Chapter 10

Adjunctions

A sculptor subtracts irrelevant stone until a sculpture emerges. A mathematician abstracts irrelevant details until a pattern emerges.

We were able to define a lot of constructions using their mapping-in and mapping-out properties. Those, in turn, could be compactly written as isomorphisms between hom-sets. This pattern of natural isomorphisms between hom-sets is called an adjunction and, once recognized, pops up virtually everywhere.

10.1 The Currying Adjunction

The definition of the exponential is the classic example of an adjunction that relates mappings-out and mappings-in. Every mapping out of a product corresponds to a unique mapping into the exponential:

$$\mathcal{C}(A \times B, C) \cong \mathcal{C}(A, C^B)$$

The object C takes the role of the focus on the left hand side; the object A becomes the observer on the right hand side.

On the left, A is mapped to a product $A \times B$, and on the right, C is exponentiated.

We can spot two functors at play. They are both parameterized by B. On the left we have the functor $(- \times B)$ acting on A. On the right we have the functor $(-)^B$ acting on C.

If we write these functors as:

$$L_B A = A \times B$$

$$R_B C = C^B$$

then the natural isomorphism

$$\mathcal{C}(L_B A, C) \cong \mathcal{C}(A, R_B C)$$

is called the adjunction between them.

The shorthand notation for the adjunction is $L \dashv R$. Substituting the product functor for L and the exponential functor for R, we can write the currying adjunction concisely as:

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

10.2 The Sum and the Product Adjunctions

The currying adjunction relates two endofunctors, but an adjunction can be easily generalized to functors that go between categories. Let's see some examples first.

The diagonal functor

The sum and the product types were defined using bijections where one of the sides was a single arrow and the other was a pair of arrows. A pair of arrows can be seen as a single arrow in the product category.

To explore this idea, we need to define the diagonal functor Δ, which is a particular mapping from $\mathcal{C}$ to $\mathcal{C} \times \mathcal{C}$. It takes an object X and duplicates it, producing a pair of objects $\langle X, X \rangle$. It also takes an arrow f and duplicates it $\langle f, f \rangle$

Interestingly, the diagonal functor is related to the constant functor we've seen previously. The constant functor can be though of as a functor of two variables—it just ignores the second one. We've seen this in the Haskell definition:

```
data Const c a = Const c
```

To see the connection, let's look at the product category $\mathcal{C} \times \mathcal{C}$ as a functor category $[\mathbf{2}, \mathcal{C}]$, in other words, the exponential object $\mathcal{C}^{\mathbf{2}}$ in **Cat**. Indeed, a functor from $\mathbf{2}$ picks a pair of objects—which is a single object in the product category.

A functor $\mathcal{C} \to [\mathbf{2}, \mathcal{C}]$ can be uncurried to $\mathcal{C} \times \mathbf{2} \to \mathcal{C}$. If we do this to the diagonal functor, we see that it ignores the second argument, the one coming from $\mathbf{2}$: it does the same whether the second argument is 1 or 2. But that's exactly what the constant functor does. This is why we use the same symbol Δ for both.

Incidentally, this argument can be easily generalized to any indexing category, not just $\mathbf{2}$.

The sum adjunction

Recall that the sum is defined by its mapping out property. There is one-to one correspondence between the arrows coming out of the sum $A + B$ and pairs of arrows coming from A and B separately. In terms of hom-sets, we can write it as:

$$\mathcal{C}(A + B, X) \cong \mathcal{C}(A, X) \times \mathcal{C}(B, X)$$

where the product on the right-hand side is just a cartesian product of sets, that is the set of pairs. Moreover, we've seen earlier that this bijection is natural in X.

We know that a pair of arrows is a single arrow in the product category. We can, therefore, look at the elements on the right-hand side as arrows in $\mathcal{C} \times \mathcal{C}$ going from the object $\langle A, B \rangle$ to the object $\langle X, X \rangle$. The latter object we can be obtained by acting with the diagonal functor Δ on X. We have:

$$\mathcal{C}(A + B, X) \cong (\mathcal{C} \times \mathcal{C})(\langle A, B \rangle, \Delta X)$$

This is a bijection between hom-sets in two different categories. It satisfies naturality conditions, so it's a natural isomorphism.

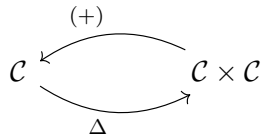
We can spot a pair of functors here as well. On the left we have the functor that takes a pair of objects $\langle A, B\rangle$ and produces their sum $A + B$

$$(+) \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

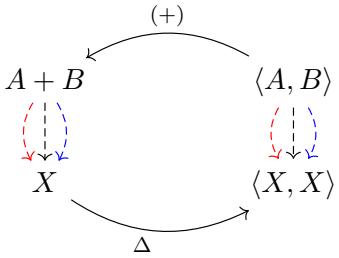
On the right, we have the diagonal functor Δ going in the opposite direction

$$\Delta \colon \mathcal{C} \to \mathcal{C} \times \mathcal{C}$$

Altogether, we have a pair of functors between a pair of categories:



and an isomorphism between the hom-sets:



The product adjunction

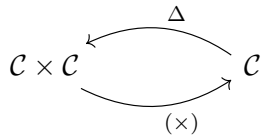
We can apply the same reasoning to the definition of a product. This time we have a natural isomorphism between pairs of arrows and and a mapping into the product.

$$\mathcal{C}(X, A) \times \mathcal{C}(X, B) \cong \mathcal{C}(X, A \times B)$$

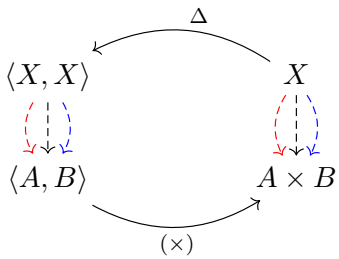
Replacing pairs of arrows with arrows in the product category we get:

$$(\mathcal{C} \times \mathcal{C})(\Delta X, \langle A, B\rangle) \cong \mathcal{C}(X, A \times B)$$

These are the two functors in action:



and this is the isomorphism of hom-sets:



In other words, we have the adjunction:

$$(+) \dashv \Delta$$

10.3 Adjunction between functors

In general, an adjunction relates two functors going in opposite directions between two categories. The left functor

$$L \colon \mathcal{D} \to \mathcal{C}$$

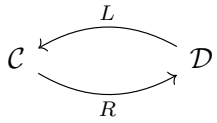
and the right functor:

$$R \colon \mathcal{C} \to \mathcal{D}$$

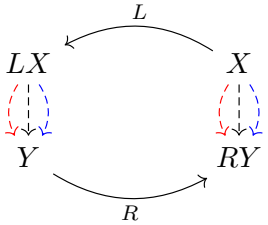
The adjunction $L \dashv R$ is defined as a natural isomorphism between two hom-sets.

$$\mathcal{C}(LX, Y) \cong \mathcal{D}(X, RY)$$

Pictorially, these are the two functors:



and this is the isomorphism of hom-sets:



These hom-sets come from two different categories, but sets are just sets. We say that L is the left adjoint of R, or that R is the right adjoint of L

In Haskell, the simplified version of this could be encoded as a multi-parameter type class:

```
class (Functor left, Functor right) => Adjunction left right where
  ltor :: (left x -> y) -> (x -> right y)
  rtol :: (x -> right y) -> (left x -> y)
```

It requires the following pragma at the top of the file:

```
{-# language MultiParamTypeClasses #-}
```

Therefore, in a bicartesian category, the sum is the left adjoint to the diagonal functor, and the product is its right adjoint. We can write this very concisely (or we could impress it in clay, in a modern version of cuneiform):

$$(+) \dashv \Delta \dashv (\times)$$

Exercise 10.3.1. *The hom-set $\mathcal{C}(LX, Y)$ on the left-hand side of the adjunction formula suggests that LX could be seen as a representing object for some functor (a co-presheaf). What is this functor? Hint: It maps a Y to a set. What set is it?*

Exercise 10.3.2. *Conversely, a representing object A for a presheaf P is defined by:*

$$PX \cong \mathcal{D}(X, A)$$

What is the presheaf for which RY, in the adjunction formula, is the representing object.

10.4 Limits and Colimits

The definition of a limit also involves a natural isomorphism between hom-sets:

$$[\mathbf{I}, \mathcal{C}](\Delta_X, D) \cong \mathcal{C}(X, \mathrm{Lim}_D)$$

The hom-set on the left is in the functor category. Its elements are cones, or natural transformations between the constant functor and the diagram functor. The one on the right is a hom-set in $\mathcal{C}$.

In a category where all limits exist, we have the adjunction between these two functors:

$$\Delta_{(-)} \colon \mathcal{C} \to [\mathbf{I}, \mathcal{C}]$$

$$Lim_{(-)} \colon [\mathbf{I}, \mathcal{C}] \to \mathcal{C}$$

Dually, the colimit is described by the following natural isomorphism:

$$[\mathbf{I}, \mathcal{C}](D, \Delta_X) \cong \mathcal{C}(\mathrm{Colim}_D, X)$$

We can write both adjunctions using one terse formula:

$$Colim \dashv \Delta \dashv Lim$$

10.5 Unit and Counit of Adjunction

We compare arrows for equality, but we prefer to use isomorphisms for comparing objects.

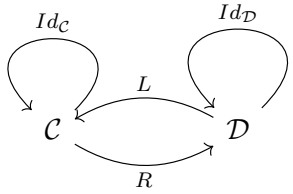
We have a problem with functors, though. On the one hand, they are objects in the functor category, so isomorphisms are the way to go; on the other hand, they are arrows in **Cat** so maybe it's okay to compare them for equality?

To shed some light on this dilemma, we should ask ourselves *why* we use equality for arrows. It's not because we like equality, but because there's nothing else for us to do in a set. Two elements of a set are either equal or not, period.

That's not the case in **Cat** which, as we know, is a 2-category. Here, hom-sets themselves have the structure of a category—the functor category. In a 2-category we have arrows between arrows so, in particular, we can define isomorphisms between arrows. In **Cat** these would be natural isomorphisms between functors.

However, even though we have the option of replacing arrow equalities with isomorphisms, categorical laws in **Cat** are still expressed as equalities. So, for instance, the composition of a functor F with the identity functor is *equal* to F, and the same for associativity. A 2-category in which the laws are satisfied "on the nose" is called *strict*, and **Cat** is an example of a strict 2-category.

As far as comparing categories goes, we have even more options. Categories are objects in **Cat**, so it's possible to define an isomorphism of categories as a pair of functors L and R:



such that

$$L \circ R = Id_{\mathcal{C}}$$

$$Id_{\mathcal{D}} = R \circ L$$

This definition involves equality of functors, though. What's worse, acting on objects, it involves equality of objects:

$$L(RX) = X$$

$$Y = R(LY)$$

This is why it's more proper to talk about a weaker notion of *equivalence* of categories, where equalities are replaced by isomorphisms:

$$L \circ R \cong Id_{\mathcal{C}}$$

$$Id_{\mathcal{D}} \cong R \circ L$$

On objects, an equivalence of categories means that a round trip produces an object that is isomorphic, rather than equal, to the original one. In most cases, this is exactly what we want.

An adjunction is also defined as a pair of functors going in opposite directions, so it makes sense to ask what the result of a round trip is. The isomorphism that defines an adjunction works for any pair of objects X and Y

$$\mathcal{C}(LX, Y) \cong \mathcal{D}(X, RY)$$

so, in particular, we can replace Y with LX

$$\mathcal{C}(LX, LX) \cong \mathcal{D}(X, R(LX))$$

We can now use the Yoneda trick and pick the identity arrow id_{LX} on the left. The isomorphism maps it to a unique arrow on the right, which we'll call η_X:

$$\eta_X : X \to R(LX)$$

Not only is this mapping defined for every X, but it's also natural in X. The natural transformation η is called the *unit* of the adjunction. If we observe that the X on the left is the action of the identity functor on X, we can write:

$$\eta : Id_{\mathcal{D}} \to R \circ L$$

We can do a similar trick by replacing X with RY:

$$\mathcal{C}(L(RY), Y) \cong \mathcal{D}(RY, RY)$$

We get another natural transformation called the *counit* of the adjunction:

$$\varepsilon : L \circ R \to Id_{\mathcal{C}}$$

Notice that, if those two natural transformations were invertible, they would witness the equivalence of categories. But this kind of "half-equivalence" is even more interesting in the context of category theory.

Triangle identities

We can use the unit/counit pair to formulate an equivalent definition of adjunction. To do that, we will start with two natural transformations:

$$\eta \colon Id_{\mathcal{D}} \to R \circ L$$

$$\varepsilon \colon L \circ R \to Id_{\mathcal{C}}$$

and impose additional *triangle identities*.

These identities are derived by noticing that η can be used to replace an identity functor by the composite $R \circ L$, effectively letting us insert $R \circ L$ anywhere an identity functor would work.

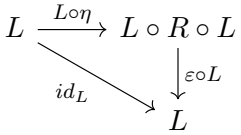
Similarly, ε can be used to eliminate the other composite $L \circ R$.

So, for instance, starting with L:

$$L = L \circ Id_{\mathcal{D}} \xrightarrow{L \circ \eta} L \circ R \circ L \xrightarrow{\varepsilon \circ L} Id_{\mathcal{C}} \circ L = L$$

Here, we used the horizontal composition of natural transformation, with one of them being the identity transformation (a.k.a., whiskering).

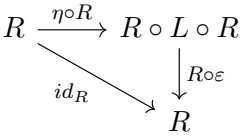
The first triangle identity is the condition that this chain of transformations result in the identity natural transformation. Pictorially:



Similarly, the following chain of natural transformations should also compose to identity:

$$R = Id_{\mathcal{D}} \circ R \xrightarrow{\eta \circ R} R \circ L \circ R \xrightarrow{R \circ \varepsilon} R \circ Id_{\mathcal{C}} = R$$

or, pictorially:



It turns out that an adjunction can be alternatively defined in terms of the two natural transformations, η and ε, as long as the triangle identities are satisfied.

The mapping of hom-sets can be easily recovered. For instance, let's start with an arrow $f \colon X \to RY$, which is an element of $\mathcal{D}(X, RY)$. We can lift it to

$$Lf \colon LX \to L(RY)$$

We can then use η to collapse the composite $L \circ R$ to identity. The result is a mapping $LX \to Y$, which is an element of $\mathcal{C}(LX, Y)$.

The definition of the adjunction using unit and counit is more general in the sense that it can be translated to a 2-category setting.

Exercise 10.5.1. *Given an arrow $g \colon LX \to Y$ implement an arrow $X \to RY$ using ε and the fact that R is a functor. Hint: Start with the object X and see how you can get from there to RY with one stopover.*

The unit and counit of the currying adjunction

Let's calculate the unit and the counit of the currying adjunction:

$$\mathcal{C}(A \times B, C) \cong \mathcal{C}(A, C^B)$$

If we replace C with $A \times B$, we get

$$\mathcal{C}(A \times B, A \times B) \cong \mathcal{C}(A, (A \times B)^B)$$

Corresponding to the identity arrow on the left, we get the unit of the adjunction:

$$\eta \colon A \to (A \times B)^B$$

This is a curried version of product constructor. In Haskell, we can write it as:

```
mkpair :: a -> (b -> (a, b))
mkpair = curry id
```

The counit is more interesting. Replacing A with C^B we get:

$$\mathcal{C}(C^B \times B, C) \cong \mathcal{C}(C^B, C^B)$$

Corresponding to the identity arrow on the right, we get:

$$\varepsilon \colon C^B \times B \to C$$

which is the function application arrow.

In Haskell:

```
apply :: (a -> b, a) -> b
apply = uncurry id
```

Exercise 10.5.2. *Derive the unit and counit for the sum and product adjunctions.*

10.6 Distributivity

In a bicartesian closed category products distribute over sums. We've seen one direction of the proof using universal constructions. Adjunctions combined with the Yoneda lemma give us more powerful tools to tackle this problem.

We want to show the natural isomorphism:

$$(B + C) \times A \cong B \times A + C \times A$$

Instead of proving this identity directly, we'll show that the mappings out from both sides to an arbitrary object X are isomorphic:

$$\mathcal{C}((B + C) \times A, X) \cong \mathcal{C}(B \times A + C \times A, X)$$

The left hand side is a mapping out of a product, so we can apply the currying adjunction to it:

$$\mathcal{C}((B + C) \times A, X) \cong \mathcal{C}(B + C, X^A)$$

This gives us a mapping out of a sum which, by the sum adjunction is isomorphic to the product of two mappings:

$$\mathcal{C}(B + C, X^A) \cong \mathcal{C}(B, X^A) \times \mathcal{C}(C, X^A)$$

We can now apply the inverse of the currying adjunction to both components:

$$\mathcal{C}(B, X^A) \times \mathcal{C}(C, X^A) \cong \mathcal{C}(B \times A, X) \times \mathcal{C}(C \times A, X)$$

Using the inverse of the sum adjunction, we arrive at the final result:

$$\mathcal{C}(B \times A, X) \times \mathcal{C}(C \times A, X) \cong \mathcal{C}(B \times A + C \times A, X)$$

Every step in this proof was a natural isomorphism, so their composition is also a natural isomorphism. By Yoneda lemma, the two objects that form the left- and the right-hand side of distributivity law are therefore isomorphic.

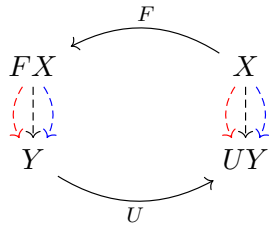
10.7 Free-Forgetful Adjunctions

The two functors in the adjunction play different roles: the picture of the adjunction is not symmetric. Nowhere is this illustrated better than in the case of the free/forgetful adjunctions.

A forgetful functor is a functor that "forgets" some of the structure of its source category. This is not a rigorous definition but, in most cases, it's pretty obvious what structure is being forgotten. Very often the target category is just the category of sets, which is considered the epitome of structurelessness. The result of the forgetful functor, in that case, is called the "underlying" set, and the functor itself is often called U.

More precisely, we say that a functor forgets *structure* if the mapping of hom-sets is not surjective, that is, there are arrows in the target hom-set that have no corresponding arrows in the source hom-set. Intuitively, it means that the arrows in the source have to preserve some structure, and that structure is absent in the target.

The left adjoint to a forgetful functor is called a *free functor*.

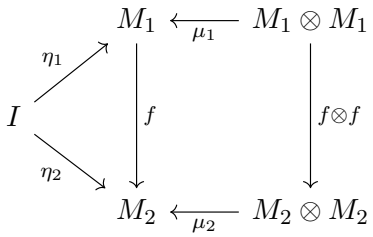


A classic example of a free/forgetful adjunction is the construction of the free monoid.

The category of monoids

Monoids in a monoidal category $\mathcal{C}$ form their own category $\mathbf{Mon}(\mathcal{C})$. Its objects are monoids, and its arrows are the arrows in $\mathcal{C}$ that preserve the monoidal structure.

The following diagram explains what it means for f to be a monoid morphism from a monoid M_1 to a monoid M_2:



A monoid morphism f must map unit to unit, which means that:

$$f \circ \eta_1 = \eta_2$$

and it must map multiplication to multiplication:

$$f \circ \mu_1 = \mu_2 \circ (f \otimes f)$$

Remember, the tensor product $\otimes$ is functorial, so it can lift pairs of arrows, here $f \otimes f$.

In particular, the category **Set** is monoidal, with cartesian product and the terminal object providing the monoidal structure.

Monoids in **Set** are sets with additional structure. They form their own category **Mon**(**Set**) and there is a forgetful functor U that simply maps the monoid to the set of its elements. When we say that a monoid is a set, we mean the underlying set.

Free monoid

We want to construct the free functor

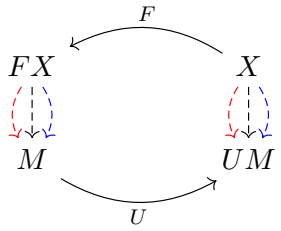
$$F \colon \mathbf{Set} \to \mathbf{Mon}(\mathbf{Set})$$

that is adjoint to the forgetful functor U. We start with an arbitrary set X and an arbitrary monoid M.

On the right-hand side of the adjunction we have the set of functions between two sets, X and UM. On the left-hand side, we have a set of highly constrained structure-preserving monoid morphisms from FX to M. How can these two sets be isomorphic?

In **Mon**(**Set**), monoids are just sets of elements, and a monoid morphism is a function between such sets, satisfying additional constraints: it has to preserve unit and multiplication.

Arrows in **Set**, on the other hand, are just functions with no additional constraints. So, in general, there are fewer arrows between monoids than there are between their underlying sets.



Here's the idea: if we want to have a one to one matching between arrows, we want FX to be much larger than X. This way, there will be many more functions from it to M—so many that, even after rejecting the ones that don't preserve the structure, we'll still have enough to match every function $f\colon X \to UM$.

We'll construct the monoid FX starting from the set X, adding more elements as necessary. We'll call X the set of *generators* of FX.

We'll construct a monoid morphism $g\colon FX \to M$ starting from the function f. On generators, g works the same as f:

$$gx = fx$$

Every time we add new element to FX, we'll extend the definition of g.

Since FX is supposed to be a monoid, it has to have a unit. We can't pick one of the generators for the unit, because it would impose a constraint of f—it would have to map it to the unit of M. So we'll just add an extra element e to X and call it a unit. We'll define the action of g on it by saying that it is mapped to the unit e' of M:

$$ge = e'$$

We also have to define monoidal multiplication in FX. Let's start with a product of two generators a and b. The result of the multiplication cannot be another generator because, again, that would constrain f—products must be mapped to products. So we have to make all products of generators new elements of FX. Again, the action of g on those products is fixed:

$$g(a \cdot b) = ga \cdot gb$$

Continuing with this construction, any new multiplication produces a new element of FX, except when it can be reduced to an existing element by applying monoid laws. For instance, the new unit e times a generator a must be equal to a. But we have made sure that e is mapped to the unit of M, so the product $ge \cdot ga$ is automatically equal to ga.

Another way of looking at this construction is to think of the set X as an alphabet. The elements of FX are then strings of characters from this alphabet. The generators are single-letter strings, "a", "b", and so on. The unit is an empty string , "". Multiplication is string concatenation, so "a" times "b" is a new string "ab". Concatenation is automatically associative and unital, with the empty string as the unit.

The intuition behind free functors is that they generate structure "freely," as in "with no additional constraints." They do it lazily: instead of performing operations, they just record them.

The free monoid "remembers to do the multiplication" at a later time. It stores the arguments to multiplication in a string, but doesn't perform the multiplication. It's only allowed to simplify its records based on generic monoidal laws. For instance, it doesn't have to store the command to multiply by the unit. It can also "skip the parentheses" because of associativity.

Exercise 10.7.1. *What is the unit and the counit of the free monoid adjunction $F \dashv U$?*

Free monoid in programming

In Haskell, monoids are defined using the following typeclass:

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

Here, `mappend` is the curried form of the mapping from the product `(m, m)` to `m`. The `mempty` element corresponds to the arrow from the terminal object (unit of the monoidal category) to `m`.

A free monoid generated by some type `a`, treated as a set of generators, is represented by a list type `[a]`. An empty list serves as the unit; and monoid multiplication is implemented as list concatenation, traditionally written in infix form:

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

A list is an instance of a `Monoid`:

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

To show that it's a free monoid, we have to be able to construct a monoid morphism from the list of `a` to an arbitrary monoid `m`, provided we have an (unconstrained) mapping from `a` to (the underlying set of) `m`. We can't express all of this in Haskell, but we can define the function:

```
foldMap :: Monoid m => (a -> m) -> ([a] -> m)
foldMap f = foldr mappend mempty . fmap f
```

This function transforms the elements of the list to monoidal values using `f` and then folds them using `mappend`, starting with the unit `mempty`.

It's easy to see that an empty list is mapped to the monoidal unit. It's not too hard to see that a concatenation of two lists is mapped to the monoidal product of the results. So, indeed, `foldMap` produces a monoid morphism.

Following the intuition of a free monoid being a domain-specific program, `foldMap` provides an *interpreter* for this program. It performs all the multiplications that have been postponed. Note that the same program may be interpreted in many ways, depending on the choice of the concrete monoid and the function `f`.

Exercise 10.7.2. *Write a program that takes a list of integers and interprets it in two ways: once using the additive and once using the multiplicative monoid of integers.*