \arrow[r, "\mu"'] & (L \circ R)
\end{tikzcd}
\qquad
\begin{tikzcd}
(L \circ R) \arrow[r, "\eta\circ(L\circ R)"] \arrow[d, "(L\circ R)\circ\eta"'] & (L \circ R)^2 \arrow[d, "\mu"] \\
(L \circ R)^2 \arrow[r, "\mu"'] & (L \circ R)
\end{tikzcd}$$

- *Left & Right identity* - $\mu \circ (T \circ \eta) = 1_T = \mu \circ (\eta \circ T)$

$$\begin{aligned}
\mu \circ (T \circ \eta) &= R(\epsilon_L) \circ ((R \circ L) \circ \eta) \\
&= R(\epsilon_L) \circ R(L \circ \eta) \\
&= R(\epsilon_L \circ \eta_L) \quad \eta_L : L \to R \circ L^2 \\
&= R(1_L) \\
&= 1_{R\circ L} = 1_T \\
\mu \circ (\eta \circ T) &= R(\epsilon_L) \circ (\eta \circ (R \circ L)) \\
&= R(\epsilon_L) \circ (\eta_{R\circ L}) \\
&= 1_{R\circ L} = 1_T
\end{aligned}$$

We can show that both the left and right identity laws hold by showing that the following diagram commutes

$$\begin{tikzcd}
 & (R \circ L)^2_A \arrow[dr, "R(\epsilon_L)_A \equiv \mu_A"] & \\
(R \circ L)_A \arrow[ur, "(R \circ L) \circ \eta_A"] \arrow[rr, "\text{id}"] \arrow[dr, "\eta \circ (R \circ L)_A"'] & & (R \circ L)_A \\
 & (R \circ L)^2_A \arrow[ur, "R(\epsilon_L)_A \equiv \mu_A"'] &
\end{tikzcd}
\equiv
\begin{tikzcd}
T \arrow[r, "\eta \circ T"] \arrow[d, "T \circ \eta"'] \arrow[dr, "\text{id}"] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

All we did here was for the upper part of the diagram we took the right triangle identity of the adjunction and applied the functor R to it. For the lower part of the diagram we took the left triangle identity of the adjunction for each component B on which R acts we substituted $B = L_A$.

- *Associativity* - $\mu \circ (\mu \circ T) = \mu \circ (T \circ \mu)$

$$\begin{aligned}
\mu \circ (\mu \circ T) &= R(\epsilon_L) \circ (R(\epsilon_L) \circ (R \circ L)) \\
&= R(\epsilon_L) \circ R(\epsilon_L \circ L) \\
&= R(\epsilon_L \circ \epsilon_{L \circ R}) \quad \epsilon_{L \circ R} : L \circ R^2 \to L \circ R \\
&= R(\epsilon_L) \circ R(\epsilon_{L \circ R}) \\
&= R(\epsilon_L) \circ ((R \circ L) \circ R(\epsilon_L)) \\
&= \mu \circ (T \circ \mu)
\end{aligned}$$

Similarly to the identity laws we can leverage existing commutative diagrams and preform some substitutions to show that the associativity law holds.

$$\begin{tikzcd}
(L \circ R)_A \arrow[r, "\epsilon_A"] \arrow[d, "(L \circ R)_f"'] & A \arrow[d, "f"] \\
(L \circ R)_{A'} \arrow[r, "\epsilon_{A'}"'] & A'
\end{tikzcd}
\qquad
\begin{tikzcd}
(R \circ L)^3 \arrow[r, "R(\epsilon_{L \circ R \circ L})"] \arrow[d, "(R \circ L \circ R) \circ (\epsilon_L)"'] & (R \circ L)^2 \arrow[d, "R(\epsilon_L)"] \\
(R \circ L)^2 \arrow[r, "R(\epsilon_L)"'] & R \circ L
\end{tikzcd}$$

The diagram on the left here is just the naturality square for the counit ϵ. The diagram on the right is generated by taking the naturality square, substituting $f = \epsilon_L$ then applying the functor R to the entire diagram. Note here that

$$(R \circ L \circ R) \circ (\epsilon_L) = (R \circ L) \circ R(\epsilon_L)$$

And similarly

$$R(\epsilon_{L \circ R \circ L}) = R(\epsilon_L) \circ (R \circ L)$$

The nice thing we can do now is see that the state monad is literally just the monad induced by the curring adjunction. So we have the adjunction

$$S \times - \dashv (-)^S$$

Composing the two functors we get the monad endofunctor

$$T = (-)^S \circ (S \times -) : \textbf{Set} \to \textbf{Set}$$

Component wise for each set X we have

$$T(X) = (X \times S)^S \cong S \to (S \times X)$$

Then the multiplication $\mu : ((-)^S \circ (S \times -))^2 \to (-)^S \circ (S \times -)$ is defined by the following composition

$$\mu = (-)^S(\epsilon_{S\times -}) : ((-)^S \circ (S \times -))^2 \to (-)^S \circ (S \times -)$$

So component wise for each set X we have

$$\mu_X = (-)^S(\epsilon_{X \times S}) : ((X \times S)^S \times S)^S \to (X \times S)^S$$

Something very important to recognize here is how the flattening function μ is defined via the counit of the adjunction. As a reminder the counit $\epsilon : (S \times -) \circ (-)^S \to 1_{\mathbf{Set}}$ provides us a way of essentially forgetting - or concretelty in this case 'uncurrying' - the structure of the adjunction.

$$e_{X\times S} : \ldots = (s' \mapsto (x, s'))(s) = (x, s)$$

So naturally then if we want to do this 'forgetting' operation but *inside* of the context of the adjunction we have to apply the functor $(-)^S$ to the counit. This is what gives us the flattening function μ.

To give a more granular instance of the component wise behavior of μ lets assume an input function $f : S \to (X \times S)^S \times S$. So for an input state $s \in S$ the function f returns a pair consisting of a function $g : S \to X \times S$ and a state $s' \in S$.

$$\begin{aligned} \mu_X(f)(s) &= \epsilon_{X\times S}(f(s)) \\ &= \epsilon_{X\times S}((g, s')) \quad \text{where} \quad (g, s') : (X \times S)^S \times S \\ &= g(s') \\ &= (s' \mapsto (x, s''))(s') = (x, s'') \end{aligned}$$

The high level intuition what we can take away from this is that adjoint functors informally can be thought of as a way of taking something from a base context and putting it into a structure context and then having a way of forgetting that structure and bringing it back to the base context.
Naturally then when we combine or more accurately compose these two functors, we get a functor which represents the simple combination of these ideas - i.e. putting something into a structured context and then forgetting that structure and bringing it back to the base context. Along with this combined monadic functor we then naturally also get answers to the two main questions

- *How do we put something into this structured context?*
  This is answered by the unit of the adjunction
- *What if we arent starting from the base context?*
  This is answered by the multiplication function, defined via the counit of the adjunction

2.1.6 Whiskering

Whiskering can be understood as the higher dimensional analogue pre and post-composition of morphisms. Below is a diagrammatic representation of how we can understand whiskering

$$\begin{tikzcd} C \arrow[r, "F"] & D \arrow[r, bend left=50, "G", ""{name=U, below}] \arrow[r, bend right=50, "H"', ""{name=L, above}] & E \arrow[from=U, to=L, Rightarrow, "\eta"'] \end{tikzcd} \qquad \begin{tikzcd} C \arrow[r, bend left=50, "F", ""{name=U, below}] \arrow[r, bend right=50, "G"', ""{name=L, above}] & D \arrow[r, "H"] & E \arrow[from=U, to=L, Rightarrow, "\eta"'] \end{tikzcd}$$

- *Left whiskering* - Given functors $F : \mathcal{C} \to \mathcal{D}$, $G, H : \mathcal{D} \to \mathcal{E}$ and a natural transformation $\eta : G \to H$ we can define the left whiskering of η by F as the natural transformation

$$F \circ \eta : G \circ F \to H \circ F \equiv (F\eta)_c = \eta_{F(c)} : G(F(c)) \to H(F(c)) \tag{4}$$

- *Right whiskering* - Given functors $F, G : \mathcal{C} \to \mathcal{D}$, $H : \mathcal{D} \to \mathcal{E}$ and a natural transformation $\eta : F \to G$ we can define the right whiskering of η by H as the natural transformation

$$\eta \circ H : H \circ F \to H \circ G \equiv (\eta H)_c = H(\eta_c) : H(F(c)) \to H(G(c)) \tag{5}$$

3 Functoral semantics

As a reminder, a functor is simply a structure preserving map or function between two categories. To relate this to the contet of monads we also have to remember that a monad itself is just a functor - specifically a functor from a category to itself - along with some additional structure.

3.1 Functor map

Within haskell the mapping functors provide is represented by the `fmap` function. The type signature of `fmap` is as follows

```haskell
fmap :: (a -> b) -> f a -> f b
```

So we can see that `fmap` takes a function from `a -> b` - different from the monadic bind operator which takes a function from `a -> m b` - and a functorial value of type `f a` and returns a functorial value of type `f b`. As with various other constructs there is also the commonly used infix operator `<$>` which is just an alias for `fmap`.
Some basic examples of using `fmap` are as follows

```haskell
-- List functor
fmap (+1) [1,2,3] -- [2,3,4]

-- Maybe functor
fmap (+1) (Just 1) -- Just 2
fmap show (Just 1) -- Just "1" :: Maybe String
```

3.2 Lax functors

Something which you might have already noticed as a pattern within category theory is that we often move up dimensions when we generalize concepts. For example we had

- Objects
- Maps between objects (morphisms)
- Maps between categories (functors)
- Maps between functors (natural transformations)

A concept which was used in some previous definitions is the idea of a hom-set - the set of morphisms between two objects in a category. It's important to note that the reason we are saying hom-*set* and not hom-*object* is because the 'default' category we are generally working with is one enriched in the monoidal category of sets. The idea of enrichment is typically described as a way of generalizing categories by allowing their hom-sets to be objects in different monoidal categories.

> The idea is that monoidal categories generally represent some kind of structure - for example the category of sets with the cartesian product as the tensor product represents the structure of having multiple values. So when we are talking about enrichment we are parametrizing the dynamics of our category by the structure represented by the monoidal category we are enriching in.

We can say that given a monoidal category $(\mathcal{V}, \otimes, I)$ a $\mathcal{V}$-enriched category $\mathcal{C}$ consists of the following data

- A collection of objects
- For each pair of objects $A, B \in \mathcal{C}$ a hom-object $\mathcal{C}(A, B) \in \mathcal{V}$
- For each object $A \in \mathcal{C}$ a morphism in $\mathcal{V}$ known as the identity map

$$I \to \mathcal{C}(A, A)$$

- For each triple of objects $A, B, C \in \mathcal{C}$ a morphism in $\mathcal{V}$ known as the composition map

$$\mathcal{C}(B, C) \otimes \mathcal{C}(A, B) \to \mathcal{C}(A, C)$$

- The identity and composition maps must satisfy the usual associativity and identity laws

When we take $\mathcal{V} = \textbf{Set}$ we get the usual definition of a category. To get back to the idea of generalizing we can similarly generalize the notion of enrichment to not just categories but also categories of categories. The most classical example of 'categories of categories' is **Cat** - the category of small categories [1] - also as the 2-category of categories. Here we have

- Objects - small categories
- 1-morphisms - functors between small categories
- 2-morphisms - natural transformations between functors

We can then asked the same question here, what does it mean to enrich this 'default' 2-category of categories? Since we are now working within 2-categories we are enriching in monoidal 2-categories (specifically monoidal bicategories).

> As a reminder when we ask this question of enrichment we are really just asking how we can define a more general descriptor of something, in this case n-categories.

We can define a bicategory $\mathcal{B}$ as consisting of the following data

- (Objects) - A collection of objects (0-cells)
- (Morphisms) - For each pair of objects (0-cells) $A, B \in \mathcal{B}$ a hom-category $\mathcal{B}(A, B)$
  - Objects of the hom-category are called 1-morphisms or 1-cells
  - Morphisms of the hom-category are called 2-morphisms or 2-cells

  > We can see here that the hom-sets have been generalized to hom-categories.

- (Identity) - For each 0-cell $A \in \mathcal{B}$ a *distinguished 1-cell* $1_A \in \mathcal{B}(A, A)$ also known as the identity 1-cell or identity morphism
- (Composition) - For each triple of 0-cells $A, B, C \in \mathcal{B}$ the horizontal composition functor

  $$\circ : \mathcal{B}(B, C) \times \mathcal{B}(A, B) \to \mathcal{B}(A, C)$$

- (Unitality) - For each pair of 0-cells $A, B \in \mathcal{B}$ natural isomorphisms known as the left and right unitors

  $$\left(R_A = \begin{cases} f \mapsto f \circ 1_A \\ \theta \mapsto \theta \circ 1_{1_A} \end{cases} \right) \cong \left(L_B = \begin{cases} f \mapsto 1_B \circ f \\ \theta \mapsto 1_{1_B} \circ \theta \end{cases} \right)$$

  Both of which are naturally isomorphic to the identity functor on $\mathcal{B}(A, B)$

  $$\mathrm{id}_{\mathcal{B}(A,\ B)} \ : \ \mathcal{B}(A,\ B) \to \mathcal{B}(A,\ B)$$

  > While this might look a bit confusing at first all we are saying here is that if we take any 1-cell $f \in \mathcal{B}(A, B)$ and compose it with the identity 1-cell on either side we get something which is naturally isomorphic to f. The same applies for 2-cells $\theta : f \to g$ where $f, g \in \mathcal{B}(A, B)$.

- (Associativity) - For each quadruple of 0-cells $A, B, C, D \in \mathcal{B}$ a natural isomorphism known as the associator
- (Coherence laws) - The left and right unitors and associator must satisfy the usual coherence laws (pentagon and triangle identities)

Before we start exploring functors between two categories there is an important idea I want to talk about to help with the intuition of some of the things we are going to introduce later.

[1] The restriction to small is to avoid issues related to Russels paradox

3.3 Profunctors

Something to observe in the case of binary relations is that by their vary nature they relate two things by a yes or no answer. But the fact is that things can obviously be related in much more complex ways, such as distance, cost, probability etc. This idea of generalizing or parametrizing relations over different kinds (as opposed to just a boolean yes or no) of relationships is what gives rise to profunctors.

Before defining general profunctors lets start with the concrete instance of a **Bool**-profunctor. In the context of preorders.

A preorder is just a set with a binary relation which is reflexive and transitive. A basic example being the set of natural numbers with the less than or equal to relation, denote as $(\mathbb{N}, \leqslant)$.

Given two preorders $\mathcal{X} = (X, \leqslant_X)$ and $\mathcal{Y} = (Y, \leqslant_Y)$ a **Bool**-profunctor can be expressed as the following monotone map

$$\phi : \mathcal{X}^{\text{op}} \times \mathcal{Y} \to \textbf{Bool}$$

The reason we are using the opposite category $\mathcal{X}^{\text{op}} = (X, \geqslant_X)$ here is to ensure that the monotonicity condition holds. A monotone map is a function $f : A \to B$ between two preorders $\mathcal{A} = (A, \leqslant_A)$ and $\mathcal{B} = (B, \leqslant_B)$ such that for all $a_1, a_2 \in A$

$$a_1 \leqslant_A a_2 \implies f(a_1) \leqslant_B f(a_2)$$

Intuitively it preserves the ordering (or structure) of the preorders under the mapping. The mapping itself just associates elements of A to elements of B.

Commonly this is written as

$$\phi : \mathcal{X} \nrightarrow \mathcal{Y}$$

Given $x \in X$ and $y \in Y$ if $\phi(x, y) = \texttt{true}$ we can say that x can be obtained from y or that x is related to y. The monotonicity condition in this instance means that

$$(x' \leqslant_X x) \land (y \leqslant_Y y') \implies \phi(x, y) \leqslant_{\textbf{Bool}} \phi(x', y')$$

So if x can be obtained from y and x' is available given x, then x' can also be obtained from y. Similarly if y' is available given y and x can be obtained from y, then x can also be obtained from y'.

An alternative way of phrasing the monotonicity condition in this case is that it describes how the relationship represented by the profunctor changes as we move along the preorders (or the arrows) in either direction. First we consider $x' \to x \in \mathcal{X} \iff x' \leqslant x$ here we are saying that if we have an arrow

$$x \to x' \in \mathcal{X}^{\text{op}} \equiv x' \leqslant x \in \mathcal{X}$$

Then we have that

$$\phi(x, y) \leqslant_{\textbf{Bool}} \phi(x', y) \in \mathbf{2}$$

Equivelantly

$$\phi(x, y) = \top \implies \phi(x', y) = \top$$

The idea here being that if x is related to y then x' which is 'less than' x is also related to y. This is understood to be the contravariant or 'pullback' direction as it goes against the direction of the arrows in $\mathcal{X}$.
Similarly for the other direction we consider $y \to y' \in \mathcal{Y} \iff y \leqslant y' \in \mathcal{Y}$ here we are saying that if we have an arrow

$$y \to y' \in \mathcal{Y} \equiv y \leqslant y' \in \mathcal{Y}$$

Then we have that

$$\phi(x, y) \leqslant_{\textbf{Bool}} \phi(x, y') \in \mathbf{2}$$

Equivelantly

$$\phi(x, y) = \top \implies \phi(x, y') = \top$$

The idea here being that if x is related to y then x which is 'greater than' y is also related to y'. This is understood to be the covariant or 'pushforward' direction as it goes with the direction of the arrows in $\mathcal{Y}$.

We can also create a more diagrammatic representation of the above notion as follows

$$\begin{tikzcd}
x' \arrow[r, "f=x'\leqslant x"] \arrow[rr, dashed, bend right, "f\circ\phi"'] & x \arrow[r, "\phi"] \arrow[rr, dashed, bend left, "\phi\circ g"] & y \arrow[r, "g=y\leqslant y'"'] & y'
\end{tikzcd}$$

Again to emphasize the high level idea of this is that we can decompose the dynamics of a relationship into two directions - a contravariant direction and a covariant direction. Starting with the contravariant direction

$$f \circ \phi = \{(x', y) \mid f(x') \leqslant y\}$$

Here we are saying that if we have a way of obtaining x from x' and a way of obtaining x from y then we also have a way of obtaining x' from y. Simply following the arrows should make the idea somewhat trivially apparent.

$$\phi \circ g = \{(x, y') \mid \exists y, (x \leqslant y) \land g(y) = y'\}$$

We observe the same idea here, if we have a way of obtaining y' from y and a way of obtaining x from y then we also have a way of obtaining x from y'. Again this is enabled through the trivial observation that we clearly have a path from x to y' by following the arrows.

The most common type of profunctor you will see is the **Set**-profunctor. Given two categories $\mathcal{C}$ and $\mathcal{D}$ a **Set**-profunctor is defined as the following functor

$$\phi : \mathcal{C}^{op} \times \mathcal{D} \to \textbf{Set}$$

Here as opposed to relating elements of two preorders with a boolean yes or no answer we are now relating objects of two categories with a set of possible relationships. Given $c \in \mathcal{C}$ and $d \in \mathcal{D}$ the set $\phi(c, d)$ can be thought of as the set of ways in which c is related to d. A nice way to understand these types of profunctors is in terms of how they relate to adjunctions. As a reminder we have an adjunction $L \vdash R$ between two categories $\mathcal{C}$ and $\mathcal{D}$ if there is a natural isomorphism $\alpha_{c,d}$ between the following hom-sets

$$\alpha_{c,d} : \mathcal{D}(L(c), d) \xrightarrow{\cong} \mathcal{C}(c, R(d))$$

That is natural in both c and d. Something *being natural* in general means that it is compatible with the morphisms in the categories. So for any morphism $f : c' \to c$ in $\mathcal{C}^{op}$ and any morphism $g : d \to d'$ in $\mathcal{D}$ the following diagrams commute

$$\begin{tikzcd}
\mathcal{C}(c, R_d) \arrow[r, "\alpha_{c,d}"] \arrow[d, "\mathcal{C}(f{,}R_g)"'] & \mathcal{D}(L_c, d) \arrow[d, "\mathcal{D}(L_f{,}g)"] \\
\mathcal{C}(c', R_{d'}) \arrow[r, "\alpha_{c',d'}"'] & \mathcal{D}(L_{c'}, d')
\end{tikzcd}$$

Something to observe here is that this is a naturality between functors of the form $\mathcal{C}^{op} \times \mathcal{D} \to \textbf{Set}$, in other words, profunctors. Here f is our contravariant direction and g is our covariant direction. The implication of this fact is that every adjunction, so there being an isomorphism between two hom-sets, gives rise or induces two profunctors

$$\mathcal{D}(L(-), -) : \mathcal{C}^{op} \times \mathcal{D} \to \textbf{Set}$$
$$\mathcal{C}(-, R(-)) : \mathcal{C}^{op} \times \mathcal{D} \to \textbf{Set}$$

In plain english the natural isomorphism between the two hom-sets is saying that for every object $c \in \mathcal{C}$ and $d \in \mathcal{D}$ there is a one to one correspondence between the set of ways in which L can map c to d and the set of ways in which R can map d to c.

The set of ways in which things can be related is exactly what profunctors describe. Hence an adjunction in the weakest sense (i.e. just the natural isomorphism between the two hom-sets) is simply the statement that the ways in which L relates objects in $\mathcal{C}$ to objects in $\mathcal{D}$ is the same as the ways in which R relates objects in $\mathcal{D}$ to objects in $\mathcal{C}$.

Importantly we can contrast this with the idea of an adjoint equivalence. An adjoint equivalence is an adjunction $L \vdash R$ such that the unit and counit of the adjunction are natural isomorphisms. Making the unit and counit natural isos means that embedding (unit) and forgetting (counit) structure is done in a way which preserves all the information, in other words we have an object level symmetry or equivalence between the two categories. Something which you might then recognize here is that an adjoint equivalence definitionally gives rise to an equivalence of categories.

> A nice way of understanding this is that we are talking about comparisons at different levels of structure or abstraction. An adjunction is a comparison at the level of relationships between objects, i.e. profunctors, while an adjoint equivalence is a comparison at the level of the objects themselves.

3.3.1 Composition

Before we talk about how profunctors are composed we need to learn to talk about the idea of of pullbacks and pushouts and their generalizations. Well start with pushouts as they are generally a bit more interesting. In the classical **Set**-enriched case a pushout is the colimit of a diagram consisting of two morphisms with a common domain. The intuition behind a pushout is often described as a way of 'gluing' objects among a common part. Alternatively we can say that a pushout a merges two objects which have the same source. The classical diagram for a pushout is as follows

$$\begin{tikzcd}
A \arrow[r, "f_1"] \arrow[d, "f_2"'] & B \arrow[d, "q_1"] \arrow[ddr, bend left, "p_1"] & \\
C \arrow[r, "q_2"'] \arrow[drr, bend right, "p_2"] & B \sqcup_A C \arrow[dr, dashed, "!k"] & \\
& & D
\end{tikzcd}$$

We can also draw a more compressed but equivalent variant as follows

$$\begin{tikzcd}
A \arrow[r, shift left, "q_1 \circ f_1"] \arrow[r, shift right, "q_2 \circ f_2"'] & B \sqcup_A C \arrow[r, "(p_1{,}p_2)"] & D
\end{tikzcd}$$

A question you can ask here is, do the parallel arrows we see have to specifically be projections into a coproduct? Or alternatively we can ask what the general phenomena which is being described here. The answer to these questions is what leads us to coequalizers.

A coequalizer is nothing more then the generalization of the idea of quotienting by an equivalence relation. In general we would say that given the parallel arrows

$$\begin{tikzcd}
X \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & Y
\end{tikzcd}$$

We can form a quotented set $Y/\sim$ where the equivalence relation $\sim$ is generated by the pairs

$$\{(f(x), g(x)) \mid x \in X\} \equiv f(x) \sim g(x) \quad \forall x \in X$$

The idea being that the quotented set $Y/\sim$ is formed by identifying elements of Y which are related by the equivalence relation $\sim$. So if we have $y, y' \in Y$ such that $y \sim y'$ then in the quotented set $Y/\sim$ we have that $[y] = [y']$.

> In the case of a pushout we have the set $B \sqcup_A C$ being formed by
>
> $$(q_1 \circ f_1(a)) \sim (q_2 \circ f_2(a)) \quad \forall a \in A$$

The quotient function $p : Y \to Y/\sim$ then satisfies the following

$$p \circ f = p \circ g$$

> Our quotient function $p : B \sqcup C \to B \sqcup_A C$ then satisfies the following
>
> $$p \circ (q_1 \circ f_1) = p \circ (q_2 \circ f_2)$$

Finally we have that p is universal with respect to this property. Meaning that for any other function $h : Y \to Z$ such that

$$h \circ f = h \circ g$$

There is a unique function $k : Y/\sim \to Z$ such that

$$h = k \circ p$$

For our pushout we have that for any other function $h : B \sqcup C \to D$ such that

$$h \circ (q_1 \circ f_1) = h \circ (q_2 \circ f_2)$$

There is a unique function $!k : B \sqcup_A C \to D$ such that

$$h =!k \circ p$$

A coend is a generalization of the idea of a colimit. Specifically it is a colimit over a functor of the form $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{D}$. Given such a functor we can define the coend of F as follows

$$\coprod_{f:c\to c'} F(c', c) \rightrightarrows \coprod_{c} F(c, c) \equiv \int^{c\in\mathcal{C}} F(c, c)$$

Since this is a bit abstract lets take the concrete instance of our **Set**-profunctor $\phi : \mathcal{C}^{op} \times \mathcal{D} \to \mathbf{Set}$. We can start with defining what the coproduct means in Set. Fromally we would say that the coproduct of a family of sets $\{X_i\}_{i\in I}$ is the disjoint union given by

$$\coprod_{i\in I} X_i = \{(x, i) \mid x \in X_i, i \in I\}$$

So it represents the collection of all the elements in all the sets in the family while also keeping track of which set each element came from. More generally we would say that given a functor $F : \mathcal{J} \to \mathbf{Set}$ the coproduct of F is given by

$$\coprod_{j\in\mathcal{J}} F(j) = \{(x, j) \mid x \in F(j), j \in \mathcal{J}\}$$

equipped with maps

$$i_j : F(j) \to \coprod_{j'\in\mathcal{J}} F(j') \quad \forall j \in \mathcal{J}$$

Meaning that a given map i_j takes an element $x \in F(j)$ and maps it to the corresponding element (x, j) in the coproduct. The coproduct is then universal with respect to this property. If we then go back to our profunctor $\phi : \mathcal{C}^{op} \times \mathcal{D} \to \mathbf{Set}$ we define our coend as coequalizers over $c \in \mathcal{C}$ and $d \in \mathcal{D}$ as follows

$$\int^{c\in\mathcal{C}} \phi(c, d) = \text{coeq}\left(\coprod_{f:c\to c'} \phi(c', d) \rightrightarrows \coprod_{c} \phi(c, d)\right) \tag{6}$$

$$\int^{d\in\mathcal{D}} \phi(c, d) = \text{coeq}\left(\coprod_{g:d\to d'} \phi(c, d') \rightrightarrows \coprod_{d} \phi(c, d')\right) \tag{7}$$

With the respective quotient maps

$$p_c : \coprod_{c} \phi(c, d) \to \int^{c\in\mathcal{C}} \phi(c, d) \tag{8}$$

$$p_d : \coprod_{d} \phi(c, d) \to \int^{d\in\mathcal{D}} \phi(c, d) \tag{9}$$

We can then use this to define the composition of two profunctors. Given three categories $\mathcal{C}, \mathcal{D}$ and $\mathcal{E}$ and two profunctors

$$\phi : C \nrightarrow D \quad \psi : D \nrightarrow E$$

We can define their composition as follows

$$(\phi \diamond \psi)(c, e) = \left(\int^{d\in\mathcal{D}} \phi(c, d) \times \psi(d, e)\right) : \mathcal{C} \nrightarrow \mathcal{E}$$

Within set-enrichment we can define this as the following coequalizer

$$(\phi \diamond \psi)(c, e) = \mathrm{coeq}\left(\coprod_{g : d \to d'} \phi(c, d') \times \psi(d, e) \rightrightarrows \coprod_{d} \phi(c, d) \times \psi(d, e) \right)$$

Equivalently we can express this as the quotient of the coproduct

$$(\phi \diamond \psi)(c, e) = \left(\coprod_{d} \phi(c, d) \times \psi(d, e) \right) / \sim$$

With the equivalence relation $\sim$ generated by the pairs

$$\{((x, \psi(g, e)(y)), (\phi(c, g)(x), y)) \mid g : d \to d', x \in \phi(c, d'), y \in \psi(d, e)\}$$

An alternative way of writing the equivalence relation is as follows

$$(\phi(c, g)(x), y) \sim (x, \psi(g, e)(y)) \quad \forall g : d \to d', x \in \phi(c, d'), y \in \psi(d, e)$$

The value $\phi(c, g)(x)$ here represents the action of the profunctor ϕ in the contravariant direction while $\psi(g, e)(y)$ represents the action of the profunctor ψ in the covariant direction.

The idea being that the composition of profunctors along some common codomain and domain — in this instance $\mathcal{D}$ — is given by the ways in which we can relate objects in $\mathcal{C}$ to objects in $\mathcal{E}$ through objects in $\mathcal{D}$. The equivalence relation is then saying that if we have a way of relating $c \in \mathcal{C}$ to $d' \in \mathcal{D}$ and a way of relating d' to $e \in \mathcal{E}$ then this is equivalent to having a way of relating c to $d \in \mathcal{D}$ and a way of relating d to e. This is exactly the same idea as the composition of relations.

We can compare this back to the simple case of composing binary relations. Given three sets A, B and C and two relations

$$R : A \to B \quad S : B \to C \equiv R \subseteq A \times B \quad S \subseteq B \times C$$

We can define their composition as follows

$$R \circ S = \{(a, c) \mid \exists b \in B, (a, b) \in R \land (b, c) \in S\} \subseteq A \times C$$

A nice way the coend action is described — and something quite apparent via the above definition — is that the coend 'traces out' the common category $\mathcal{D}$ in the same way that the existential quantifier 'traces out' the common set B. An important property of profunctor composition is that it yields a bicategory known as **Prof**. We can describe **Prof** as follows

- (Objects) - Small categories
- (1-morphisms) - Profunctors between small categories
- (2-morphisms) - Natural transformations between profunctors
- (Composition) - Profunctor composition

3.3.2 Bimodules

There is a nice way of understanding both profunctors and lax functors in terms of bimodules. Before we define bimoduels there are two concepts we need to quickly explain first. A module V is an object equipped with an action by a monoid M. A ring is just a monoid in the category of abelian groups. So a module over a ring is an abelian group equipped with an action by a ring. A bimodule is then an abelian group equipped with two actions by two rings, one on the left and one on the right, such that the two actions commute.

When described in terms of monoids we would say that given two monoids A and B an (A, B)-bimodule in a monoidal category $(\mathcal{C}, \otimes, I)$ is given by

- (Object) - An object $M \in \mathcal{C}$
- (Left action) - A morphism in $\mathcal{C}$ known as the left action

$$\lambda : A \otimes M \to M$$

- (Right action) - A morphism in $\mathcal{C}$ known as the right action

$$\rho : M \otimes B \to M$$

- (Left module) - The left action must satisfy the usual module laws

$$\begin{tikzcd}
(A \otimes A) \otimes N \arrow[r, "1_A \otimes \lambda"] \arrow[d, "(-)\otimes 1_N"'] & A \otimes N \arrow[d, "\lambda"] \\
A \otimes N \arrow[r, "\lambda"] & N
\end{tikzcd}
\qquad
\begin{tikzcd}
I \otimes N \arrow[rr, "e \otimes 1_N"] \arrow[dr] & & A \otimes N \arrow[dl, "\lambda"] \\
& N &
\end{tikzcd}$$

If we are talking about a monoid over an ablian group then we can express the left module laws in terms of elements as follows

- (Associativity) - For all $a, a' \in A$ and $m \in M$

$$\lambda(a, \lambda(a', m)) = \lambda(a \cdot a', m) \mapsto ma \cdot ma'$$

- (Unitality) - For all $m \in M$

$$\lambda(e, m) = m$$

The element m here expresses the idea of an identity for the action, i.e. doing nothing.

- (Right module) - The right action must satisfy the usual module laws

$$\begin{tikzcd}
M \otimes (B \otimes B) \arrow[r, "1_M \otimes 1_B"] \arrow[d, "(-)\otimes 1_B"'] & M \otimes B \arrow[d, "\rho"] \\
M \otimes B \arrow[r, "\rho"] & M
\end{tikzcd}
\qquad
\begin{tikzcd}
M \otimes I \arrow[rr, "1_M \otimes e"] \arrow[dr] & & M \otimes B \arrow[dl, "\rho"] \\
& M &
\end{tikzcd}$$

If we again consider the case of a monoid over an abelian group we can express the right module laws in terms of elements as follows

- (Associativity) - For all $b, b' \in B$ and $m \in M$

$$\rho(\rho(m, b), b') = \rho(m, b \cdot b') \mapsto mb \cdot mb'$$

- (Unitality) - For all $m \in M$

$$\rho(m, e) = m$$

- (Commutativity) - The left and right actions must commute, i.e.

$$\begin{tikzcd}
A \otimes M \otimes B \arrow[r, "A \otimes \rho"] \arrow[d, "\lambda \otimes B"'] & A \otimes M \arrow[d, "\lambda"] \\
M \otimes B \arrow[r, "\rho"'] & M
\end{tikzcd}$$

One way of describing the left and right actions is that they are are a sort of pre and postcomposition of the object M by the monoids A and B. The commutativity condition is then saying that it doesn't matter whether we precompose M by A first and then postcompose by B or vice versa.

Naturally this creates a direct analogue to profunctors. Given two categories $\mathcal{C}$ and $\mathcal{D}$ and the profunctor

$$P : C \nrightarrow D \equiv P : \mathcal{C}^{\mathrm{op}} \times \mathcal{D} \to \mathbf{Set}$$

We have the following data

- (Object) - A functor $P : \mathcal{C}^{\mathrm{op}} \times \mathcal{D} \to \mathbf{Set}$
- (Left action) - For each morphism $f : c' \to c$ in $\mathcal{C}$ a natural transformation known as the left action

$$\lambda_f : P(c, -) \to P(c', -)$$

- (Right action) - For each morphism $g : d \to d'$ in $\mathcal{D}$ a natural transformation known as the right action

$$\rho_g : P(-, d) \to P(-, d')$$

- (Left module) - The left action must satisfy the usual module laws
- (Right module) - The right action must satisfy the usual module laws
- (Commutativity) - The left and right actions must commute

$$\begin{tikzcd}
\mathcal{C}_{c',c} \otimes P_{c,d} \otimes \mathcal{D}_{d,d'} \arrow[r, "C(c',c)\otimes\rho"] \arrow[d, "\lambda\otimes\mathcal{D}(d,d')"'] & P(c,d) \otimes \mathcal{D}(d,d') \arrow[d, "\rho"] \\
\mathcal{C}(c',c) \otimes P(c,d') \arrow[r, "\lambda"'] & P(c',d')
\end{tikzcd}$$

As profunctors can be equivelantly understood as bimodules we can also in turn define a composition of bimodules. Thus given three monoids A, B and C in a monoidal category $(\mathcal{C}, \otimes, I)$ and two bimodules

$$N : A\text{-}B \quad M : B\text{-}C$$

We can define their composition over B as follows

$$\begin{tikzcd}
N \otimes B \otimes M \arrow[r, shift left, "\rho^N\otimes M"] \arrow[r, shift right, "M\otimes\lambda^M"'] & N \otimes M \arrow[r] \arrow[d, "!k"] & N \otimes_B M \arrow[dl, dashed] \\
 & Q &
\end{tikzcd}$$

Equivelantly we can express this as the coend

$$N \otimes_B M = \left(\int^{b \in B} N \otimes M\right) : A\text{-}C$$

Of note being is that the bimodule tensor product is an epimoprhism in each argument. Meaning that given two bimodules $N, N' : A\text{-}B$ and a bimodule $M : B\text{-}C$ and a bimodule morphism $f : N \to N'$ the induced map

$$f \otimes_B 1_M : N \otimes_B M \to N' \otimes_B M$$

is also a bimodule morphism. The same holds for the other argument.

The tensor product of bimodules becomes a new bimodule denoted $N \otimes_B M : \mathcal{A}\text{-}C$. With its left action expressed via the following diagram

$$\begin{tikzcd}
\mathcal{A} \otimes N \otimes B \otimes M \arrow[r, shift left, "\mathcal{A}\otimes(\rho^N\otimes M)"] \arrow[r, shift right, "\mathcal{A}\otimes(M\otimes\lambda^M)"'] \arrow[d, "\lambda^N\otimes B\otimes M"'] & \mathcal{A} \otimes (N \otimes M) \arrow[r, "\mathcal{A}\otimes\pi_{N,M}"] \arrow[d, "\lambda^N\otimes B"] & \mathcal{A} \otimes (N \otimes_B M) \arrow[d, dashed, "\lambda^{N\otimes_B M}"] \\
N \otimes B \otimes M \arrow[r, shift left, "\rho^N\otimes M"] \arrow[r, shift right, "M\otimes\lambda^M"'] & N \otimes M \arrow[r, "\pi_{N,M}"'] & N \otimes_B M
\end{tikzcd}$$

And the right action expressed via the following diagram

$$\begin{tikzcd}
N \otimes B \otimes M \otimes C \arrow[r, shift left, "(\rho^N \otimes M) \otimes C"] \arrow[r, shift right, "(M \otimes \lambda^M) \otimes C"'] \arrow[d, "N \otimes M \otimes \rho^M"'] & (N \otimes M) \otimes C \arrow[r, "\pi_{N,M} \otimes C"] \arrow[d, "N \otimes \rho^M"] & (N \otimes_B M) \otimes C \arrow[d, dashed, "\rho^{N \otimes_B M}"] \\
N \otimes B \otimes M \arrow[r, shift left, "\rho^N \otimes M"] \arrow[r, shift right, "M \otimes \lambda^M"'] & N \otimes M \arrow[r, "\pi_{N,M}"'] & N \otimes_B M
\end{tikzcd}$$

Then assuming all reflexive-coequalizers exist and are preserved by the tensor product we have that the tensor product of bimodules is associative and unital up to isomorphism. Formally

$$(M \otimes_B N) \otimes_C P \cong M \otimes_B (N \otimes_C P) \quad M \otimes_A A \cong M \cong B \otimes_B M \tag{10}$$

An observation to make here is that the left and right actions of the composed bimodule correspond precisely to the pentagon identity. The left and right action here express that the coequalizer which defines the tensor product of bimodules is compatible with the left and right actions of the respective monoids.

Another attribute which we can describe is the idea of a morphism between two bimodules. Given two monoids A and B in a monoidal category $(\mathcal{C}, \otimes, I)$ and two (A, B)-bimodules M and N a morphism of (A, B)-bimodules is a morphism in $\mathcal{C}$

$$f : M \to N$$

Such that f is compatible with the left and right actions. Meaning that the following diagrams commute

$$\begin{tikzcd}
A \otimes M \arrow[r, "1_A \otimes f"] \arrow[d, "\rho_M"'] & A \otimes N \arrow[d, "\lambda_N"] \\
M \arrow[r, "f"'] & N
\end{tikzcd} \qquad \begin{tikzcd}
N \otimes B \arrow[r, "f \otimes 1_A"] \arrow[d, "\lambda_M"'] & N \otimes B \arrow[d, "\rho_N"] \\
M \arrow[r, "f"'] & N
\end{tikzcd}$$

With these constructs we can define the bicategory of bimodules **BiMod**$(\mathcal{C})$ internal to a monoidal category $(\mathcal{C}, \otimes, I)$. This bicategory has

- (0-cells) - Alebras (monoids) in $\mathcal{C}$
- (1-cells) - (A, B)-modules M

$$\lambda : A \otimes M \to M \quad \rho : M \otimes B \to M$$

- (2-cells) - Morphisms of (A, B)-bimodules $f : M \to N$
- (Composition) - Given (A, B)-bimodule M and (B, C)-bimodule N their composition is given by the relative tensor product $M \otimes_B N$ which again is just the coend
- (Unitors) - For each pair of algebras A and B we have unitors exhibiting the isomorphisms

$$A \otimes_A M \cong M \cong M \otimes_B B$$

- (Associators) - For each triple of algebras A, B and C we have associators exhibiting the isomorphism

$$(M \otimes_B N) \otimes_C P \cong M \otimes_B (N \otimes_C P)$$

  As established above

The equivalent understanding here is that the bicategory of bimodules is just the bicategory of profunctors internal to a monoidal category. So its the category made up of a bunch of monoids and the ways in which they can be related via bimodules i.e. profunctors.

Free category Γ on a graph $\mathcal{G}$

- (Objects) - Vertices of $\mathcal{G}$
- (1-cells) - Paths in $\mathcal{G}$
- (2-cells) - Path equivalences / Identities

Bifunctor $F : \Gamma \to \mathbf{BiMod}(\mathcal{C})$ represents a graph where

- (Vertices) - Algebras in $\mathcal{C}$
- (Edges) - For an edge $r : A \to B$ in $\mathcal{G}$ a (A, B)-bimodule $F(r)$
- (Paths) - For a path $r : A \to B$ and $s : B \to C$ in $\mathcal{G}$ the composition of bimodules

$$\circ'_{r,s} : F(r) \otimes_B F(s) \to F(r \circ s)$$

- Unitors and identities satisfy the usual coherence laws

Canonical forgetful functor

$$U : \mathbf{BiMod}(\mathcal{C}) \to \Sigma(\mathcal{C})$$

Which

- (On objects) - Sends an algebra to itself
- (On 1-cells) - Sends a (A, B)-bimodule M to its underlying object $M \in \mathcal{C}$
- (On Composites) - Sends the relative tensor product $M \otimes_B N$ to $|M \otimes_B N| \in \mathcal{C}$ via the comparison map

$$|M| \otimes |N| \xrightarrow{q} |M \otimes_B N|$$

Where $\Sigma(\mathcal{C})$ is the category of algebras in $\mathcal{C}$ and algebra homomorphisms. In other words its the bicategory with

- (0-cells) - One object
- (1-cells) - Algebras in $\mathcal{C}$
- (2-cells) - Algebra homomorphisms
- (Composition) - Given two algebras A and B their composition is given by the tensor product $A \otimes B$
- (Identity) - The unit object $I \in \mathcal{C}$

The composite of U with F gives the lax functor

$$U \circ F : \Gamma \to \Sigma(\mathcal{C})$$

Which

- (On objects) - Given $a \in \mathcal{G}$

$$a \mapsto F(a) = *$$

- (On 1-cells) - Given $r : a \to b$ in $\mathcal{G}$

$$r \mapsto F(r) = |F(r)| \in \mathcal{C}$$

- (Unitors) - Given $a \in \mathcal{G}$ the unitor

$$\eta_a : I \to |F(1_a)|$$

- (Composites) - for r, s

$$\circ_{r,s} : F(s) \otimes F(r) = |F(r)| \otimes |F(s)| \xrightarrow{q} |F(r) \otimes_B F(s)| \xrightarrow{|\circ'_{r,s}|} F(r \circ s)$$

A cool property we can observe here is that if our free category Γ corresponds to the termainl bicategory with two objects then a lax functor

$$F : \Gamma \to \Sigma(\mathcal{C})$$

Definitionally describes a monad in $\Sigma(\mathcal{C})$.

$$\begin{tikzcd}
a \arrow[r, "1_a"] & a & \to & F(a) \arrow[rr, bend left, "1"] \arrow[rr, bend right, "F(1_a)"'] \arrow[rr, phantom, "\Downarrow i_a"] & & F(a)
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & & & F(b) \arrow[d, Rightarrow, "\circ_{r,s}"] & \\
a \arrow[r, "r"'] \arrow[rr, bend left, dashed, "s\circ r"] & b \arrow[r, "s"'] & c & \to & F(a) \arrow[ur, "F(r)"] \arrow[rr, "F(s\circ r)"'] & {} & F(a) \arrow[ul, "F(s)"']
\end{tikzcd}$$

$$\begin{tikzcd}
F(1_b) \arrow[dr] & F(s) \arrow[d] & & F(r) \arrow[d] & F(1_b) \arrow[dl] \\
F(r) \arrow[dr] & \circ_{1_b,s} \arrow[d] & & \circ_{r,1_b} \arrow[d] & F(s) \arrow[dl] \\
 & \circ_{r,s} \arrow[d] & \equiv & \circ_{r,s} \arrow[d] & \\
 & F(s \circ r) & & F(s \circ r) &
\end{tikzcd}$$

4 The Dependent State Monad

4.1 Slice Categories

$$\begin{tikzcd}
X \arrow[rr, "g"] \arrow[dr, "f"] \arrow[dr, bend right, dashed, "g\circ f'"'] & & X' \arrow[dl, "f'"'] \\
 & A &
\end{tikzcd}$$