The Dao of Functional Programming

Bartosz Milewski

(Last updated: October 28, 2021)

Contents

Preface

Most programming texts, following Brian Kernighan, start with "Hello World!". It's natural to want to get the immediate gratification of making the computer do your bidding and print these famous words. But the real mastery of computer programming goes deeper than that, and rushing into it may only give you a false feeling of power, when in reality you're just parroting the masters. If your ambition is just to learn a useful, well-paid skill then, by all means, write your "Hello World!" program. There are tons of books and courses that will teach you to write code in any language of your choice. However, if you really want to get to the essence of programming, you need to be patient and persistent.

Chapter 1

Clean Slate

Programming starts with types and functions. You probably have some preconceptions about what types and functions are: get rid of them! They will cloud your mind.

Don't think about how things are implemented in hardware. What computers are is just one of the many models of computation. We shouldn't get attached to it. You can perform computations in your mind, or with pen and paper. The physical substrate is irrelevant to the idea of programming.

1.1 Types and Functions

Paraphrasing Lao Tzu: *The type that can be described is not the eternal type.* In other words, type is a primitive notion. It cannot be defined.

Instead of calling it a *type*, we could as well call it an *object* or a *proposition*. These are the words that are used to describe it in different areas of mathematics (type theory, category theory, and logic, respectively).

There may be more than one type, so we need a way to name them. We could do it by pointing fingers at them, but since we want to effectively communicate with other people, we usually name them. So we'll talk about type A, B, C; or `Int`, `Bool`, `Double`, and so on. These are just names.

A type by itself has no meaning. What makes it special is how it connects to other types. The connections are described by arrows. An arrow has one type as its source and one type as its target. The target could be the same as the source, in which case the arrow loops around.

An arrow between types is called a *function*. An arrow between objects is called a *morphism*. An arrow between propositions is called an *entailment*. These are just words that are used to describe arrows in different areas of mathematics. You can use them interchangeably.

A proposition is something that may be true. In logic, we interpret an arrow between two objects as A entails B, or B is derivable from A.

There may be more than one arrow between two types, so we need to name them. For instance, here's an arrow called f that goes from type A to type B

$$A \xrightarrow{f} B$$

One way to interpret this is to say that the function f takes an argument of type A and produces a result of type B. Or that f is a proof that if A is true then B is also true.

Note: The connection between type theory, lambda calculus (which is the foundation of programming), logic, and category theory is known as Curry-Howard-Lambek correspondence.

1.2 Yin and Yang

An object is defined by its connections. An arrow is a proof, a witness, of the fact that two objects are connected. Sometimes there's no proof, the objects are disconnected; sometimes there are many proofs; and sometimes there's a single proof—a unique arrow between two objects.

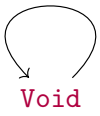
What does it mean to be *unique*? It means that if you can find two of those, then they must be equal.

An object that has a unique outgoing arrow to every object is called the *initial object*.

Its dual is an object that has a unique incoming arrow from every object. It's called the *terminal object*.

In mathematics, the initial object is often denoted by 0 and the terminal object by 1.

The initial object is the source of everything. As a type it's known as `Void`. It symbolizes the chaos from which everything arises. Since there is an arrow from `Void` to everything, there is also an arrow from `Void` to itself.



Thus `Void` begets `Void` and everything else.

The terminal object unites everything. As a type it's know as Unit. It symbolizes the ultimate order.

In logic, the terminal object signifies the ultimate truth, symbolized by T or $\top$. The fact that there's an arrow to it from any object means that $\top$ is true no matter what your assumptions are.

Dually, the initial object signifies logical falsehood, contradiction, or a counterfactual. It's written as False and symbolized by an upside down T, $\bot$. The fact that there is an arrow from it to any object means that you can prove anything starting from false premises.

In English, there is special grammatical construct for counterfactual implications. When we say, "If wishes were horses, beggars would ride," we mean that the equality between wishes and horses implies that beggars be able to ride. But we know that the premise is false.

A programming language lets us communicate with each other and with computers. Some languages are easier for the computer to understand, others are closer to the theory. We will use Haskell as a compromise.

In Haskell, the initial object corresponds to the type called `Void`. The name for the terminal type is `()`, a pair of empty parentheses, pronounced Unit. This notation will make sense later.

There are infinitely many types in Haskell, and there is a unique function/arrow from `Void` to each one of them. All these functions are known under the same name: `absurd`.

Programming	Category theory	Logic
type	object	proposition
function	morphism (arrow)	implication
`Void`	initial object, 0	False $\bot$
`()`	terminal object, 1	True $\top$

1.3 Elements

An object has no parts but it may have structure. The structure is defined by the arrows pointing at the object. We can *probe* the object with arrows.

In programming and in logic we want our initial object to have no structure. So we'll assume that it has no incoming arrows (other than the one that's looping back from it). Therefore `Void` has no structure.

The terminal object has the simplest structure. There is only one incoming arrow from any object to it: there is only one way of probing it from any direction. In this respect, the terminal object behaves like an indivisible point. Its only property is that it exists, and the arrow from any other object proves it.

Because the terminal object is so simple, we can use it to probe other, more complex objects.

If there is more than one arrow coming from the terminal object to some object A, it means that A has some structure: there is more than one way of looking at it. Since the terminal object behaves like a point, we can visualize each arrow from it as picking a different point or element of its target.

In category theory we say that x is a *global element* of A if it's an arrow

$$1 \xrightarrow{x} A$$

We'll often simply call it an element (omitting "global").

In type theory, $x\colon A$ means that x is of type A.

In Haskell, we use the double-colon notation instead:

```
x :: A
```

We say that `x` is a term of type `A`, but we'll interpret it as an arrow $x : 1 \to A$, a global element of `A`. [1]

In logic, such x is called the proof of A, since it corresponds to the implication $\top \to A$ (if **True** is true then **A** is true). Notice that there may be many different proofs of A.

[1] The Haskell type system distinguishes between `x :: A` and `x :: ()->A`. However, they denote the same thing in categorical semantics.

Since we have assumed that there be no arrows from any other object to `Void`, there is no arrow from the terminal object to it. Therefore `Void` has no elements. This is why we think of `Void` as empty.

The terminal object has just one element, since there is a unique arrow coming from it to itself, $1 \to 1$. This is why we sometimes call it a singleton.

Note: In category theory there is no prohibition against the initial object having incoming arrows from other objects. However, in cartesian closed categories that we're studying here, this is not allowed.

1.4 The Object of Arrows

We describe an object by looking at its arrows. Can we describe an arrow in a similar way? Could an arrow from A to B be represented as an element of some special *object of arrows*? After all, in programming we talk about the *type* of functions from `A` to `B`. In Haskell we write:

```
f :: A -> B
```

meaning that `f` is of the type "function from A to B". Here, `A->B` is just the name we are giving to this type. To fully define this type, we would have to describe its relation to other objects, in particular to A and B. We don't have the tools to do that yet, but we'll get there.

For now, let's keep in mind the following distinction: On the one hand we have an arrow which connects two objects `A` and `B`. On the other hand we have an element of the *object of arrows* from `A` to `B`. This element is itself defined as an arrow from the terminal object `()` to the object we call `A->B`.

The notation we use in programming tends to blur this distinction. This is why in category theory we call the object of arrows an *exponential* and write it as B^A (the source is in the exponent). So the statement:

```
f :: A -> B
```

is equivalent to

$$1 \xrightarrow{f} B^A$$

In logic, an arrow $A \to B$ is an implication: it states the fact that "if A then B." An exponential object B^A is the corresponding proposition. It could be true or it could be false, we don't know. You have to prove it. Such a proof is an element of B^A.

Show me an element of B^A and I'll know that B follows from A.

Consider again the statement, "If wishes were horses, beggars would ride"—this time as an object. It's not an empty object, because you can point at a proof of it—something along the lines: "A person who has a horse rides it. Beggars have wishes. Since wishes are horses, beggars have horses. Therefore beggars ride." But, even though you have a proof of this statement, it's of no use to you, because you can never prove its premise: "wish = horse".

Chapter 2

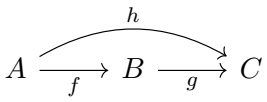
Composition

2.1 Composition

Programming is about composition. Paraphrasing Wittgenstein, one could say: "Of that which cannot be decomposed one should not speak." This is not a prohibition, it's a statement of fact. The process of studying, understanding, and describing is the same as the process of decomposing; and our language reflects this.

The reason we have built the vocabulary of objects and arrows is precisely to express the idea of composition.

Given an arrow f from A to B and an arrow g from B to C, their composition is an arrow that goes directly from A to C. In other words, if there are two arrows, the target of one being the same as the source of the other, we can always compose them to get a third arrow.



In math we denote composition using a little circle

$$h = g \circ f$$

We read this: "h is equal to g after f." The order of composition might seem backward, but this is because we think of functions as taking arguments on the right. In Haskell we replaced the circle with a dot:

```
h = g . f
```

This is every program in a nutshell. In order to accomplish `h`, we decompose it into simpler problems, `f` and `g`. These, in turn, can be decomposed further, and so on.

Now suppose that we were able to decompose g itself into $j \circ k$. We have

$$h = (j \circ k) \circ f$$

We want this decomposition to be the same as

$$h = j \circ (k \circ f)$$

We want to be able to say that we have decomposed h into three simpler problems

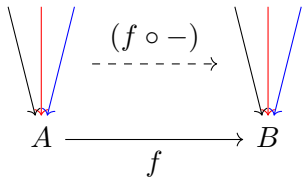
$$h = j \circ k \circ f$$

and not have to keep track which decomposition came first. This is called *associativity* of composition, and we will assume it from now on.

Composition is the source of two mappings of arrows called pre-composition and post-composition.

When an arrow f is post-composed after an arrow h, it produces the arrow $f \circ h$. Of course, f can be post-composed only after arrows whose target is the source of f. Post-composition by f is written as $(f \circ -)$, leaving a hole for h. As Lao Tzu would say, "Usefulness of post-composition comes from what is not there."

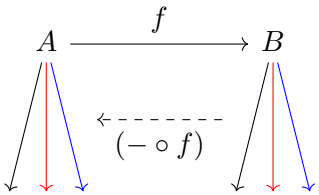
Thus an arrow $f\colon A \to B$ induces a mapping of arrows $(f \circ -)$ that maps arrows which are probing A to arrows which are probing B.



Since objects have no internal structure, when we say that f transforms A to B, this is exactly what we mean.

Post-composition lets us shift focus from one object to another.

Dually, you can pre-compose f, or apply $(- \circ f)$ to arrows originating in B and map them to arrows originating in A.



Pre-composition let us shift the perspective from one observer to another. Notice that the outgoing arrows are mapped in the direction opposite to the shifting arrow f.

Another way of looking at pre- and post-composition is that they are the result of partial application of the two-hole composition operator $(- \circ -)$, in which we pre-fill one hole or the other with an arrow.

In programming, an outgoing arrow is interpreted as extracting data from the source. An incoming arrow is interpreted as producing or constructing the target. Outgoing arrows define the interface, incoming arrows define the constructors.

Do the following exercises to convince yourself that shifts in focus and perspective are composable.

Exercise 2.1.1. *Suppose that you have two arrows, $f\colon A \to B$ and $g\colon B \to C$. Their composition $g \circ f$ induces a mapping of arrows $((g \circ f) \circ -)$. Show that the result is the same if you first apply $(f \circ -)$ and follow it by $(g \circ -)$. Symbolically:*

$$((g \circ f) \circ -) = (g \circ -) \circ (f \circ -)$$

Hint: Pick an arbitrary object X and an arrow $h\colon X \to A$ and see if you get the same result.

Exercise 2.1.2. *Convince yourself that the composition from the previous exercise is associative. Hint: Start with three composable arrows.*

Exercise 2.1.3. *Show that pre-composition $(- \circ f)$ is composable, but the order of composition is reversed:*

$$(- \circ (g \circ f)) = (- \circ f) \circ (- \circ g)$$

2.2 Function application

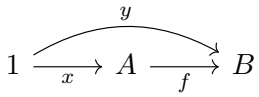
We are ready to write our first program. There is a saying: "A journey of a thousand miles begins with a single step." Our journey is from 1 to B. The single step is an arrow from the terminal object 1 to A. It's an element of A. We can write it as

$$1 \xrightarrow{x} A$$

The rest of the journey is the arrow

$$A \xrightarrow{f} B$$

These two arrows are composable (they share the object A) and their composition is the arrow y from 1 to B. In other words, y is an *element* of B



We can write it as

$$y = f \circ x$$

We used f to map an *element* of A to an *element* of B. Since this is something we do quite often, we call it the *application* of a function f to x, and use the shorthand notation

$$y = fx$$

Let's translate it to Haskell. We start with an element of A (a shorthand for `()->A`)

```
x :: A
```

We declare the function f as an element of the "object of arrows" from A to B

```
f :: A -> B
```

with the understanding (which will be elaborated upon later) that it corresponds to an arrow from `A` to `B`. The result is an element of B

```
y :: B
```

and it is defined as

```
y = f x
```

We call this the application of a function to an argument, but we expressed it purely in terms of function composition. (Note: In other programming languages function application requires the use of parentheses, e.g., `y = f(x)`.)

2.3 Identity

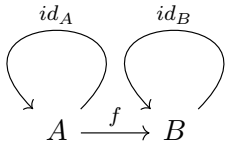
You may think of arrows as representing change: object A becomes object B. An arrow that loops back represents a change in an object itself. But change has its dual: lack of change, inaction or, as Lao Tzu would say *wu wei*.

Every object has a special arrow called the identity, which leaves the object unchanged. It means that, when you compose this arrow with any other arrow, either incoming or outgoing, you get that arrow back. Identity arrow does nothing to an arrow.

An identity arrow on the object A is called id_A. So if we have an arrow $f\colon A \to B$, we can compose it with identities on either side

$$id_B \circ f = f = f \circ id_A$$

or, pictorially:



We can easily check what an identity does to elements. Let's take an element $x\colon 1 \to A$ and compose it with id_A. The result is:

$$id_A \circ x = x$$

which means that identity leaves elements unchanged.

In Haskell, we use the same name `id` for all identity functions (we don't subscript it with the type it's acting on). The above equation, which specifies the action of *id* on elements, translates directly to

```
id x = x
```

and it becomes the definition of the function `id`.

We've seen before that both the initial object and the terminal object have unique arrows circling back to them. Now we are saying that every object has an identity arrow circling back to it. Remember what we said about uniqueness: If you can find two of those, then they must be equal. We must conclude that these unique looping arrows we talked about must be the identity arrows. We can now label these diagrams:

In logic, identity arrow translates to a tautology. It's a trivial proof that, "if A is true then A is true." It's also called the *identity rule*.

If identity does nothing then why do we care about it? Imagine going on a trip, composing a few arrows, and finding yourself back at the starting point. The question is: Have you done anything, or have you wasted your time? The only way to answer this question is to compare your path with the identity arrow.

Some round trips bring change, others don't.

Exercise 2.3.1. *What does* $(id_A \circ -)$ *do to arrows terminating in* A*? What does* $(- \circ id_A)$ *do to arrows originating from* A*?*

Chapter 3

Isomorphisms

When we say that

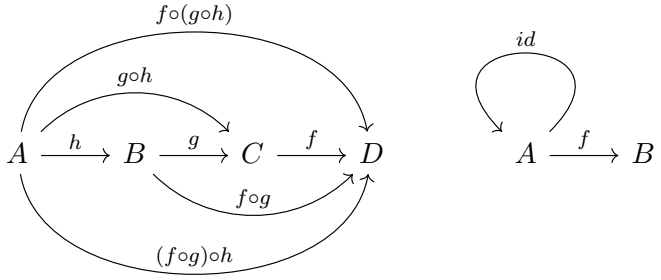
$$f \circ (g \circ h) = (f \circ g) \circ h$$

or

$$f = f \circ id$$

we are asserting the *equality* of arrows. The arrow on the left is the result of one operation, and the arrow on the right is the result of another. But the results are *equal*.

We often illustrate such equalities by drawing *commuting* diagrams, e.g.,



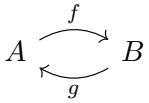
Thus we compare arrows for equality.

We do *not* compare objects for equality. We see objects as confluences of arrows, so if we want to compare two objects, we look at the arrows.

3.1 Isomorphic Objects

The simplest relation between two objects is an arrow.

The simplest round trip is a composition of two arrows going in opposite directions.



There are two possible round trips. One is $g \circ f$, which goes from A to A. The other is $f \circ g$, which goes from B to B.

If both of them result in identities, than we say that g is the *inverse* of f

$$g \circ f = id_A$$

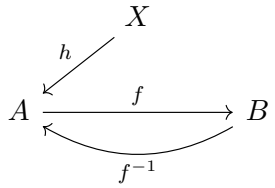
$$f \circ g = id_B$$

and we write it as $g = f^{-1}$ (pronounced f *inverse*). The arrow f^{-1} undoes the work of the arrow f.

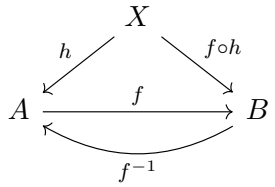
Such a pair of arrows is called an *isomorphism* and the two objects are called *isomorphic*.

What does the existence of an isomorphisms tell us about the two objects they connect?

We have said that objects are described by their interactions with other objects. So let's consider what the two isomorphic objects look like from the perspective of an observer object X. Take an arrow h coming from X to A.



There is a corresponding arrow coming from X to B. It's just the composition of $f \circ h$, or the action of $(f \circ -)$ on h.



Similarly, for any arrow probing B there is a corresponding arrow probing A. It is given by the action of $(f^{-1} \circ -)$.

We can move focus back and forth between A and B using the mappings $(f \circ -)$ and $(f^{-1} \circ -)$.

We can combine these two mappings (see exercise 2.1.1) to form a round trip. The result is the same as if we applied the composite $((f^{-1} \circ f) \circ -)$. But this is equal to $(id_A \circ -)$ which, as we know from exercise 2.3.1, leaves the arrows unchanged.

Similarly, the round trip induced by $f \circ f^{-1}$ leaves the arrows $X \to B$ unchanged.

This creates a "buddy system" between the two groups of arrows. Imagine each arrow sending a message to its buddy, as determined by f or f^{-1}. Each arrow would then receive exactly one message, and it would be a message from its buddy. No arrow would be left behind, and no arrow would receive more than one message. Mathematicians call this kind of buddy system a *bijection* or one-to-one correspondence.

Therefore, arrow by arrow, the two objects A and B look exactly the same from the perspective of X. Arrow-wise, there is no difference between the two objects.

In particular, if you replace X with the terminal object 1, you'll see that the two objects have the same elements. For every element $x\colon 1 \to A$ there is a corresponding element $y\colon 1 \to B$, namely $y = f \circ x$, and vice versa. There is a bijection between the elements of isomorphic objects.

Such indistinguishable objects are called *isomorphic* because they have "the same shape." You've seen one, you've seen them all.

We write this isomorphism as:

$$A \cong B$$

When dealing with objects, we use isomorphism in place of equality.

In programming, two isomorphic types have the same external behavior. One type can be implemented in terms of the other and vice versa. One can be replaced by the other without changing the behavior of the system (except, possibly, the performance).

In classical logic, if B follows from A and A follows from B then A and B are logically equivalent. We often say that B is true "if and only if" A is true. However, unlike previous parallels between logic and type theory, this one is not as straightforward if you consider proofs to be relevant. In fact, it led to the development of a new branch of fundamental mathematics, homotopy type theory, or HoTT for short.

Exercise 3.1.1. *Make an argument that there is a bijection between arrows that are* outgoing *from two isomorphic objects. Draw the corresponding diagrams.*

Exercise 3.1.2. *Show that every object is isomorphic to itself*

Exercise 3.1.3. *If there are two terminal objects, show that they are isomorphic*

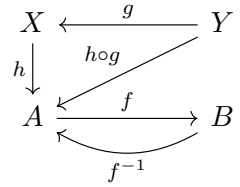
Exercise 3.1.4. *Show that the isomorphism from the previous exercise is unique.*

3.2 Naturality

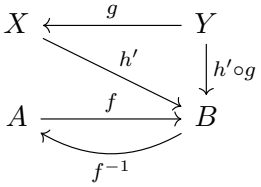
We've seen that, when two objects are isomorphic, we can switch focus from one to another using post-composition: either $(f \circ -)$ or $(f^{-1} \circ -)$.

Conversely, to switch between different observers, we use pre-composition.

Indeed, an arrow h probing A from X is related to the arrow $h \circ g$ probing the same object from Y.



Similarly, an arrow h' probing B from X corresponds to the arrow $h' \circ g$ probing it from Y.



In both cases, we change perspective from X to Y by applying precomposition $(- \circ g)$.

The important observation is that the change of perspective preserves the buddy system established by the isomorphism. If two arrows were buddies from the perspective

of X, they are still buddies from the perspective of Y. This is as simple as saying that it doesn't matter if you first pre-compose with g (switch perspective) and then post-compose with f (switch focus), or first post-compose with f and then pre-compose with g. Symbolically, we write it as:

$$(- \circ g) \circ (f \circ -) = (f \circ -) \circ (- \circ g)$$

and we call it the *naturality* condition.

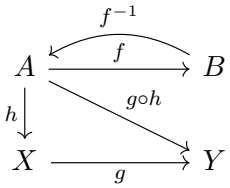
The meaning of this equation is revealed when you apply it to a morphism $h\colon X \to A$. Both sides evaluate to $f \circ h \circ g$.

$$Y \xrightarrow{g} X \xrightarrow{h} A \xrightarrow{f} B$$

Here, the naturality condition is just an identity, but we'll soon see it generalized to less trivial circumstances.

Arrows are used to broadcast information about an isomorphism. Naturality tells us that all objects get a consistent view of it, independent of the path.

We can also reverse the roles of observers and subjects. For instance, using an arrow $h\colon A \to X$, the object A can probe an arbitrary object X. If there is an arrow $g\colon X \to Y$, it can switch focus to Y. Switching the perspective to B is done by precomposition with f^{-1}.



Again, we have the naturality condition, this time from the point of view of the isomorphic pair:

$$(- \circ f^{-1}) \circ (g \circ -) = (g \circ -) \circ (- \circ f^{-1})$$

Exercise 3.2.1. *Show that both sides of the naturality condition for f^{-1}, when acting on h, reduce to:*

$$B \xrightarrow{f^{-1}} A \xrightarrow{h} X \xrightarrow{g} Y$$

3.3 Reasoning with Arrows

Master Yoneda says: "At the arrows look!"

If two objects are isomorphic, they have the same sets of incoming arrows.

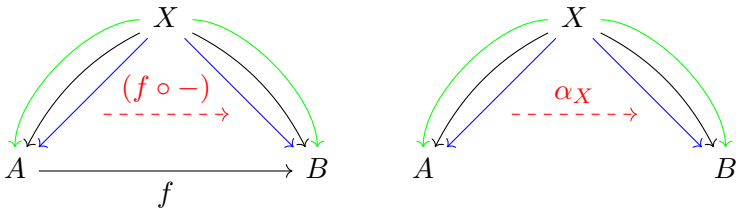
If two objects are isomorphic, they also have the same sets of outgoing arrows.

If you want to see if two objects are isomorphic, at the arrows look!

When two objects A and B are isomorphic, any isomorphism f induces a one-to-one mapping $(f \circ -)$ between corresponding sets of arrows.

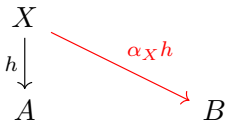
Suppose that we don't know if the objects are isomorphic, but we know that there is an invertible mapping, α_X, between sets of arrows impinging on A and B from every

object X.



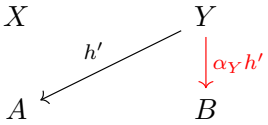
Does it mean that the two objects are isomorphic? Can we reconstruct the isomorphism f from the family of mappings α_X?

Here's the action of α_X on a particular arrow h.



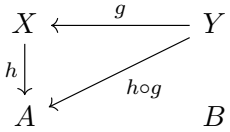
This mapping, along with its inverse α_X^{-1}, which takes arrows $X \to B$ to arrows $X \to A$, would play the role of $(f \circ -)$ and $(f^{-1} \circ -)$, if there was indeed an isomorphism f. The family of maps α describes an "artificial" way of switching focus between our two objects.

Here's the same situation from the point of view of another observer Y:



Notice that Y is using the mapping α_Y to switch focus from A to B.

These two mappings, α_X and α_Y, become entangled whenever there is a morphism $g\colon Y \to X$. In that case, pre-composition with g allows us to switch perspective from X to Y (notice the direction)



We have separated the switching of focus from the switching of perspective. The former is done by α, the latter by pre-composition.

But now, starting with some h, we can either apply $(- \circ g)$ to switch to Y's point of view, and the apply α_Y to switch focus to B

$$\alpha_Y \circ (- \circ g)$$

or we can first let X switch focus to B using α_X, and then switch perspective using $(- \circ g)$

$$(- \circ g) \circ \alpha_X$$

In both cases we end up looking at B from Y. We've done this exercise before, when we had an isomorphism between A and B, and we've found out that the results were the same. We called it the naturality condition.

If we want the αs to give us an isomorphism, we have to impose the equivalent naturality condition:

$$\alpha_Y \circ (- \circ g) = (- \circ g) \circ \alpha_X$$

Or, when acting on some arrow $h\colon X \to A$:

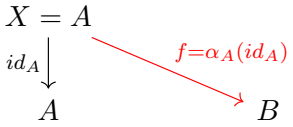
$$\alpha_Y (h \circ g) = (\alpha_X h) \circ g$$

This way, if we replace all αs with $(f \circ -)$, things will work out.

And indeed they do! This is called the Yoneda trick. It's a way to reconstruct f from the αs. Since α_X is defined for any object X, it is also defined for A. By definition, α_A takes a morphism $A \to A$ to a morphism $A \to B$. We know for sure that there is at least one such morphism, namely the identity id_A. The isomorphism f we are seeking is

$$f = \alpha_A(id_A)$$

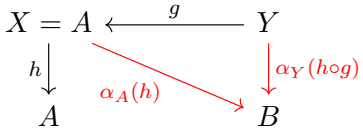
or, pictorially,



If this is indeed an isomorphism then our α_X should be equal to $(f \circ -)$ for any X. To see that, let's rewrite the naturality condition replacing X with A. We get

$$\alpha_Y (h \circ g) = (\alpha_A h) \circ g$$

as illustrated in the following diagram



Since both the source and the target of h is A, this equality must also be true for $h = id_A$

$$\alpha_Y (id_A \circ g) = (\alpha_A(id_A)) \circ g$$

But $\alpha_A(id_A)$ is our f, so we get

$$\alpha_Y g = f \circ g = (f \circ -)g$$

In other words, $\alpha_Y = (f \circ -)$ for every object Y and every morphism $g\colon Y \to A$

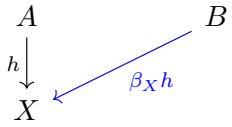
Even though α_X was defined individually for every X and every arrow $X \to A$, it turned out to be completely determined by its value at a single identity arrow. This is the power of naturality!

Reversing the Arrows

As Lao Tzu would say, the duality between the observer and the observed cannot be complete unless the observer is allowed to switch roles with the observed.

Again, we want to show that two objects A and B are isomorphic, but this time we want to treat them as observers. An arrow $h\colon A \to X$ probes an arbitrary object X

from the perspective of A. When we knew that the two objects were isomorphic, we were able to switch perspective to B using $(- \circ f^{-1})$. This time we have at our disposal a transformation β_X instead.



Similarly, if we want to observe another object, Y, we will use β_Y to switch perspectives between A and B, and so on.

If the two objects X and Y are connected by an arrow $g\colon X \to Y$ then we also have an option of switching focus using $(g \circ -)$. If we want to both switch perspective and switch focus, there are two ways of doing it. Naturality demands that the results be equal

$$(g \circ -) \circ \beta_X = \beta_Y \circ (g \circ -)$$

Indeed, if we replace β with $(- \circ f^{-1})$, we recover the naturality condition for an isomorphism.

Exercise 3.3.1. *Use the trick with the identity morphism to recover f^{-1} from the family of mappings β.*

Exercise 3.3.2. *Using f^{-1} from the previous exercise, evaluate $\beta_Y g$ for an arbitrary object Y and an arbitrary arrow $g\colon A \to Y$.*

As Lao Tzu would say: To show an isomorphism, it is often easier to define a natural transformation between ten thousand arrows than it is to find a pair of arrows between two objects.

Chapter 4

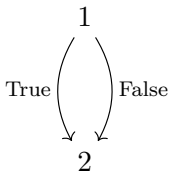
Sum Types

4.1 Bool

We know how to compose arrows. But how do we compose objects?

We have defined 0 (the initial object) and 1 (the terminal object). What is 2 if not 1 plus 1?

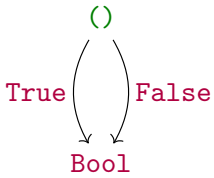
A 2 is an object with two elements: two arrows coming from 1. Let's call one arrow `True` and the other `False`. Don't confuse those names with the logical interpretations of the initial and the terminal objects. These two are *arrows*.



This simple idea can be immediately expressed in Haskell as the definition of a type, traditionally called `Bool`, after its inventor George Boole (1815-1864).

```
data Bool where
  True  :: () -> Bool
  False :: () -> Bool
```

It corresponds to the same diagram, only with some Haskell renamings:



As we've seen before, there is a shortcut notation for elements, so here's a more compact version:

```
data Bool where
  True  :: Bool
  False :: Bool
```

We can now define a term of the type `Bool`, for instance

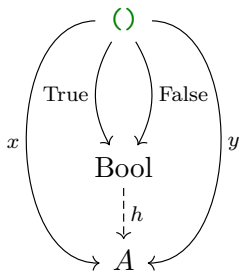
```
x :: Bool
x = True
```

The first line declares `x` to be an element of `Bool` (really, a function `()->Bool`), and the second line tells us which one of the two.

The functions `True` and `False` that we used in the definition of `Bool` are called *data constructors*. They can be used to construct specific terms, like in the example above. As a side note, in Haskell, function names start with lower-case letters, except when they are data constructors.

Our definition of the type `Bool` is still incomplete. We know how to construct a `Bool` term, but we don't know what to do with it. We have to be able to define arrows that go out of `Bool`—the *mappings out* of `Bool`.

The first observation is that, if we have an arrow `h` from `Bool` to some type `A` then we automatically get two arrows x and y from unit to `A`, just by composition. The following diagram commutes:



In other words, every function `Bool->A` produces a pair of elements of `A`.

Given a concrete type `A`:

```
h :: Bool -> A
```

we have:

```
x = h True
y = h False
```

where

```
x :: A
y :: A
```

Notice the use of the shorthand notation for the application of a function to an element:

```
h True -- meaning: h . True
```

We are now ready to complete our definition of `Bool` by adding the condition that any function from `Bool` to `A` not only produces but *is equivalent* to a pair of of elements of `A`. In other words, a pair of elements uniquely determines a function from `Bool`.

What this means is that we can interpret the diagram above in two ways: Given `h`, we can easily get `x` and `y`. But the converse is also true: a pair of elements `x` and `y` uniquely *defines* `h`.

We have a "buddy system," or a bijection, at work here. This time it's a one-to-one mapping between a pair of elements (x, y) and an arrow h.

In Haskell, this definition of `h` is encapsulated in the `if`, `then`, `else` construct. Given

```
x :: A
y :: A
```

we define the mapping out

```
h :: Bool -> A
h b = if b then x else y
```

Here, `b` is a term of type `Bool`.

In general, a data type is created using *introduction* rules and deconstructed using *elimination* rules. The `Bool` data type has two introduction rules, one using `True` and another using `False`. The `if`, `then`, `else` construct defines the elimination rule.

The fact that, given `h`, we can reconstruct the two terms used to define it, is called the *computation* rule. It tells us how to compute the result of `h`. If we call `h` with `True`, then the result is `x`; if we call it with `False`, the result is `y`.

We should never lose sight of the purpose of programming: to decompose complex problems into a series of simpler ones. The definition of `Bool` illustrates this idea. Whenever we have to construct a mapping out of `Bool`, we decompose it into two smaller tasks of constructing a pair of elements of the target type. We traded one larger problem for two simpler ones.

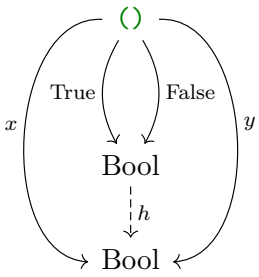
Examples

Let's do a few examples. We haven't defined many types yet, so we'll be limited to mappings out from `Bool` to either `Void`, `()`, or `Bool`. Such edge cases, however, may offer new insights into well known results.

We have decided that there can be no functions (other than identity) with `Void` as a target, so we don't expect any functions from `Bool` to `Void`. And indeed, we have zero pairs of elements of `Void`.

What about functions from `Bool` to `()`? Since `()` is terminal, there can be only one function from `Bool` to it. And, indeed, this function corresponds to the single possible pair of functions from `()` to `()`—both being identities. So far so good.

The interesting case is functions from `Bool` to `Bool`. Let's plug `Bool` in place of `A` in our diagram:



How many pairs of functions from `()` to `Bool` do we have at our disposal? There are only two such functions, `True` and `False`, so we can form four pairs. These are $(True, True)$, $(False, False)$, $(True, False)$, and $(False, True)$. Therefore there can only be four functions from `Bool` to `Bool`.

We can write them in Haskell using the `if`, `then`, `else` construct. For instance, the last one, which we'll call `not` is defined as:

```
not :: Bool -> Bool
not b = if b then False else True
```

We can also look at functions from `Bool` to `A` as elements of the object of arrows, or the exponential A^2, where 2 is the `Bool` object. Accoording to our count, we have zero elements in 0^2, one element in 1^2, and four elements in 2^2. This is exactly what we'd expect from high-school algebra, where numbers actually meant numbers.

Exercise 4.1.1. *Write the implementations of the three other functions* `Bool->Bool`.

4.2 Enumerations

What comes after 0, 1, and 2? An object with three data constructors. For instance:

```
data RGB where
  Red   :: RGB
  Green :: RGB
  Blue  :: RGB
```

If you're tired of redundant syntax, there is a shorthand for this type of definition:

```
data RGB = Red | Green | Blue
```

This introduction rule allows us to construct terms of the type `RGB`, for instance:

```
c :: RGB
c = Blue
```

To define mappings out of `RGB`, we need a more general elimination pattern. Just like a function from `Bool` was determined by two elements, a function from `RGB` to `A` is determined by a triple of elements of `A`: `x`, `y`, and `z`. We can write such function using *pattern matching*:

```
h :: RGB -> A
h Red   = x
h Green = y
h Blue  = z
```

This is just one function whose definition is split into three cases.

It's possible to use the same syntax for `Bool` as well, in place of `if`, `then`, `else`:

```
h :: Bool -> A
h True  = x
h False = y
```

In fact, there is a third way of writing the same thing using the `case` pattern:

```
h c = case c of
  Red   -> x
  Green -> y
  Blue  -> z
```

or even

```
h :: Bool -> A
h b = case b of
```

```
  True  -> x
  False -> y
```

You can use any of these at your convenience when programming.

These patterns will also work for types with four, five, and more data constructors. For instance, a decimal digit is one of:

```
data Digit = Zero | One | Two | Three | ... | Nine
```

There is a giant enumeration of Unicode characters called `Char`. Their constructors are given special names: you write the character itself between two apostrophes, e.g.,

```
c :: Char
c = 'a'
```

A pattern of ten thousand things would take many years to complete, therefore people came up with the wildcard pattern, the underscore, which matches everything.

Because the patterns are matched in order, you should make the wildcard pattern the last:

```
yesno :: Char -> Bool
yesno c = case c of
  'y' -> True
  'Y' -> True
  _   -> False
```

But why should we stop at that? The type `Int` could be thought of as an enumeration of integers in the range between -2^{29} and 2^{29} (or more, depending on the implementation). Of course, exhaustive pattern matching on such ranges is out of the question, but the principle holds.

In practice, the types `Char` for Unicode characters, `Int` for fixed-precision integers, `Double` for double-precision floating point numbers, and several others, are built into the language.

These are not infinite types. Their elements can be enumerated, even if it takes ten thousand years. The type `Integer` is infinite, though.

Short Haskell Digression

Since we are going to write more Haskell code, we have to establish some preliminaries. To define data types using functions, we need to use the language pragma called `GADTs` (it stands for *Generalized Algebraic Data Types*). The pragma has to be put at the top of the source file. For instance:

```
{-# language GADTs #-}

data Bool where
  True  :: () -> Bool
  False :: () -> Bool
```

The `Void` data type can be defined as:

```
data Void where
```

with the empty `where` clause (no data constructor!).

The function `absurd` works with any type as its target (it's a *polymorphic* function), so it is parameterized by a *type variable*. Unlike concrete types, type variables must start with a lowercase letter. Here, `a` is such a type variable:

```
absurd :: Void -> a
absurd v = undefined
```

We use `undefined` to placate the compiler. In this case, we are absolutely sure that the function `absurd` can never be called, because it's impossible to construct an argument of type `Void`.

You can use `undefined` when you're only interested in compiling, as opposed to running, your code. For instance, you may need to plug a function `f` to check if your definitions work together:

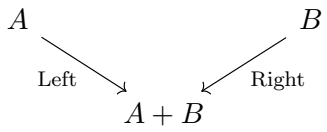
```
f :: a -> x
f = undefined
```

If you want to experiment with defining your own versions of standard types, like `Either`, you have to tell the compiler to hide the originals that are defined in the standard library called the `Prelude`. Put this line at the top of the file, after the language pragmas:

```
import Prelude hiding (Either, Left, Right)
```

4.3 Sum Types

The `Bool` type could be seen as the sum $2 = 1 + 1$. But nothing stops us from replacing 1 with another type, or even replacing each of the 1s with different types. We can define a new type $A + B$ by using two arrows. Let's call them `Left` and `Right`. The defining diagram is the introduction rule:

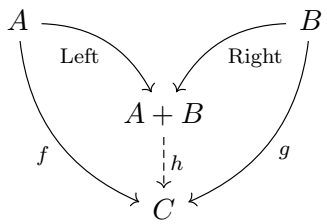


In Haskell, the type $A + B$ is called `Either a b`. By analogy with `Bool`, we can define it as

```
data Either a b where
  Left  :: a -> Either a b
  Right :: b -> Either a b
```

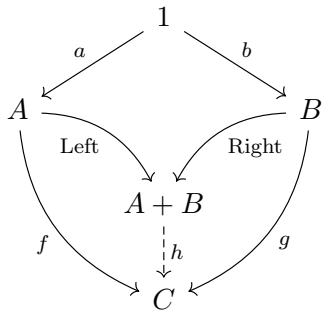
(Note the use of lower-case letters for type variables.)

Similarly, the mapping out from $A + B$ to some type C is determined by this commuting diagram:



Given a function h, we get a pair of functions f and g just by composing it with `Left` and `Right`. Conversely, such a pair of functions uniquely determines h. This is the elimination rule.

When we want to translate this diagram to Haskell, we need to select elements of the two types. We can do it by defining the arrows a and b from the terminal object.



Follow the arrows in this diagram to get

$$h \circ \text{Left} \circ a = f \circ a$$

$$h \circ \text{Right} \circ b = g \circ b$$

Haskell syntax repeats these equations almost literally, resulting in this pattern-matching syntax for the definition of `h`:

```
h :: Either a b -> c
h (Left  a) = f a
h (Right b) = g b
```

(Again, notice the use of lower-case letters for type variables and the same letters for terms of that type. Unlike humans, the compilers don't get confused by this.)

You can also read these equations right to left, and you will see the computation rules for sum types. The two functions that were used to define `h` can be recovered by applying `h` to terms constructed using `Left` and `Right`.

You can also use the `case` syntax to define `h`:

```
h e = case e of
  Left  a -> f a
  Right b -> g b
```

So what is the essence of a data type? It is but a recipe for manipulating arrows.

Maybe

A very useful data type, `Maybe` is defined as a sum $1+A$, for any A. This is its definition in Haskell:

```
data Maybe a where
  Nothing :: () -> Maybe a
  Just    ::  a -> Maybe a
```

The data constructor `Nothing` is an arrow from the unit type, and `Just` constructs `Maybe a` from `a`. `Maybe a` is isomorphic to `Either () a`. It can also be defined using the shorthand notation

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

`Maybe` is mostly used to encode the return type of partial functions: ones that are undefined for some values of their arguments. In that case, instead of failing, such functions return `Nothing`. In other programming languages partial functions are often implemented using exceptions.

Logic

In logic, the proposition $A + B$ is called the alternative, or *logical or*. You can prove it by providing the proof of A or the proof of B. Either one will suffice.

If you want to prove that C follows from $A + B$, you have to be prepared for two eventualities: either somebody proved $A + B$ by proving A or by proving B. In the first case, you have to show that C follows from A. In the second case you need a proof that C follows from B. These are exactly the arrows in the elimination rule for $A + B$.

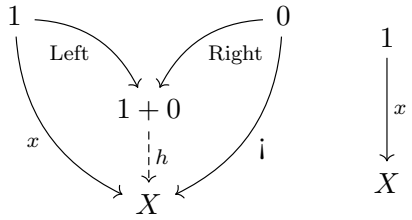
4.4 Cocartesian Categories

In Haskell, we can define a sum of any two types using `Either`. A category in which all sums exist, and the initial object exists, is called *cocartesian*, and the sum is called a *coproduct*. You might have noticed that sum types mimic addition of numbers. It turns out that the initial object plays the role of zero.

One Plus Zero

Let's first show that $1 + 0 \cong 1$, meaning the sum of the terminal object and the initial object is isomorphic to the terminal object. The standard procedure for this kind of proofs is to use the Yoneda trick. Since sum types are defined by mapping out, we should compare arrows coming *out* of either side.

Look at the definition of $1 + 0$ and it's mapping out to any object X. It's defined by a pair $(x, ¡)$, where x is an element of X and ¡ is the unique arrow from the initial object to X (the `absurd` function in Haskell).



We want to establish a one-to-one mapping between arrows originating in $1 + 0$ and the ones originating in 1. Since h is determined by the pair $(x, ¡)$, we can simply map it to the arrow x originating in 1. Since there is only one ¡, the mapping is a bijection.

So our β_X maps any pair $(x, ¡)$ to x. Conversely, β_X^{-1} maps x to the pair $(x, ¡)$. But is it a natural transformation?

To answer that, we need to consider what happens when we change focus from X to some Y that is connected to it through an arrow $g\colon X \to Y$. We have two options now:

- Make h switch focus by post-composing both x and ¡ with g. We get a new pair $(y = g \circ x, ¡)$. Follow it by β_Y.

- Use β_X to map $(x, ¡)$ to x. Follow it with the post-composition $(g \circ -)$.

In both cases we get the same arrow $y = g \circ x$. So the transformation β is natural. Therefore $1 + 0$ is isomorphic to 1.

In Haskell, we can define the two functions that form the isomorphism, but there is no way of directly expressing the fact that they are the inverse of each other.

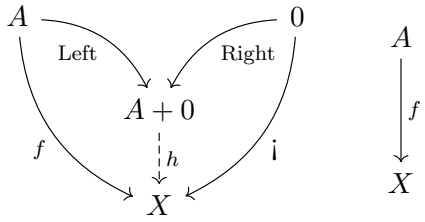
```
f :: Either () Void -> ()
f (Left ()) = ()
f (Right _) = ()

f_1 :: () -> Either () Void
f_1 _ = Left ()
```

The underscore wildcard in a function definition means that the argument is ignored. The second clause in the definition of `f` is redundant, since there are no terms of the type `Void`.

A Plus Zero

A very similar argument can be used to show that $A + 0 \cong A$. The following diagram explains it.



We can translate this argument to Haskell by implementing a (polymorphic) function `h` that works for any type `a`.

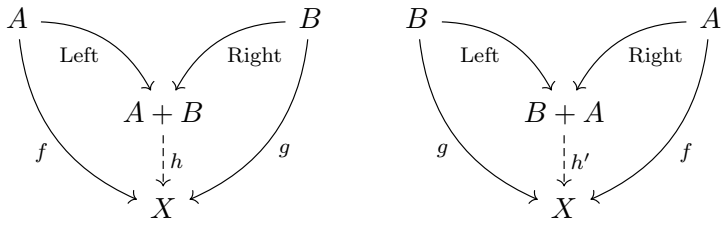
Exercise 4.4.1. *Implement, in Haskell, the two functions that form the isomorphism between* `Either a Void` *and* `a`.

We could use a similar argument to show that $0 + A \cong A$, but there is a more general property of sum types that obviates that.

Commutativity

There is a nice left-right symmetry in the diagrams that define the sum type which suggests that it satisfies the commutativity rule, $A + B \cong B + A$.

Let's consider mappings out of both sides. You can easily see that, for every h that is determined by a pair (f, g) on the left, there is a corresponding h' given by a pair (g, f) on the right. That establishes the bijection of arrows.



Exercise 4.4.2. *Show that the bijection defined above is natural. Hint: Both f and g change focus by post-composition with $k\colon X \to Y$.*

Exercise 4.4.3. *Implement, in Haskell, the function that witnesses the isomorphism between* `Either a b` *and* `Either b a`*. Notice that this function is its own inverse.*

Associativity

Just like in arithmetic, the sum that we have defined is associative:

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

It's easy to write the mapping out for the left hand side:

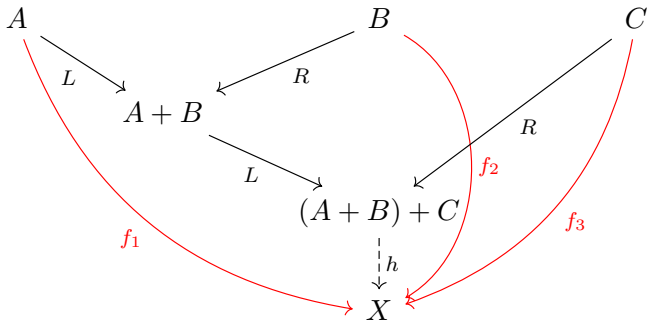
```
h :: Either (Either a b) c -> x
h (Left (Left a))  = f1 a
h (Left (Right b)) = f2 b
h (Right c)        = f3 c
```

Notice the use of nested patterns like `(Left (Left a))`, etc. The mapping is fully defined by a triple of functions. The same functions can be used to define the mapping out of the right hand side:

```
h' :: Either a (Either b c) -> x
h' (Left a)          = f1 a
h' (Right (Left b))  = f2 b
h' (Right (Right c)) = f3 c
```

This establishes a one-to-one mapping between triples of functions that define the two mappings out. This mapping is natural because all changes of focus are done using post-composition. Therefore the two sides are isomorphic.

This code can also be displayed in diagrammatical form. Here's the left hand side diagram.



Functoriality

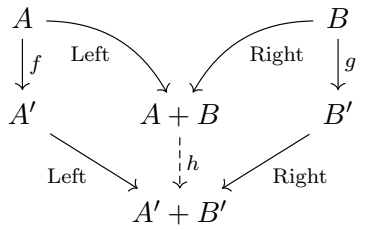
Since the sum is defined by the mapping out property, it was easy to see what happens when we change focus: it changes "naturally" with the foci of the arrows that define the product. But what happens when we move the sources of those arrows?

Suppose that we have arrows that map A and B to some A' and B':

$$f\colon A \to A'$$

$$g\colon B \to B'$$

The composition of these arrows with the constructors Left and Right, respectively, can be used to define the mapping of the sums:



The pair of arrows, $(\text{Left} \circ f, \text{Right} \circ g)$ uniquely defines the arrow $h\colon A + B \to A' + B'$.

This property of the sum is called *functoriality*. You can imagine it as allowing you to transform the two objects *inside* the sum.

Exercise 4.4.4. *Show that functoriality preserves composition. Hint: take two composable arrows, $g\colon B \to B'$ and $g'\colon B' \to B''$ and show that applying $g' \circ g$ gives the same result as first applying g to transform $A + B$ to $A + B'$ and then applying g' to transform $A + B'$ to $A + B''$.*

Exercise 4.4.5. *Show that functoriality preserves identity. Hint: use id_B and show that it is mapped to id_{A+B}.*

Symmetric Monoidal Category

When a child learns addition we call it arithmetics. When a grownup learns addition we call it a cocartesian category.

Whether we are adding numbers, composing arrows, or constructing sums of objects, we are re-using the same idea of decomposing complex things into their simpler components.

When things come together to form a new thing, and the operation is associative, and it has a neutral element, we know how to deal with ten thousand things.

The sum type we have defined satisfies these properties:

$$A + 0 \cong A$$

$$A + B \cong B + A$$

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

and it's functorial. A category with this type of operation is called *symmetric monoidal*. When the operation is the sum (coproduct), it's called *cocartesian*. In the next chapter we'll see another monoidal structure that's called *cartesian* without the "co."

Chapter 5

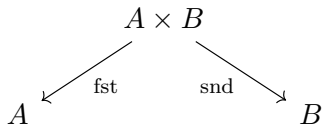
Product Types

We can use sum types to enumerate possible values of a given type, but the encoding can be wasteful. We needed ten constructors just to encode numbers between zero and nine.

```
data Digit = Zero | One | Two | Three | ... | Nine
```

But if we combine two digits into a single data structure, a two-digit decimal number, we'll able to encode a hundred numbers. Or, as Lao Tzu would say, with just four digits you can encode ten thousand numbers.

A data type that combines two types in this manner is called a product. Its defining quality is the elimination rule: there are two arrows coming from $A \times B$; one called "fst" goes to A, and another called "snd" goes to B. They are called *projections*. They let us retrive A and B from the product $A \times B$.

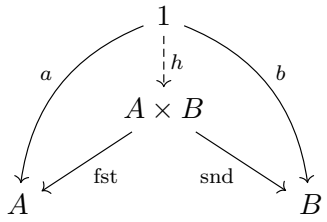


Suppose that somebody gave you an element of a product, that is an arrow h from the terminal object 1 to $A \times B$. You can easily retrieve a pair of elements, just by using composition: an element of A given by

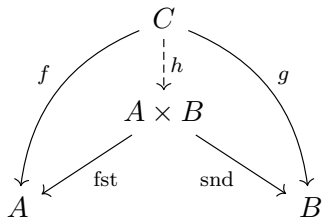
$$a = \text{fst} \circ h$$

and an element of B given by

$$b = \text{snd} \circ h$$



In fact, given an arrow from an arbitrary object C to $A \times B$, we can define, by composition, a pair of arrows $f \colon C \to A$ and $g \colon C \to B$



As we did before with the sum type, we can turn this idea around, and use this diagram to define the product type: A pair of functions f and g should be in one-to-one correspondence with a *mapping in* from C to $A \times B$. This is the *introduction* rule for the product.

In particular, the mapping out of the terminal object is used in Haskell to define a product type. Given two elements, `a :: A` and `b :: B`, we construct the product

```
(a, b) :: (A, B)
```

The built-in syntax for products is just that: a pair of parentheses and a comma in between. It works both for defining the product of two types `(A, B)` and the data constructor `(a, b)` that takes two elements and pairs them together.

We should never lose sight of the purpose of programming: to decompose complex problems into a series of simpler ones. We see it again in the definition of the product. Whenever we have to construct a mapping *into* the product, we decompose it into two smaller tasks of constructing a pair of functions, each mapping into one of the components of the product. This is as simple as saying that, in order to implement a function that returns a pair of values, it's enough to implement two functions, each returning one of the elements of the pair.

Logic

In logic, a product type corresponds to logical conjunction. In order to prove $A \times B$ (A and B), you need to provide the proofs of *both* A and B. These are the arrows targeting A and B. The elimination rule says that if you have a proof of $A \times B$, then you can get the proof of A (through fst) and the proof of B (through snd).

Tuples and Records

As Lao Tzu would say, a product of ten thousand objects is just an object with ten thousand projections.

We can form such products in Haskell using the tuple notation. For instance, a product of three types is written as `(A, B, C)`. A term of this type can be constructed from three elements: `(a, b, c)`.

In what mathematicians call "abuse of notation", a product of zero types is written as `()`, an empty tuple, which happens to be the same as the terminal object, or unit type. This is because the product behaves very much like multiplication of numbers, with the terminal object playing the role of unit.

In Haskell, rather than defining separate projections for all tuples, we use the pattern-matching syntax. For instance, to extract the third component from a triple we would write

```
thrd :: (a, b, c) -> c
thrd (_, _, c) = c
```

We use wildcards for the components that we want to ignore.

Lao Tzu said that "Naming is the origin of all particular things." In programming, keeping track of the meaning of the components of a particular tuple is difficult without giving them names. Record syntax allows us to give names to projections. This is the definition of a product written in record style:

```
data Product a b = Pair { fst :: a, snd :: b }
```

`Pair` is the data constructor and `fst` and `snd` are the projections.

This is how it could be used to declare and initialize a particular pair:

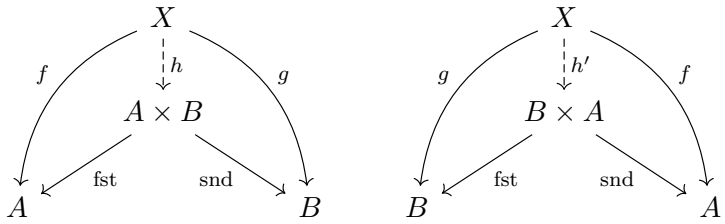
```
ic :: Product Int Char
ic = Pair 10 'A'
```

5.1 Cartesian Category

In Haskell, we can define a product of any two types. A category in which all products exist, and the terminal object exists, is called *cartesian*.

Tuple Arithmetic

The identities satisfied by the product can be derived using the mapping-in property. For instance, to show that $A \times B \cong B \times A$ consider the following two diagrams:



They show that, for any object X the arrows to $A \times B$ are in one-to-one correspondence with arrows to $B \times A$. This is because each of these arrows is determined by the same pair f and g.

You can check that the naturality condition is satisfied because, when you shift the perspective using $k \colon X' \to X$, all arrows originating in X are shifted by pre-composition $(- \circ k)$.

In Haskell, this isomorphism can be implemented as a function which is its own inverse:

```
swap :: (a, b) -> (b, a)
swap x = (snd x, fst x)
```

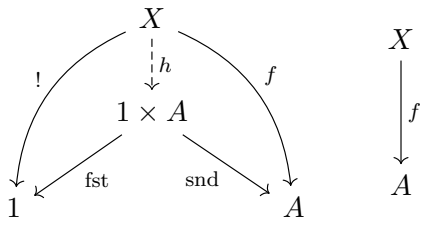
This is the same function written using pattern matching:

```
swap (x, y) = (y, x)
```

It's important to keep in mind that the product is symmetric only "up to isomorphism." It doesn't mean that swapping the order of pairs won't change the behavior

of a program. Symmetry means that the information content of a swapped pair is the same, but access to it needs to be modified.

Here's the diagram that can be used to prove that the terminal object is the unit of the product, $1 \times A \cong A$.



The unique arrow from X to 1 is called ! (pronounced, *bang*). Because of its uniqueness, the mapping-in, h, is totally determined by f.

The invertible arrow that witnesses the isomorphism between $1 \times A$ and A is called the *left unitor*:

$$\lambda \colon 1 \times A \to A$$

Here are some other isomorphisms written in Haskell (without proofs of having the inverse). This is associativity:

```
assoc :: ((a, b), c) -> (a, (b, c))
assoc ((a, b), c) = (a, (b, c))
```

And this is the left unit

```
lunit :: ((), a) -> a
lunit (_, a) = a
```

These functions correspond to the *associator*

$$\alpha \colon (A \times B) \times C \to A \times (B \times C)$$

and the *right unitor*:

$$\rho \colon A \times 1 \to A$$

Exercise 5.1.1. *Show that the bijection in the proof of left unit is natural. Hint, change focus using an arrow $g \colon A \to B$.*

Exercise 5.1.2. *Construct an arrow*

$$h \colon B + A \times B \to (1 + A) \times B$$

Is this arrow unique?

Hint: It's a mapping into a product, so it's given by a pair of arrow. These arrows, in turn, map out of a sum, so each is given by a pair of arrows.

Hint: The mapping $B \to 1 + A$ is given by $(Left \circ !)$

Exercise 5.1.3. *Re-do the previous exercise, this time treating h as a mapping* out *of a sum.*

Exercise 5.1.4. *Implement a Haskell function* `maybeAB :: Either b (a, b) -> (Maybe a, b)`. *Is this function uniquely defined by its type signature or is there some leeway?*

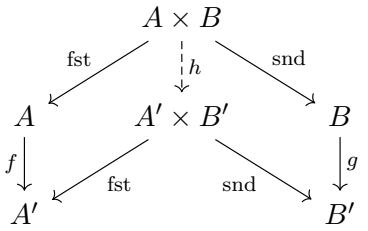
Functoriality

Suppose that we have arrows that map A and B to some A' and B':

$$f\colon A \to A'$$

$$g\colon B \to B'$$

The composition of these arrows with the projections fst and snd, respectively, can be used to define the mapping of the products:



The shorthand notation for this diagram is:

$$A \times B \xrightarrow{f \times g} A' \times B'$$

This property of the product is called *functoriality*. You can imagine it as allowing you to transform the two objects *inside* the product.

5.2 Duality

When a child sees an arrow, it knows which end points at the source, and which points at the target

$$A \to B$$

But maybe this is just a pre-conception. Would the Universe be very different if we called B the source and A the target?

We would still be able to compose this arrow with this one

$$B \to C$$

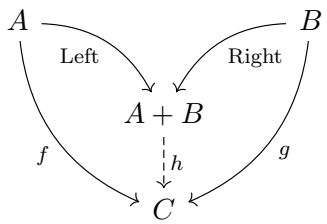
whose "target" B is the same as the same as the "source" of $A \to B$, and the result would still be an arrow

$$A \to C$$

only now we would say that it goes from C to A.

In this dual Universe, the object that we call "initial" would be called "terminal," because it's the "target" of unique arrows coming from all objects. Conversely, the terminal object would be called initial.

Now consider this diagram that we used to define the sum object:



In the new interpretation, the arrow h maps "from" an arbitrary object C "to" the object we call $A + B$. This arrow is uniquely defined by a pair of arrows (f, g) whose "source" is C. If we rename Left to fst and Right to snd, we will get the defining diagram for a product.

A product is the sum with arrows reversed.

Conversely, a sum is the product with arrows reversed.

Every construction in category theory has its dual.

If the direction of arrows is just a matter of interpretation, then what makes sum types so different from product types in programming? The difference goes back to one assumption we made at the start: There are no incoming arrows to the initial object (other than the identity arrow). This is in contrast with the terminal object having lots of outgoing arrows, which we used to define (global) elements. In fact, we assume that every object of interest has elements, and the ones that don't are isomorphic to `Void`.

We'll see an even deeper difference when we talk about function types.

5.3 Monoidal Category

We have seen that the product satisfies these simple rules:

$$1 \times A \cong A$$

$$A \times B \cong B \times A$$

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

and is functorial.

A category in which an operation with these properties is defined is called *symmetric monoidal*. We've seen this structure before when working with sums and the initial object.

A category can have multiple monoidal structures at the same time. When you don't want to name your monoidal structure, you replace the plus sign or the product sign with a tensor sign, and the neutral element with the letter I. The rules of a symmetric monoidal category can then be written as:

$$I \otimes A \cong A$$

$$A \otimes B \cong B \otimes A$$

$$(A \otimes B) \otimes C \cong A \otimes (B \otimes C)$$

You may think of a tensor product as the lowest common denominator of product and sum. It still has an introduction rule, but it requires both objects A and B; and it has no elimination rule—no projections. Some interesting examples of tensor products are not even symmetric.

Monoids

Monoids are very simple structures equipped with a binary operation and a unit. Natural numbers with addition and zero form a monoid. So do natural numbers with multiplication and one.

The intuition is that a monoid lets you combine two things to get one thing. There is also one special thing, such that combining it with anything else gives back the same thing. And the combining must be associative.

What's not assumed is that the combining is symmetric, or that there is an inverse element.

The rules that define a monoid are reminiscent of the rules of a category. The difference is that, in a monoid, any two things are composable, whereas in a category this is usually not the case: You can only compose two arrows if the target of one is the source of another. Except, that is, when the category contains only one object, in which case all arrows are composable.

A category with a single object is called a monoid. The combining operation is the composition of arrows and the unit is the identity arrow.

This is a perfectly valid definition. In practice, however, we are often interested in monoids that are embedded in larger categories. In particular, in programming, we want to be able to define monoids inside the category of types and functions.

In a category, we are forced to define the operation in bulk, rather than looking at individual elements. We start with an object M. A binary operation is a function of two arguments. Since elements of a product are pairs of elements, we can generalize it to an arrow from a product $M \times M$ to M:

$$\mu \colon M \times M \to M$$

The unit element can be defined as an arrow from the terminal object 1:

$$\eta \colon 1 \to M$$

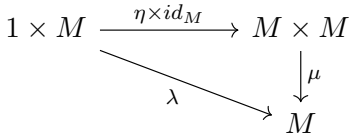
We can translate this description directly to Haskell by defining a class of types equipped with two methods, traditionally called `mappend` and `mempty`:

```
class Monoid m where
  mappend :: (m, m) -> m
  mempty  :: () -> m
```

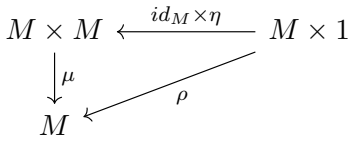
The two arrows μ and η have to satisfy monoid laws but, again, we have to formulate them in bulk, without any recourse to elements.

To formulate the left unit law, we first create the product $1 \times M$. We then use η to "pick the unit element in M" or, in terms of arrows, turn 1 into M. Since we are operating on a product $1 \times M$, we lift the pair $\langle \eta, id_M \rangle$, which ensures that we "do not touch" the M. We then perform the "multiplication" using μ.

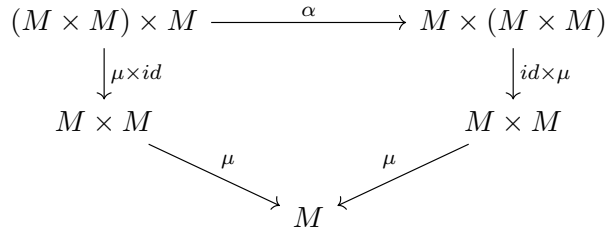
We want the result to be the same as the original element of M, but without mentioning elements. So we just use the left unitor λ to go from $1 \times M$ to M without "stirring things up."



Here is the analogous law for the right unit:



To formulate the law of associativity, we have to start with a triple product and act on it in bulk. Here, α is the associator that rearranges the product without "stirring things up."



Notice that we didn't have to assume a lot about the categorical product that we used on the objects M and 1. In particular we never had to use projections. This suggests that the above definition will work equally well for a tensor product in a monoidal category. It doesn't even have to be symmetric. All we have to assume is that there is a unit object, that the product is functorial, and that it satisfies the unit and associativity laws up to isomorphism.

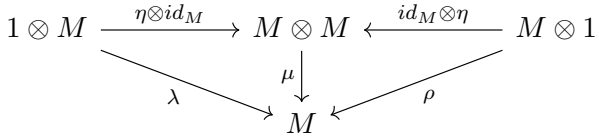
Thus if we replace $\times$ with $\otimes$ and 1 with I, we get a definition of a monoid in an arbitrary monoidal category.

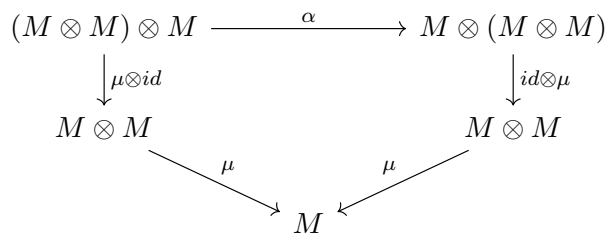
A *monoid* in a monoidal category is an object M equipped with two morphisms:

$$\mu \colon M \otimes M \to M$$

$$\eta \colon I \to M$$

satisfying the unit and associativity laws:





Chapter 6

Function Types

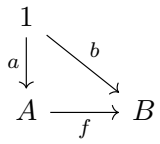
There is another kind of composition that is at the heart of functional programming. It happens when you pass a function as an argument to another function. The outer function can then use this argument as a pluggable part of its own machinery. It lets you implement, for instance, a generic sorting algorithm that accepts an arbitrary comparison function.

If we model functions as arrows between objects, then what does it mean to have a function as an argument?

We need a way to objectify functions in order to define arrows that have an "object of arrows" as a source or as a target. A function that takes a function as an argument or returns a function is called a *higher-order* function.

Elimination rule

The defining quality of a function is that it can be applied to an argument to produce the result. We have defined function application in terms of composition:



Here f is represented as an arrow from A to B, but we would like to be able to replace f with an element of the object of arrows or, as mathematicians call it, the exponential object B^A; or as we call it in programming, a function type `A->B`.

Given an element of B^A and an element of A, function application should produce an element of B. In other words, given a pair of elements

$$f \colon 1 \to B^A$$

$$a \colon 1 \to A$$

it should produce an element

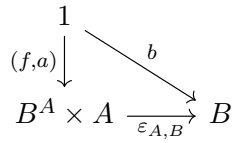
$$b \colon 1 \to B$$

Keep in mind that, here, f denotes an element of B^A. Previously, it was an arrow from A to B.

We know that a pair of elements (f, a) is equivalent to an element of a product $B^A \times A$. We can therefore define function application as a single arrow:

$$\varepsilon_{A,B} \colon B^A \times A \to B$$

This way b, the result of the application, is defined by this commuting diagram:



Function application is the *elimination rule* for function type.

When somebody gives you an element of the function object, the only thing you can do with it is to apply it to an element of the argument type using ε.

Introduction rule

To complete the definition of the function object, we also need the introduction rule.

First, suppose that there is a way of constructing a function object B^A from some other object C. It means that there is an arrow

$$h \colon C \to B^A$$

We know that we can eliminate the result of h using $\varepsilon_{A,B}$, but we have to first multiply it by A. So let's first multiply C by A and the use functoriality to map it to $B^A \times A$.

Functoriality lets us apply a pair of arrows to a product to get another product. Here, the pair of arrows is (h, id_A) (we want to turn C into B^A, but we're not interested in modifying A)

$$C \times A \xrightarrow{h \times id_A} B^A \times A$$

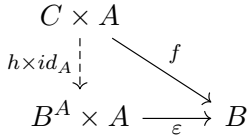
We can now follow this with function application to get to B

$$C \times A \xrightarrow{h \times id_A} B^A \times A \xrightarrow{\varepsilon_{A,B}} B$$

This composite arrow defines a mapping we'll call f:

$$f \colon C \times A \to B$$

Here's the corresponding diagram



This commuting diagram tells us that, given an h, we can construct an f; but we can also demand the converse: Every mapping out of a product, $f \colon C \times A \to B$ should uniquely define a mapping into the exponential, $h \colon C \to B^A$.

We can use this property, this one-to-one correspondence between two sets of arrows, to define the exponential object. This is the *introduction rule* for the function object.

We've seen that product was defined using its mapping-in property. Function application, on the other hand, is defined as a *mapping out* of a product.

Currying

There are several ways of looking at this definition. One is to see it as an example of currying.

So far we've been only considering functions of one argument. This is not a real limitation, since we can always implement a function of two arguments as a (single-argument) function from a product. The f in the definition of the function object is such a function:

```
f :: (C, A) -> B
```

`h` on the other hand is a function that returns a function (object)

```
h :: C -> (A -> B)
```

Currying is the isomorphism between these two types.

This isomorphism can be represented in Haskell by a pair of (higher-order) functions. Since, in Haskell, currying works for any types, these functions are written using type variables—they are *polymorphic*:

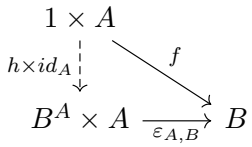
```
curry   :: ((c, a) -> b)   -> (c -> (a -> b))
```

```
uncurry :: (c -> (a -> b)) -> ((c, a) -> b)
```

In other words, the h in the definition of the function object can be written as

$$h = curry\, f$$

Of course, written this way, the types of `curry` and `uncurry` correspond to function objects rather than arrows. This distinction is usually glossed over because there is a one-to-one correspondence between the *elements* of the exponential and the *arrows* that define them. This is easy to see when we replace the arbitrary object C with the terminal object. We get:



In this case, h is an element of the object B^A, and f is an arrow from $1 \times A$ to B. But we know that $1 \times A$ is isomorphic to A so, effectively, f is an arrow from A to B.

Therefore, from now on, we'll call an arrow `->` an arrow $\to$, without making much fuss about it. The correct incantation for this kind of phenomenon is to say that the category is self-enriched.

We can write $\varepsilon_{A,B}$ as a Haskell function `apply`:

```
apply :: (a -> b, a) -> b
apply (f, x) = f x
```

but it's just a syntactic trick: function application is built into the language: `f x` means `f` applied to `x`. Other programming languages require the arguments to a function to be enclosed in parentheses, not so in Haskell.

Even though defining function application as a separate function may seem redundant, Haskell library does provide an infix operator `$` for that purpose:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

The trick, though, is that normal function application binds to the left, e.g., `f x y` is the same as `(f x) y`; the dollar sign binds to the right, so that `f $ g x` is the same as `f (g x)`. In the first example, `f` must be a function of (at least) two arguments; in the second, it could be a function of one argument.

In Haskell, currying is ubiquitous. A function of two arguments is almost always written as a function returning a function. Because the function arrow `->` binds to the right, there is no need to parenthesize such types. For instance, the pair constructor has the signature:

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

You may think of if as a function of two arguments returning a pair, or a function of one argument returning a function of one argument, `b->(a, b)`. This way it's okay to partially apply such a function, the result being another function. For instance, we can define:

```
pairWithTen :: a -> (Int, a)
pairWithTen = pair 10 -- partial application of pair
```

Relation to lambda calculus

Another way of looking at the definition of the function object is to interpret C as the type of the environment in which f is defined. In that case it's customary to call it Γ. The arrow is interpreted as an expression that can be defined in the environment Γ.

Consider a simple example, the expression:

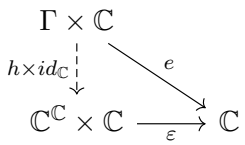
$$ax^2 + bx + c$$

You may think of it as being parameterized by a triple of real numbers (a, b, c) and a variable x, taken to be, let's say, a complex number. The triple is an element of a product $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$. This product is the environment Γ for our expression.

The variable x is an element of $\mathbb{C}$. The expression is an arrow from the product $\Gamma \times \mathbb{C}$ to the result type (here, also $\mathbb{C}$)

$$e \colon \Gamma \times \mathbb{C} \to \mathbb{C}$$

This is a mapping-out from a product, so we can use it to construct a function object $\mathbb{C}^{\mathbb{C}}$ and define a mapping $h \colon \Gamma \to \mathbb{C}^{\mathbb{C}}$



This new mapping h can be seen as a constructor of the function object. The resulting function object represents all functions from $\mathbb{C}$ to $\mathbb{C}$ that have access to the environment Γ; that is, to the triple of parameters (a, b, c).

Corresponding to our original expression $ax^2 + bx + c$ there is a particular arrow h that we write as:

$$\lambda x.\, ax^2 + bx + c$$

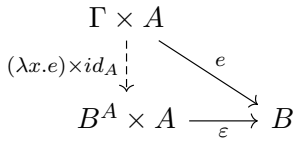
or, in Haskell, with the backslash replacing λ,

```
h = \x -> a * x^2 + b * x + c
```

This is just notation for the result of the one-to-one mapping between arrows: Given an arrow e that represents an expression in the context Γ, this mapping produces an arrow that we call $\lambda x.e$. In typed lambda calculus, we also specify the type of x:

$$h = \lambda\,(x\colon A).\,e$$

The defining diagram for the function object becomes



The environment Γ that provides free parameters for the expression e is a product of multiple objects representing the types of the parameters (in our example, it was $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$). An empty environment is represented by the terminal object, the unit of the product. In the latter case, h simply picks an element from the function object.

It's important to keep in mind that, in general, a function object represents functions that depend on external parameters. Such functions are called *closures*. Closures are functions that capture values from their environment.

Here's an example of a function returning a closure (for simplicity, we use `Double` for all types)

```
quadratic :: Double -> Double -> Double -> (Double -> Double)
quadratic a b c = \x -> a * x^2 + b * x + c
```

The same function can be written as a function of four variables:

```
quadratic :: Double -> Double -> Double -> Double -> Double
quadratic a b c x = a * x^2 + b * x + c
```

Modus ponens

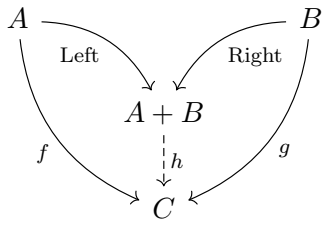
In logic, the function object corresponds to an implication. An arrow from the terminal object to the function object is the proof of an implication. Function application ε corresponds to what logicians call *modus ponens*: if you have a proof of the implication $A \Rightarrow B$ and a proof of A then this constitutes the proof of B.

6.1 Sum and Product Revisited

When functions gain the same status as elements of other types, we have the tools to directly translate diagrams into code.

Sum types

Let's start with the definition of the sum.



We said that the pair of arrows (f, g) uniquely determines the mapping h out of the sum. We can write it concisely using a higher-order function

```
h = mapOut (f, g)
```

where

```
mapOut :: (a -> c, b -> c) -> (Either a b -> c)
mapOut (f, g) = \aorb -> case aorb of
                           Left  a -> f a
                           Right b -> g b
```

This function takes a pair of functions as an argument and it returns a function.

First, we pattern-match the pair `(f, g)` to extract `f` and `g`. Then we construct a new function using a lambda. This lambda takes an argument of the type `Either a b`, which we call `aorb`, and does the case analysis on it. If it was constructed using `Left`, we apply `f` to its contents, otherwise we apply `g`.

Note that the function we are returning is a closure. It captures `f` and `g` from its environment.

The function we have implemented closely follows the diagram, but it's not written in the usual Haskell style. Haskell programmers prefer to curry functions of multiple arguments. Also, if possible, they prefer to eliminate lambdas.

Here's the version of the same function taken from the Haskell standard library, where it goes under the name (lower-case) `either`:

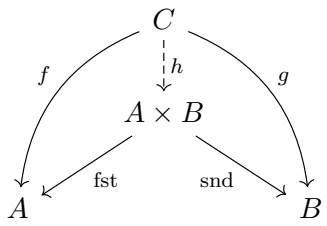
```
either                  :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)     =  f x
either _ g (Right y)    =  g y
```

The other direction of the bijection, from h to the pair (f, g), also follows the arrows of the diagram.

```
unEither :: (Either a b -> c) -> (a -> c, b -> c)
unEither h = (h . Left, h . Right)
```

Product types

Product types are dually defined by their mapping-in property.



Here's the direct Haskell reading of this diagram

```
h :: (c -> a, c -> b) -> (c -> (a, b))
h (f, g) = \c -> (f c, g c)
```

And this is the stylized version written in Haskell style as an infix operator `&&&`

```
(&&&) :: (c -> a) -> (c -> b) -> (c -> (a, b))
(f &&& g) c = (f c, g c)
```

The other direction of the bijection is given by:

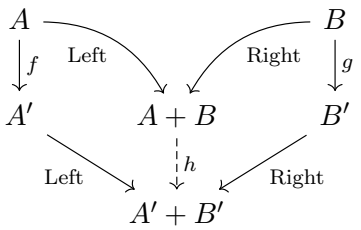
```
fork :: (c -> (a, b)) -> (c -> a, c -> b)
fork h = (fst . h, snd . h)
```

which also closely follows the reading of the diagram.

Functoriality revisited

Both sum and product are functorial, which means that we can apply functions to their contents. We are ready to translate those diagrams into code.

This is the functoriality of the sum type:



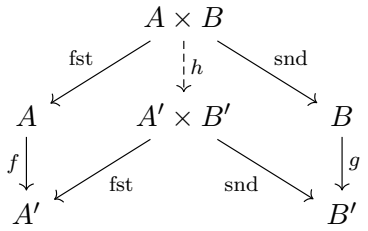
Reading this diagram we can immediately write h using `either`:

```
h f g = either (Left . f) (Right . g)
```

Or we could expand it and call it `bimap`:

```
bimap :: (a -> a') -> (b -> b') -> Either a b -> Either a' b'
bimap f g (Left  a) = Left  (f a)
bimap f g (Right b) = Right (g b)
```

Similarly for the product type:



h can be written as

```
h f g = (f . fst) &&& (g . snd)
```

Or it could be expanded to

```
bimap :: (a -> a') -> (b -> b') -> (a, b) -> (a', b')
bimap f g (a, b) = (f a, g b)
```

In both cases we call this higher-order function `bimap` since, in Haskell, both the sum and the product are instances of a more general class called `Bifunctor`.

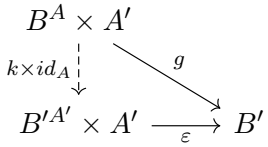
6.2 Functoriality of the Function Type

The function type, or the exponential, is also functorial, but with a twist. We are interested in a mapping from B^A to $B'^{A'}$, where the primed objects are related to the non-primed ones through some arrows, to be determined.

The exponential is defined by it's mapping-in property, so if we're looking for

$$k \colon B^A \to B'^{A'}$$

we should draw the diagram that has k as a mapping into $B'^{A'}$. We get this diagram from the original definition by substituting B^A for C and primed objects for the non-primed ones:



The question is: can we find an arrow g to complete this diagram?

$$g \colon B^A \times A' \to B'$$

If we find such a g, it will uniquely define our k.

The way to think about this problem is to consider how we would implement g. It takes the product $B^A \times A'$ as its argument. Think of it as a pair: an element of the function object from A to B and an element of A'. The only thing we can do with the function object is to apply it to something. But B^A requires an argument of type A, and all we have at our disposal is A'. We can't do anything unless somebody gives us an arrow $A' \to A$. This arrow applied to A' will generate the argument for B^A. However, the result of the application is of type B, and g is supposed to produce a B'. Again, we'll request an arrow $B \to B'$ to complete our assignment.

This may sound complicated, but the bottom line is that we require two arrows between the primed and non-primed objects. The twist is that the first arrow goes from A' to A, which feels backward from the usual. In order to map B^A to $B'^{A'}$ we are asking for a pair of arrows

$$f\colon A' \to A$$

$$g\colon B \to B'$$

This is somewhat easier to explain in Haskell. Our goal is to implement a function `a' -> b'`, given a function `h :: a -> b`.

This new function takes an argument of the type `a'` so, before we can pass it to `h`, we need to convert `a'` to `a`. That's why we need a function `f :: a' -> a`.

Since `h` produces a `b`, and we want to return a `b'`, we need another function `g :: b -> b'`. All this fits nicely into one higher-order function:

```
dimap :: (a' -> a) -> (b -> b') -> (a -> b) -> (a' -> b')
dimap f g h = g . h . f
```

Similar to `bimap` being an interface to the typeclass `Bifunctor`, `dimap` is a member of the typeclass `Profunctor`.

6.3 Bicartesian Closed Categories

A category in which both the product and the exponential is defined for any pair of objects, and which has a terminal object, is called *cartesian closed*. If it also has sums (coproducts) and the initial object, it's called *bicartesian closed*.

This is the minimum structure for modeling programming languages.

Data types constructed using these operations are called *algebraic data types*.

We have addition, multiplication, and exponentiation (but not subtraction or division) of types, with all the familiar laws we know from high-school algebra. They are satisfied up to isomorphism. There is one more algebraic law that we haven't discussed yet: distributivity.

Distributivity

Multiplication of numbers distributes over addition. Should we expect the same in a bicartesian closed category?

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

The left to right mapping is easy to construct, since it's simultaneously a mapping out of a sum and a mapping into a product. We can construct it by gradually decomposing it into simpler mappings. In Haskell, this means implementing a function

```
dist :: Either (b, a) (c, a) -> (Either b c, a)
```

A mapping out of the sum on the left is given by a pair of arrows:

$$f\colon B \times A \to (B + C) \times A$$

$$g\colon C \times A \to (B + C) \times A$$

We write it in Haskell as:

```
dist = either f g
  where
    f   :: (b, a) -> (Either b c, a)
    g   :: (c, a) -> (Either b c, a)
```

The `where` clause is used to introduce the definitions of sub-functions.

Now we need to implement f and g. They are mappings into the product, so each of them is equivalent to a pair of arrows. For instance, the first one is given by the pair:

$$f' \colon B \times A \to (B + C)$$

$$f'' \colon B \times A \to A$$

In Haskell:

```
    f = f' &&& f''
    f'  :: (b, a) -> Either b c
    f'' :: (b, a) -> a
```

The first arrow can be implemented by projecting the first component B and then using Left to construct the sum. The second is just the projection snd.

$$f' = \text{Left} \circ \text{fst}$$

$$f'' = \text{snd}$$

Combining all these together, we get:

```
dist = either f g
  where
    f   = f' &&& f''
    f'  = Left . fst
    f'' = snd
    g   = g' &&& g''
    g'  = Right . fst
    g'' = snd
```

These are the type signatures of the helper functions:

```
    f   :: (b, a) -> (Either b c, a)
    g   :: (c, a) -> (Either b c, a)
    f'  :: (b, a) -> Either b c
    f'' :: (b, a) -> a
    g'  :: (c, a) -> Either b c
    g'' :: (c, a) -> a
```

They can also be inlined to produce this terse form:

```
dist = either ((Left . fst) &&& snd) ((Right . fst) &&& snd)
```

This style of programming is called *point free* because it omits the arguments (points). For readability reasons, Haskell programmers prefer a more explicit style. The above function would normally be implemented as:

```
dist (Left  (b, a)) = (Left b, a)
dist (Right (c, a)) = (Right c, a)
```

Notice that we have only used the definitions of sum and product. The other direction of the isomorphism, though, requires the use of the exponential, so it's only valid in a bicartesian *closed* category. This is not immediately clear from the straightforward Haskell implementation:

```
undist :: (Either b c, a) -> Either (b, a) (c, a)
undist (Left b, a)  = Left (b, a)
undist (Right c, a) = Right (c, a)
```

but that's because currying is implicit in Haskell.

Here's the point-free version of this function:

```
undist = uncurry (either (curry Left) (curry Right))
```

This may not be the most readable implementation, but it underscores the fact that we need the exponential: we use both `curry` and `uncurry` to implement the mapping.

We'll come back to this identity when we are equipped with more powerful tools: adjunctions.

Exercise 6.3.1. *Show that:*

$$2 \times A \cong A + A$$

where 2 *is the Boolean type. Do the proof diagrammatically first, and then implement two Haskell functions witnessing the isomorphism.*

Chapter 7

Recursion

When you step between two mirrors, you see your reflection, the reflection of your reflection, the reflection of that reflection, and so on. Each reflection is defined in terms of the previous reflection, but together they produce infinity.

Recursion is a decomposition pattern that splits a single task into many steps, the number of which is potentially unbounded.

Recursion is based on suspension of disbelief. You are faced with a task that may take arbitrarily many steps. You tentatively assume that you know how to solve it. Then you ask yourself the question: "How would I make the last step if I had the solution to everything *but* the last step?"

7.1 Natural Numbers

An object of natural numbers N does not contain numbers. Objects have no internal structure. Structure is defined by arrows.

We can use an arrow from the terminal object to define one special element. By convention, we'll call this arrow Z for "zero."

$$Z\colon 1 \to N$$

But we have to be able to define infinitely many arrows to account for the fact that, for every natural number, there is another number that is one larger than it.

We can formalize this statement by saying: Suppose that we know how to create a natural number $n\colon 1 \to N$. How do we make the next step, the step that will point us to the next number—its successor?

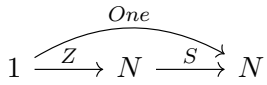
This next step doesn't have to be any more complex than just post-composing n with an arrow that loops back from N to N. This arrow should not be the identity, because we want the successor of a number to be different from that number. But a single such arrow, which we'll call S for "successor" will suffice.

The element corresponding to the successor of n is given by the composition:

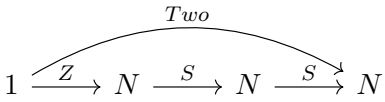
$$1 \xrightarrow{n} N \xrightarrow{S} N$$

(We sometimes draw the same object multiple times in a single diagram, if we want to straighten the looping arrows.)

In particular, we can define One as the successor of Z:



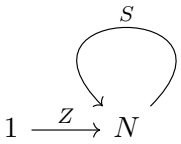
and Two as the successor of the successor of Z



and so on.

Introduction Rules

The two arrows, Z and S, serve as the introduction rules for the natural number object N. The twist is that one of them is recursive: S uses N as its source as well as its target.



The two introduction rules translate directly to Haskell

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

They can be used to define arbitrary natural numbers; for instance:

```
zero, one, two :: Nat
zero = Z
one  = S zero
two  = S one
```

This definition of natural number type is not very useful in practice. However, it's often used in defining type-level naturals, where each number is its own type.

You may encounter this construction under the name of *Peano arithmetic*.

Elimination Rules

The fact that the introduction rules are recursive complicates the matters slightly when it comes to defining elimination rules. We will follow the pattern from previous chapters of first assuming that we are given a mapping out of N:

$$h \colon N \to A$$

and see what we can deduce from there.

Previously, we were able to decompose such an h into simpler mappings (pairs of mappings for sum and product; a mapping out of a product for the exponential).

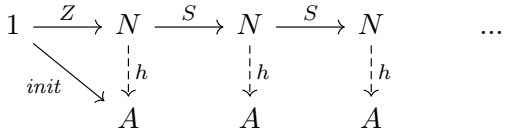
The introduction rules for N look similar to those for the sum, so we would expect that h could be split into two arrows corresponding to two introduction rules. And,

indeed, we can easily get the first one by composing $h \circ Z$. This is an arrow that picks an element of A. We call it *init*:

$$init \colon 1 \to A$$

But there is no obvious way to find the second one.

Let's try to plug h into the definition of N.



The intuition is that an arrow from N to A represents a *sequence* a_n of elements of A. The zeroth element is given by $a_0 = init$. The next element is

$$a_1 = h \circ S \circ Z$$

followed by

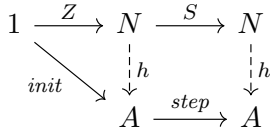
$$a_2 = h \circ S \circ S \circ Z$$

and so on.

We have thus replaced one arrow h with infinitely many arrows a_n. Granted, the new arrows are simpler, since they represent elements of A, but there are infinitely many of them.

The problem is that, no matter how you look at it, an arbitrary mapping out of N contains infinite amount of information.

We have to drastically simplify the problem. Since we used a single arrow S to generate all natural numbers, we can try to use a single arrow $A \to A$ to generate all the elements a_n. We'll call this arrow *step*:



The mappings out of N that are generated by such pairs, *init* and *step*, are called *recursive*. Not all mappings out of N are recursive. In fact very few are; but recursive mappings are enough to define the object of natural numbers.

We use the above diagram as the elimination rule. Every recursive mapping out of N is in one-to-one correspondence with a pair *init* and *step*.

This means that the *evaluation rule* (extracting $(init, step)$ for a given h) cannot be formulated for an arbitrary arrow $h \colon N \to A$, only for recursive arrows that have been defined using a pair $(init, step)$.

The arrow *init* can be always recovered by composing $h \circ Z$. The arrow *step* is a solution to the equation:

$$step \circ h = h \circ S$$

If h was defined using some *init* and *step*, then this equation obviously has a solution.

The important part is that we demand that this solution be *unique*.

Intuitively, the pair *init* and *step* generate the sequence of elements a_0, a_1, a_2, ... If two arrows h and h' are given by the same pair $(init, step)$, it means that the sequences they generate are the same.

So if h were different from h', it would mean that N contains more than just the sequence of elements Z, SZ, $S(SZ)$, ... For instance, if we added -1 to N (that is, made Z somebody's successor), we could have h and h' differ at -1 and yet be generated by the same *init* and *step*. Uniqueness means there are no natural number before, after, or in between the numbers generated by Z and S.

The elimination rule we've discussed here corresponds to *primitive recursion*. We'll see a more advanced version of this rule, corresponding to the induction principle, in the chapter on dependent types.

In Programming

The elimination rule can be implemented as a recursive function in Haskell:

```
rec :: a -> (a -> a) -> (Nat -> a)
rec init step = \n ->
  case n of
    Z     -> init
    (S m) -> step (rec init step m)
```

This single function, which is called a *recursor*, is enough to implement all recursive functions of natural numbers. For instance, this is how we could implement addition:

```
plus :: Nat -> Nat -> Nat
plus n = rec init step
  where
    init = n
    step = S
```

This function takes `n` as an argument and produces a function (a closure) that takes another number and adds `n` to it.

In practice, programmers prefer to implement recursion directly—an approach that is equivalent to inlining the recursor `rec`. The following implementation is arguably easier to understand:

```
plus n m = case m of
  Z -> n
  (S k) -> S (plus k n)
```

It can be read as: If `m` is zero then the result is `n`. Otherwise, if `m` is a successor of some `k`, then the result is the successor of `k + n`. This is exactly the same as saying that `init = n` and `step = S`.

In imperative languages recursion is often replaced by iteration. Conceptually, iteration seems to be easier to understand, as it corresponds to sequential decomposition. The steps in the sequence usually follow some natural order. This is in contrast with recursive decomposition, where we assume that we have done all the work up to the n'th step, and we combine that result with the next consecutive step.

On the other hand, recursion is more natural when processing recursively defined data structures, such as lists or trees.

The two approaches are equivalent, and compilers often convert recursive functions to loops in what is called *tail recursion optimization*.

Exercise 7.1.1. *Implement a curried version of addition as a mapping out of N into the function object N^N. Hint: use these types in the recursor:*

```
init :: Nat -> Nat
step :: (Nat -> Nat) -> (Nat -> Nat)
```

7.2 Lists

A list of things is either empty or a thing followed by a list of things.

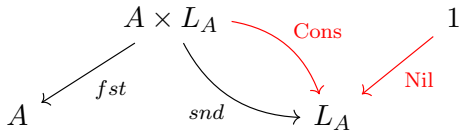
This recursive definition translates into two introduction rules for the type L_A, the list of A:

$$\text{Nil} \colon 1 \to L_A$$

$$\text{Cons} \colon A \times L_A \to L_A$$

The Nil element describes an empty list, and $Cons$ constructs a list from a head and a tail.

The following diagram depicts the relationship between projections and list constructors. The projections extract the head and the tail of the list that was constructed using $Cons$.

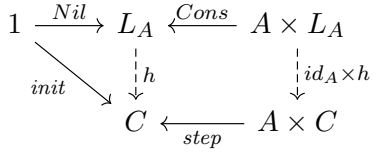


This description can be immediately translated to Haskell:

```
data List a where
  Nil  :: List a
  Cons :: (a, List a) -> List a
```

Elimination Rule

Given a mapping out, $h \colon L_A \to C$, from a list of A to some arbitrary type C, this is how we can plug it into the definition of the list:



We used the functoriality of the product to apply the pair (id_A, h) to the product $A \times L_A$.

Similar to the natural number object, we can try to define two arrows, $init = h \circ Nil$ and $step$. The arrow $step$ is a solution to:

$$step \circ (id_A \times h) = h \circ Cons$$

Again, not every h can be reduced to such a pair of arrows.

However, given $init$ and $step$, we can define an h. Such a function is called a *fold*, or a list catamorphism.

This is the list recursor in Haskell:

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
recList init step = \as ->
  case as of
    Nil          -> init
    Cons (a, as) -> step (a, recList init step as)
```

Given `init` and `step`, it produces a mapping out of a list.

A list is such a basic data type that Haskell has a built-in syntax for it. The type `List a` is written as `[a]`. The `Nil` constructor is an empty pair of square brackets, `[]`, and the `Cons` constructor is an infix colon `:`.

We can pattern match on these constructors. A generic mapping out of a list has the form:

```
h :: [a] -> c
h []       = -- empty-list case
h (a: as) = -- case for the head and the tail of a non-empty list
```

Corresponding to the recursor, here's the type signature of the function `foldr` (fold *right*), which you can find in the standard library:

```
foldr :: (a -> c -> c) -> c -> [a] -> c
```

Here's a possible implementation:

```
foldr step init = \as ->
  case as of
    [] -> init
    a : as -> step a (foldr step init as)
```

As an example, we can use `foldr` to calculate the sum of the elements of a list of natural numbers:

```
sum :: [Nat] -> Nat
sum = foldr plus Z
```

Exercise 7.2.1. *Consider what happens when you replace A in the definition of a list with the terminal object. Hint: What is base-one encoding of natural numbers?*

Exercise 7.2.2. *How many mappings $h\colon L_A \to 1 + A$ are there? Can we get all of them using a list recursor? How about Haskell functions of the signature:*

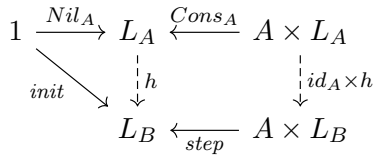
```
h :: [a] -> Maybe a
```

Exercise 7.2.3. *Implement a function that extracts the third element from a list, if the list is long enough. Hint: Use `Maybe a` for the result type.*

7.3 Functoriality

Functoriality means, roughly, the ability to transform the "contents" of a data structure. The contents of a list L_A is of the type A. Given an arrow $f\colon A \to B$, we need to define a mapping of lists $h\colon L_A \to L_B$.

Lists are defined by the mapping out property, so let's replace the target C of the elimination rule by L_B. We get:



Since we are dealing with two different lists here, we have to distinguish between their constructors. For instance, we have:

$$\mathrm{Nil}_A \colon 1 \to L_A$$

$$\mathrm{Nil}_B \colon 1 \to L_B$$

and similarly for $Cons$.

The only candidate for $init$ is Nil_B, which is to say that h acting on an empty list of As produces an empty list of Bs:

$$h \circ Nil_A = Nil_B$$

What remains is to define the arrow:

$$step \colon A \times L_B \to L_B$$

We can take:

$$step = \mathrm{Cons}_B \circ (f \times id_{L_B})$$

This corresponds to the Haskell function:

```
mapList :: (a -> b) -> List a -> List b
mapList f = recList init step
  where
    init = Nil
    step (a, bs) = Cons (f a, bs)
```

or, using the built-in list syntax and inlining the recursor,

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (a : as) = f a : map f as
```

You might wonder what prevents us from choosing $step = snd$, resulting in:

```
badMap :: (a -> b) -> [a] -> [b]
badMap f [] = []
badMap f (a : as) = badMap f as
```

We'll see, in the next chapter, why this is a bad choice. (Hint: What happens when we apply `badMap` to id?)

Chapter 8

Functors

8.1 Categories

So far we've only seen one category—that of types and functions. So let's quickly gather the essential info about a category.

A category is a collection of objects and arrows that go between them. Every pair of composable arrows can be composed. The composition is associative, and there is an identity arrow looping back on every object.

The fact that types and functions form a category can be expressed in Haskell by defining composition as:

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

The composition of two functions `g` after `f` is a new function that first applies `f` to its argument and then applies `g` to the result.

The identity is a polymorphic "do nothing" function:

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

You can easily convince yourself that such composition is associative, and composing with `id` does nothing to a function.

Based on the definition of a category, we can come up with all kinds of weird categories. For instance, there is a category that has no objects and no arrows. It satisfies all the condition of a category vacuously. There's another that contains a single object and a single arrow (can you guess what arrow it is?). There's one with two unconnected objects, and one where the two objects are connected by a single arrow (plus two identity arrows), and so on. These are example of what I call stick-figure categories.

Category of sets

We can also strip a category of all arrows (except for the identity arrows). Such a bare-object category is called a *discrete* category or a set[1]. Since we associate arrows with structure, a set is a category with no structure.

[1] Ignoring "size" issues.

Sets form their own category called **Set**[2]. The objects in that category are sets, and the arrows are functions between sets. Such functions are defined as special kind of relations, which themselves are defined as sets of pairs.

To lowest approximation, we can model programming in the category of sets. We often think of types as sets of values, and functions as set-theoretical functions. There's nothing wrong with that. In fact all of categorical construction we've described so far have their set-theoretical roots. The categorical product is a generalization of the cartesian product of sets, the sum is the disjoint union, and so on.

What category theory offers is more precision: the fine distinction between the structure that is absolutely necessary, and the superfluous details.

A set-theoretical function, for instance, doesn't fit the definition of a function we work with as programmers. Functions must have underlying algorithms because they have to be computable by some physical systems, be it computers or a human brains.

Opposite and product categories

In programming, the focus is on the category of types and functions, but we can use this category as a starting point to construct other categories.

One such category is called the *opposite* category. This is the category in which all the original arrows are inverted: what is called the source of an arrow on the original category is now called its target, and vice versa.

The opposite of a category $\mathcal{C}$ is called $\mathcal{C}^{op}$. We've had a glimpse of this category when we discussed duality. The terminal object in $\mathcal{C}$ is the initial object in $\mathcal{C}^{op}$, the product in $\mathcal{C}$ is the sum in $\mathcal{C}^{op}$, and so on.

Given two categories $\mathcal{C}$ and $\mathcal{D}$, we can construct a product category $\mathcal{C} \times \mathcal{D}$. The objects in this category are pairs of objects $\langle C, D \rangle$, and the arrows are pairs of arrows.

If we have an arrow $f \colon C \to C'$ in $\mathcal{C}$ and an arrow $g \colon D \to D'$ in $\mathcal{D}$ then there is a corresponding arrow $\langle f, g \rangle$ in $\mathcal{C} \times \mathcal{D}$. This arrow goes from $\langle C, D \rangle$ to $\langle C', D' \rangle$, both being objects in $\mathcal{C} \times \mathcal{D}$. Two such arrows can be composed if their components are composable in, respectively, $\mathcal{C}$ and $\mathcal{D}$. An identity arrow is a pair of identity arrows.

The two product categories we're most interested in are $\mathcal{C} \times \mathcal{C}$ and $\mathcal{C}^{op} \times \mathcal{C}$, where $\mathcal{C}$ is our familiar category of types and functions.

In both of these categories, objects are pairs of objects from $\mathcal{C}$. In the first category, $\mathcal{C} \times \mathcal{C}$, a morphism from $\langle A, B \rangle$ to $\langle A', B' \rangle$ is a pair $\langle f \colon A \to A', g \colon B \to B' \rangle$. In the second category, $\mathcal{C}^{op} \times \mathcal{C}$, a morphism is a pair $\langle f \colon A' \to A, g \colon B \to B' \rangle$, in which the first arrow goes in the opposite direction.

8.2 Functors

We've seen examples of functoriality when discussing algebraic data types. The idea is that such a data type "remembers" the way it was created, and we can manipulate this memory by applying an arrow to its "contents."

In some cases this intuition is very convincing: we think of a product type as a pair that "contains" its ingredients. After all, we can retrieve them using projections.

This is less obvious in the case of function objects. You can visualize a function object as secretly storing all possible results and using the function argument to index

[2]Again, ignoring "size" issues, in particular the non-existence of the set of all sets.

into them. A function of `Bool` is obviously equivalent to a pair of values, one for `True` and one for `False`. It's a known programming trick to implement some functions as lookup tables. It's called *memoization*.

Even though it's not practical to memoize functions that take, say, natural numbers as arguments; we can still conceptualize them as (infinite, or even uncountable) lookup tables.

If you can think of a data type as a container of values, it makes sense to apply a function to transform all these values and create a transformed container. When this is possible, we call the data type functorial.

Again, function types require some more suspension of disbelief. You visualize a function object as a lookup table, keyed by some type. If you want to use another type as your key, you need a function that translates the new key to the original key. This is why functoriality of the function object has one of the arrows reversed:

```
dimap :: (a' -> a) -> (b -> b') -> (a -> b) -> (a' -> b')
dimap f g h = g . h . f
```

You are applying the transformation to a function `h :: a -> b` that has a "receptor" that responds to values of type `a`, and you want to use it to process input of type `a'`. This is only possible if you have a converter from `a'` to `a`, namely `f :: a' -> a`.

The idea of a data type "containing" values of another type can be also expressed by saying that one data type is paremeterized by another. For instance, the type `List a` is parameterized by the type `a`.

In other words, `List` maps the type `a` to the type `List a`. `List` by itself, without the argument, is called a *type constructor*.

Functors between categories

In category theory, a type constructor is modeled as a mapping of objects to objects. It's a function on objects. This is not to be confused with arrows between objects, which are part of the structure of the category.

In fact, it's easier to imagine a mapping *between* categories. Every object in the source category is mapped to an object in the target category. If A is an object in $\mathcal{C}$, there is a corrsponding object FA in $\mathcal{D}$.

A functorial mapping, or a *functor*, not only maps objects but also arrows between them. Every arrow

$$f \colon A \to B$$

in the first category has a corresponding arrow in the second category:

$$Ff \colon FA \to FB$$

$$\begin{array}{ccc} A & \dashrightarrow & FA \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle Ff} \\ B & \dashrightarrow & FB \end{array}$$

We use the same letter, here F, to name both, the mapping of objects and the mapping of arrows.

If categories distill the essence of *structure*, then functors are mappings that preserve this structure. Objects that are related in the source category are related in the target category.

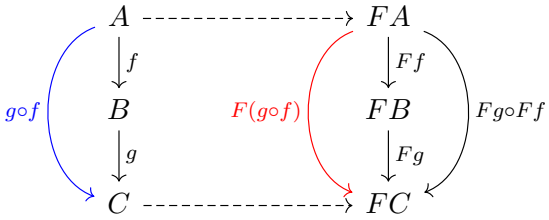
The structure of a category is defined by arrows and their composition. Therefore a functor must preserve composition. What is composed in one category:

$$h = g \circ f$$

should remain composed in the second category:

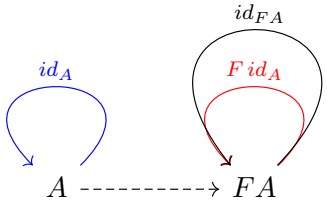
$$Fh = F(g \circ f) = Fg \circ Ff$$

We can either compose two arrows in $\mathcal{C}$ and map the composite to $\mathcal{D}$, or we can map individual arrows and then compose them in $\mathcal{D}$. We demand that the result be the same.



Finally, a functor must preserve identity arrows:

$$F\, id_A = id_{Fa}$$



These conditions taken together define what it means for a functor to preserve the structure of a category.

It's also important to realize what conditions are *not* part of the definition. For instance, a functor is allowed to map multiple objects into the same object. It can also map multiple arrows into the same arrow, as long as the endpoints match.

In the extreme, any category can be mapped to a singleton category with one object and one arrow.

Also, not all object or arrows in the target category must be covered by a functor. In the extreme, we can have a functor from the singleton category to any (non-empty) category. Such a functor picks a single object together with its identity arrow.

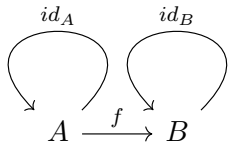
A *constant functor* Δ_C is an example of a functor that maps all objects from the source category to a single object C in the target category, and all arrows from the source category to a single identity arrow id_C.

In category theory, functors are often used for creating models of one category inside another. The fact that they can merge multiple objects and arrows into one means that they produce simplified views of the source category. They "abstract" some aspects of the source category.

The fact that they may only cover parts of the target category means that the models are embedded in a larger environment.

Functors from some minimalistic, stick-figure, categories can be used to define patterns in larger categories.

Exercise 8.2.1. *Describe a functor whose source is the "walking arrow" category. It's a stick-figure category with two objects and a single arrow between them (plus the mandatory identity arrows).*



Exercise 8.2.2. *The "walking iso" category is just like the "walking arrow" category, plus one more arrow going back from B to A. Show that a functor from this category always picks an isomorphism in the target category.*

8.3 Functors in Programming

Endofunctors are the class of functors that are the easiest to express in a programming language. These are functors that map a category (here, the category of types and functions) to itself.

Endofunctors

The first part of the endofunctor is the mapping of types to types. This is done using type constructors, which are type-level functions.

The list type constructor, `List`, maps an arbitrary type `a` to the type `List a`.

The `Maybe` type constructor maps `a` to `Maybe a`.

The second part of an endofunctor is the mapping of arrows. Given a function `a -> b`, we want to be able to define a function `List a -> List b`, or `Maybe a -> Maybe b`. This is the "functoriality" property of these data types that we have discussed before. Functoriality lets us *lift* an arbitrary function to a function between transformed types.

Functoriality can be expressed in Haskell using a *typeclass*. In this case, the typeclass is parameterized by a type constructor `f`. We say that `f` is a `Functor` if there is a corresponding mapping of functions called `fmap`:

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

The compiler knows that `f` is a type constructor because it's applied to types, as in `f a` and `f b`.

To prove to the compiler that a particular type constructor is a `Functor`, we have to provide the implementation of `fmap` for it. This is done by defining an *instance* of the typeclass `Functor`. For example:

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

A functor must also satisfy some laws: it must preserve composition and identity. These laws cannot be expressed in Haskell, but should be checked by the programmer. We have previously seen a definition of `badMap` that didn't satisfy the identity laws, yet it would be accepted by the compiler. It would define an "unlawful" instance of `Functor` for the list type constructor `[]`.

Exercise 8.3.1. *Show that `WithInt` is a functor*

```
data WithInt a = WithInt a Int
```

There are some elementary functors that might seem trivial, but they serve as building blocks for other functors.

We have the identity endofunctor that maps all objects to themselves, and all arrows to themselves.

```
data Id a = Id a
```

Exercise 8.3.2. *Show that `Id` is a `Functor`. Hint: implement the `Functor` instance for it.*

We also have a constant functor Δ_C that maps all objects to a single object C, and all arrows to the identity arrow on this object. In Haskell, it's a family of functors parameterized by the target object `c`:

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

This type constructor ignores its second argument.

Exercise 8.3.3. *Show that `(Const c)` is a `Functor`. Hint: The type constructor takes two arguments, but here it's partially applied to the first argument. It is functorial in the second argument.*

Bifunctors

We have also seen data constructors that take two types as arguments: the product and the sum. They were functorial as well, but instead of lifting a single function, they lift a pair of functions. In category theory, we would define these as functors from the product category $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$.

Such functors map a pair of objects to an object, and a pair of arrows to an arrow.

In Haskell, we treat such functors as members of a separate class called a `Bifunctor`.

```
class Bifunctor f where
  bimap :: (a -> a') -> (b -> b') -> (f a b -> f a' b')
```

Again, the compiler deduces that `f` is a two-argument type constructor because it sees it applied to two types, e.g., `f a b`.

To prove to the compiler that a particular type constructor is a `Bifunctor`, we define an instance. For example, bifunctoriality of a pair can be defined as:

```
instance Bifunctor (,) where
  bimap g h (a, b) = (g a, h b)
```

Exercise 8.3.4. *Show that `MoreThanA` is a bifunctor.*

```
data MoreThanA a b = More a (Maybe b)
```

Profunctors

We've seen that the function type is also functorial. It lifts two functions at a time, just like `Bifunctor`, except that one of the functions goes in the opposite direction.

In category theory this corresponds to a functor from a product of two categories, one of them being the opposite category: it's a functor from $\mathcal{C}^{op} \times \mathcal{C}$. Functors from $\mathcal{C}^{op} \times \mathcal{C}$ to to **Set** are called *profunctors*.

In Haskell, profunctors form a typeclass:

```
class Profunctor f where
  dimap :: (a' -> a) -> (b -> b') -> (f a b -> f a' b')
```

The function type, which can be written as an infix operator `(->)`, is an instance of `Profunctor`

```
instance Profunctor (->) where
  dimap f g h = g . h . f
```

In programming, all non-trivial profunctors are variations on the function type.

Contravariant functors

Functors from the opposite category $\mathcal{C}^{op}$ are called *contravariant*. They have the property of lifting arrows that go in the opposite direction. Regular functors are sometimes called *covariant*.

In Haskell, contravariant functors form the typeclass `Contravariant`:

```
class Contravariant f where
  contramap :: (b -> a) -> (f a -> f b)
```

A predicate is a function returning `True` or `False`:

```
data Predicate a = Predicate (a -> Bool)
```

It's easy to see that it's a contravariant functor:

```
instance Contravariant Predicate where
  contramap f (Predicate h) = Predicate (h . f)
```

The only non-trivial examples of contravariant functors are variations on the theme of function objects.

One way to recognize them is by assigning polarities to types that define a function type. We say that the return type is in a *positive* position, so it's covariant; and the argument type is in the *negative* position, so it's contravariant. But if you put the whole function object in the negative position of another function, then the polarities get reversed.

Consider this data type:

```
data Tester a = Tester ((a -> Bool) -> Bool)
```

It has `a` in a double-negative, therefore a positive position. This is why it's a covariant `Functor`:

```
instance Functor Tester where
  fmap f (Tester g) = Tester g'
    where g' h = g (h . f)
```

Notice that parentheses are important here. A similar function `a -> Bool -> Bool` has `a` in a *negative* position. That's because it's a function of `a` returning a function `(Bool -> Bool)`. Equivalently, you may uncurry it to get a function that takes a pair: `(a, Bool) -> Bool`.

8.4 The Hom Functor

Arrows between any two objects form a set. This set is called a hom-set and is usually written using the name of the category followed by the names of the objects:

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

We can interpret the hom-set $\mathcal{C}(A, B)$ as all the ways B can be observed from A.

Another way of looking at hom-sets is to say that they define a mapping that assigns a set $\mathcal{C}(A, B)$ to every pair of objects. Sets themselves are objects in the category **Set**. So we have a mapping between categories.

This mapping is functorial. To see that, let's consider what happens when we transform the two objects A and B. We are interested in a transformation that would map the set $\mathcal{C}(A, B)$ to the set $\mathcal{C}(A', B')$. Arrows in **Set** are regular functions, so it's enough to define their action on individual elements of a set.

An element of $\mathcal{C}(A, B)$ is an arrow $h \colon A \to B$ and an element of $\mathcal{C}(A', B')$ is an arrow $h' \colon A' \to B'$. We know how to transform one into another: we need to pre-compose h with an arrow $g' \colon A' \to A$ and post-compose it with an arrow $g \colon B \to B'$.

In other words, the mapping that takes a pair $\langle A, B \rangle$ to the set $\mathcal{C}(A, B)$ is a *profunctor*:

$$\mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$$

Frequently, we are interested in varying only one of the objects, keeping the other fixed. When we fix the source object and vary the target, the result is a functor that is written as:

$$\mathcal{C}(A, -) \colon \mathcal{C} \to \mathbf{Set}$$

The action of this functor on an arrow $g \colon B \to B'$ is written as:

$$\mathcal{C}(A, g) \colon \mathcal{C}(A, B) \to \mathcal{C}(A, B')$$

and is given by post-composition:

$$\mathcal{C}(A, g) = (g \circ -)$$

Varying B means switching focus from one object to another, so the complete functor $\mathcal{C}(A, -)$ combines all the arrows emanating from A into a coherent view of the category from the perspective of A. It is "the world according to A."

Conversely, when we fix the target and vary the source of the hom-profunctor, we get a contravariant functor:

$$\mathcal{C}(-, B) \colon \mathcal{C}^{op} \to \mathbf{Set}$$

whose action on an arrow $g' \colon A' \to A$ is written as:

$$\mathcal{C}(g', B) \colon \mathcal{C}(A, B) \to \mathcal{C}(A', B)$$

and is given by pre-composition:

$$\mathcal{C}(g', B) = (- \circ g')$$

The functor $\mathcal{C}(-, B)$ organizes all the arrows pointing at B into one coherent view. It is the picture of B "as seen by the world."

We can now reformulate the results from the chapter on isomorphisms. If two objects A and B are isomorphic, than their hom-sets are also isomorphic. In particular:

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

and

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

We'll discuss naturality conditions in the next chapter.

8.5 Functor Composition

Just like we can compose functions, we can compose functors. Two functors are composable if the target category of one is the source category of the other.

On objects, functor composition of G after F first applies F to an object, then applies G to the result; and similarly on arrows.

Obviously, you can only compose composable functors. However all *endofunctors* are composable, since their target category is the same as the source category.

In Haskell, a functor is a parameterized data type, so the composition of two functors is again a parameterized data type. On objects, we define:

```
data Compose g f a = Compose (g (f a))
```

The compiler figures out that `f` and `g` must be type constructors because they are applied to types: `f` is applied to the type parameter `a`, and `g` is applied to the resulting type.

Alternatively, you can tell the compiler that the first two arguments to `Compose` are type constructors. You do this by providing a *kind signature*, which requires a language extension `KindSignatures` that you put at the top of the source file:

```
{-# language KindSignatures #-}
```

You should also import the `Data.Kind` library that defines `Type`:

```
import Data.Kind
```

A kind signature is just like a type signature, except that it can be used to describe functions operating on types.

Regular types have the kind `Type`. Type constructors have the kind `Type -> Type`, since they map types to types.

`Compose` takes two type constructors and produces a type constructor, so its kind signature is:

```
(Type -> Type) -> (Type -> Type) -> (Type -> Type)
```

and the full definition is:

```
data Compose :: (Type -> Type) -> (Type -> Type) -> (Type -> Type)
  where
    Compose :: (g (f a)) -> Compose g f a
```

Any two type constructors can be composed this way. There is no requirement, at this point, that they be functors.

However, if we want to lift a function using the composition of type constructors, `g` after `f`, then they must be functors. This requirement is encoded as a constraint in the instance declaration:

```
instance (Functor g, Functor f) => Functor (Compose g f) where
  fmap h (Compose gfa) = Compose (fmap (fmap h) gfa)
```

The constraint `(Functor g, Functor f)` expresses the condition that both type constructors be instances of the `Functor` class. The constraints are followed by a double arrow.

The type constructor whose functoriality we are establishing is `Compose f g`, which is a partial application of `Compose` to two functors.

In the implementation of `fmap`, we pattern match on the data constructor `Compose`. Its argument `gfa` is of the type `g (f a)`. We use one `fmap` to "get under" `g`. Then we use `(fmap h)` to get under `f`. The compiler knows which `fmap` to use by analyzing the types.

You may visualize a composite functor as a container of containers. For instance, the composition of `[]` with `Maybe` is a list of optional values.

Exercise 8.5.1. *Define a composition of a `Functor` after `Contravariant`. Hint: You can reuse `Compose`, but you have to provide a different instance declaration.*

Category of categories

We can view functors as arrows between categories. As we've just seen, functors are composable and it's easy to check that this composition is associative. We also have an identity (endo-) functor for every category. So categories themselves seem to form a category, let's call it **Cat**.

And this is where mathematicians start worrying about "size" issues. It's a shorthand for saying that there are paradoxes lurking around. So the correct incantation is that **Cat** is a category of *small* categories. But as long as we are not engaged in proofs of existence, we can ignore size problems.

Chapter 9

Natural Transformations

We've seen that, when two objects A and B are isomorphic, they generate bijections between sets of arrows, which we can now express as isomorphisms between hom-sets:

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

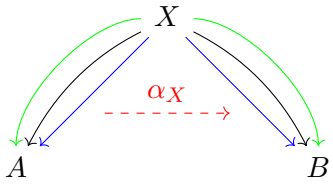
$$\mathcal{C}(X, A) \cong \mathcal{C}(X, B)$$

The converse is not true, though. An isomorphism between hom-sets does not result in an isomorphism between object *unless* additional naturality conditions are satisfied. We'll now re-formulate these naturality conditions in progressively more general settings.

9.1 Natural Transformations Between Hom-Functors

One way an isomorphism between two objects can be established is by directly providing two arrows—one the inverse of the other. But quite often it's easier to do it indirectly, by defining bijections between arrows, either the ones impinging on the two objects, or the ones emanating from the two objects.

For instance, as we've seen before, we may have, for every X, an invertible mapping of arrows α_X.



In other words, for every X, there is a mapping of hom-sets:

$$\alpha_X : \mathcal{C}(X, A) \to \mathcal{C}(X, B)$$

When we vary X, the two hom-sets become two (contravariant) functors, and α can be seen as a mapping between two functors: $\mathcal{C}(-, A)$ and $\mathcal{C}(-, B)$. Such a mapping, or a transformation, is really a family of individual mappings α_X, one per each object X in the category $\mathcal{C}$.

The functor $\mathcal{C}(-, A)$ describes the way the worlds sees A, and the functor $\mathcal{C}(-, B)$ describes the way the world sees B.

The transformation α switches back and forth between these two views. Every component of α, the bijection α_X, shows that the view of A from X is isomorphic to the view of B from X.

The naturality condition we discussed before was the condition:

$$\alpha_Y \circ (- \circ g) = (- \circ g) \circ \alpha_X$$

It relates components of α taken at different objects. In other words, it relates the views from two different observers X and Y, who are connected by the arrow $g\colon Y \to X$.

Both sides of this equation are acting on the hom-set $\mathcal{C}(X, A)$. The result is in the hom-set $\mathcal{C}(Y, B)$

Precomposition with $g\colon Y \to X$ is also a mapping of hom-sets. In fact it is the lifting of g by the contravariant hom-functor. We can write it as $\mathcal{C}(g, A)$ and $\mathcal{C}(g, B)$, respectively.

The naturality condition can therefore be rewritten as:

$$\alpha_Y \circ \mathcal{C}(g, A) = \mathcal{C}(g, B) \circ \alpha_X$$

It can be illustrated by this commuting diagram:

$$\begin{array}{ccc} \mathcal{C}(X, A) & \xrightarrow{\mathcal{C}(g,A)} & \mathcal{C}(Y, A) \\ \downarrow{\scriptstyle \alpha_X} & & \downarrow{\scriptstyle \alpha_Y} \\ \mathcal{C}(X, B) & \xrightarrow{\mathcal{C}(g,B)} & \mathcal{C}(Y, B) \end{array}$$

We can now say that an invertible transformation α between the functors $\mathcal{C}(-, A)$ and $\mathcal{C}(-, B)$ that satisfies the naturality condition is equivalent to an isomorphism between A and B.

We can follow exactly the same reasoning for the outgoing arrows. This time we start with a transformation β whose components are:

$$\beta_X\colon \mathcal{C}(A, X) \to \mathcal{C}(B, X)$$

The two (covariant) functors $\mathcal{C}(A, -)$ and $\mathcal{C}(B, -)$ describe the view of the world from the perspective of A and B, respectively. The invertible transformation β tells us that these two views are equivalent, and the naturality condition

$$(g \circ -) \circ \beta_X = \beta_Y \circ (g \circ -)$$

tells us that they behave nicely when we switch focus.

Here's the commuting diagram that illustrates the naturality condition:

$$\begin{array}{ccc} \mathcal{C}(A, X) & \xrightarrow{\mathcal{C}(A,g)} & \mathcal{C}(A, Y) \\ \downarrow{\scriptstyle \beta_X} & & \downarrow{\scriptstyle \beta_Y} \\ \mathcal{C}(B, X) & \xrightarrow{\mathcal{C}(B,g)} & \mathcal{C}(B, Y) \end{array}$$

Again, such an invertible natural transformation β establishes the isomorphism between A and B.

9.2 Natural Transformation Between Functors

The two hom-functors from the previous section were

$$FX = \mathcal{C}(A, X)$$

$$GX = \mathcal{C}(B, X)$$

They both map the category $\mathcal{C}$ to **Set**, because that's where the hom-sets live. We can say that they create two different *models* of $\mathcal{C}$ inside **Set**.

A natural transformation is a structure-preserving mapping between such models.

This idea naturally extends to functors between any pair of categories. Any two functors

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

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

may be seen as two different models of $\mathcal{C}$ inside $\mathcal{D}$.

To transform one model into another we connect the corresponding dots using arrows in $\mathcal{D}$.

For every object X in $\mathcal{C}$ we pick an arrow that goes from FX to GX:

$$\alpha_X \colon FX \to GX$$

A natural transformation thus maps objects to arrows.

The structure of the models, though, has as much to do with objects as it does with arrows, so let's see what happens to arrows. For every arrow $f \colon X \to Y$ in $\mathcal{C}$, there are two corresponding arrows in $\mathcal{D}$:

$$Ff \colon FX \to FY$$

$$Gf \colon GX \to GY$$

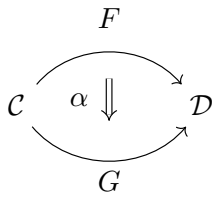
These are the two liftings of f. You can use them to move within the bounds of each of the two models. Then there are the components of α which let you switch between models.

Naturality says that it shouldn't matter whether you first move inside the first model and then jump to the second one, or first jump to the second one and then move within it. This is illustrated by the commuting *naturality square*:

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ \downarrow{\scriptstyle \alpha_X} & & \downarrow{\scriptstyle \alpha_Y} \\ GX & \xrightarrow{Gf} & GY \end{array}$$

A family of arrows α_X that satisfies the naturality condition is called a *natural transformation*.

This is a diagram that shows a pair of categories, two functors between them, and a natural transformation α between the functors:



Since for every arrow in $\mathcal{C}$ there is a corresponding naturality square, we can say that a natural transformation maps objects to arrows, and arrows to commuting squares.

If every component α_X of a natural transformation is an isomorphism, α is called a *natural isomorphism*.

We can now restate the main result about isomorphisms: Two objects are isomorphic if and only if there is a natural isomorphism between their hom-functors (either the covariant, or the contravariant ones—either one will do).

Natural transformations provide a very convenient high-level way of expressing commuting conditions in a variety of situations. We'll use them in this capacity to reformulate the definitions of algebraic data types.

9.3 Natural Transformations in Programming

A natural transformation is a family of arrows parameterized by objects. In programming, this corresponds to a family of functions parameterized by types, that is a *polymorphic function*.

The type of the argument to a natural transformation is constructed using one functor, and the return type using another.

In Haskell, we can define a data type that accepts two type constructors representing two functors, and produces a type of natural transformations:

```
data Natural :: (Type -> Type) -> (Type -> Type) -> Type where
  Natural :: (forall a. f a -> g a) -> Natural f g
```

The `forall` quantifier tells the compiler that the function is polymorphic—that is, it's defined for every type `a`. As long as `f` and `g` are functors, this formula defines a natural transformation.

The types defined by `forall` are very special, though. The are polymorphic in the sense of *parametric polymorphism*. It means that a single formula is used for all types. We've seen the example of the identity function, which can be written as:

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

The body of this function is very simple, just the variable `x`. It doesn't matter what type `x` is, the formula remains the same.

This is in contrast to *ad-hoc polymorphism*. An ad-hoc polymorphic function may use different implementations for different types. An example of such a function is `fmap`, the member function of the `Functor` typeclass. There is one implementation of `fmap` for lists, a different one for `Maybe`, and so on, case by case.

It turns out that limiting the type of a natural transformation to adhere to parametric polymorphism has far reaching consequences. Such a function automatically satisfies the naturality condition. It's an example of parametricity producing so called *theorems for free*.

The standard definition of a (parametric) natural transformation in Haskell uses a *type synonym:*

```
type Natural f g = forall a. f a -> g a
```

A `type` declaration introduces an alias, a shorthand, for the right-hand-side.

Here's an example of a useful function that is a natural transformation between the list functor and the `Maybe` functor:

```
safeHead :: Natural [] Maybe
safeHead [] = Nothing
safeHead (a : as) = Just a
```

(The standard library `head` function is "unsafe" in that it faults when given an empty list.)

Another example is the function `reverse`, which reverses a list. It's a natural transformation from the list functor to the list functor:

```
reverse :: Natural [] []
reverse [] = []
reverse (a : as) = reverse as ++ [a]
```

Incidentally, this is a very inefficient implementation. The actual library function uses an optimized algorithm.

A useful intuition for understanding natural transformations builds on the idea that functors acts like containers of data. There are two completely orthogonal things that you can do with a container: You can transform the data it contains, without changing the shape of the container. This is what `fmap` does. Or you can transfer the data, without modifying it, to another container. This is what a natural transformation does: It's a procedure of moving "stuff" between containers without knowing what kind of "stuff" it is.

Naturality condition enforces the orthogonality of these two operations. It doesn't matter if you first modify the data and then move it to another container; or first move it, and then modify.

This is another example of successfully decomposing a complex problem into a sequence of simpler ones. Keep in mind, though, that not every operation with containers of data can be decomposed in that way. Filtering, for instance, requires both examining the data, as well as changing the size or even the shape of the container.

On the other hand, almost every parametrically polymorphic function is a natural transformation. In some cases you may have to consider the identity or the constant functor as either source or the target. For instance, the polymorphic identity function can be though of as a natural transformation between two identity functors.

9.4 The Functor Category

Objects and arrows are drawn differently. Objects are dots and arrows are pointy lines.

In **Cat**, the category of categories, functors are drawn as arrows. But we have natural transformations that go between functors, so it looks like functors could be objects as well.

What is an arrow in one category could be an object in another.

Vertical composition of natural transformations

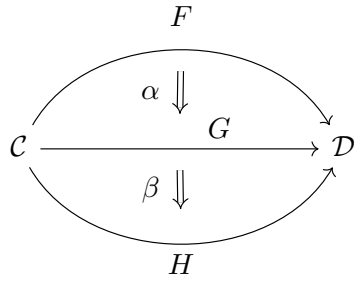
Natural transformations can only be defined between *parallel* functors, that is functors that share the same source category and the same target category. Such parallel functors form a *functor category*. The standard notation for a functor category between two

categories $\mathcal{C}$ and $\mathcal{D}$ is $[\mathcal{C}, \mathcal{D}]$, that is the names of the two categories between square brackets.

The objects in $[\mathcal{C}, \mathcal{D}]$ are functors, the arrows are natural transformations.

To show that this is indeed a category, we have to define the composition of natural transformations. This is easy if we keep in mind that components of natural transformations are regular arrows in the target category. These arrows compose.

Indeed, suppose that we have a natural transformation α between two functors F and G. We want to compose it with another natural transformation β that goes from G to H.



Let's look at the components of these transformations at some object X

$$\alpha_X \colon F\,X \to G\,X$$

$$\beta_X \colon G\,X \to H\,X$$

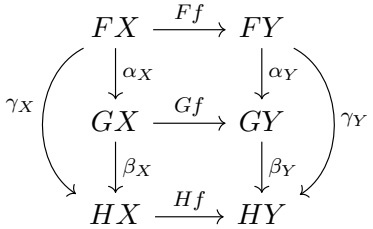
These are just two arrows in $\mathcal{D}$ that are composable. So we can define a composite natural transformation γ as follows:

$$\gamma \colon F \to H$$

$$\gamma_X = \beta_X \circ \alpha_X$$

This is called the *vertical composition* of natural transformations. You'll see it written using a dot $\gamma = \beta \cdot \alpha$ or a simple juxtaposition $\gamma = \beta\alpha$.

Naturality condition for γ can be shown by pasting together (vertically) two naturality squares for α and β:



Since the composition of natural transformations is defined in terms of composition of arrows, it is automatically associative.

There is also an identity natural transformation id_F defined for every functor F. Its component at X is the usual identity arrow at the object FX:

$$(id_F)_X = id_{FX}$$

To summarize, for every pair of categories $\mathcal{C}$ and $\mathcal{D}$ there is a category of functors $[\mathcal{C}, \mathcal{D}]$ with natural transformations as arrows.

The hom-set in that category is the set of natural transformations between two functors F and G. Following the standard notational convention, we write it as:

$$[\mathcal{C}, \mathcal{D}](F, G)$$

with the name of the category followed by the names of the two objects (here, functors) in parentheses.

Exercise 9.4.1. *Prove the naturality condition of the composition of natural transformations:*

$$\gamma_Y \circ Ff = Hf \circ \gamma_X$$

Hint: Use the definition of γ and the two naturality conditions for α and β.

Horizontal composition of natural transformations

The second kind of composition of natural transformations is induced by composition of functors. Suppose that we have a pair of composable functors

$$F \colon \mathcal{C} \to \mathcal{D} \qquad\qquad G \colon \mathcal{D} \to \mathcal{E}$$

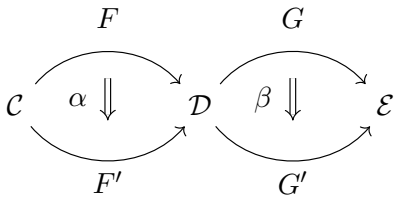
and that this pair is parallel to another pair of composable functors:

$$F' \colon \mathcal{C} \to \mathcal{D} \qquad\qquad G' \colon \mathcal{D} \to \mathcal{E}$$

We also have two natural transformations:

$$\alpha \colon F \to F' \qquad\qquad \beta \colon G \to G'$$

Pictorially:



The *horizontal composition* $\beta \circ \alpha$ maps $G \circ F$ to $G' \circ F'$.

Let's pick an object X in $\mathcal{C}$. We use α to map it to an arrow

$$\alpha_X \colon FX \to F'X$$

We can lift this arrow using G

$$G(\alpha_X) \colon G(FX) \to G(F'X)$$

What we need to define $\beta \circ \alpha$ is an arrow from $G(FX)$ to $G'(F'X)$. To get there, we can use the appropriate component of β

$$\beta_{F'X} \colon G(F'X) \to G'(F'X)$$

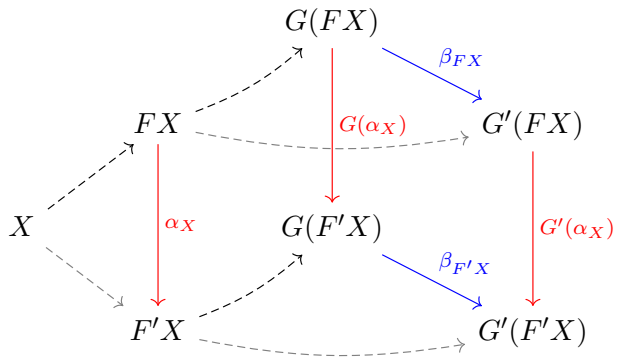
Altogether, we have

$$(\beta \circ \alpha)_X = \beta_{F'X} \circ G(\alpha_X)$$

But there is another equally plausible candidate:

$$(\beta \circ \alpha)_X = G'(\alpha_X) \circ \beta_{FX}$$

Fortunately, the are equal due to naturality of β.

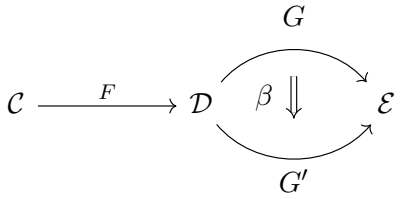


The proof of naturality of $\beta \circ \alpha$ is left as an exercise to a dedicated reader.

Whiskering

Quite often, horizontal composition is used with one of the natural transformations being the identity. There is a shorthand notation for such composition. For instance, $\beta \circ id_F$ is written as $\beta \circ F$.

Because of the characteristic shape of the diagram, such composition is called "whiskering".



In components, we have:

$$(\beta \circ F)_X = \beta_{FX}$$

Let's consider how we would translate this to Haskell. A natural transformation is a polymorphic function. Because of parametricity, it's defined by the same formula for all types. So whiskering on the right doesn't change the formula, it changes function signature.

For instance, if this is the declaration of `beta`:

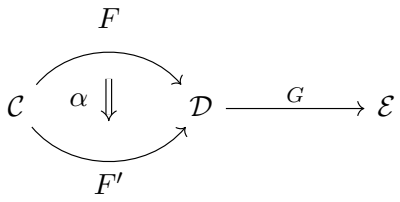
```
beta :: G x -> G' x
```

then its whiskered version would be:

```
beta_f :: G (F x) -> G' (F x)
```

Because of Haskell's type inference, this shift is often implicit.

Similarly, $id_G \circ \alpha$ is written as $G \circ \alpha$.



In components:

$$(G \circ \alpha)_X = G(\alpha_X)$$

In Haskell, the lifting of α_X by G is done using `fmap`, so given:

```
alpha :: F x -> F' x
```

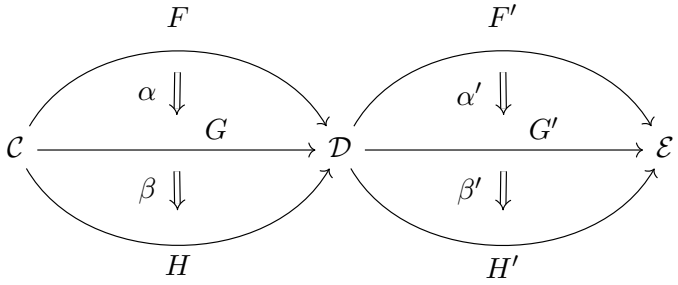
the whiskered version would be:

```
g_alpha :: G (F x) -> G (F' x)
g_alpha = fmap alpha
```

Again, Haskell's type inference engine figures out which version of `fmap` to use (here, it's the one from the `Functor` instance of `G`).

Interchange law

We can combine vertical composition with horizontal composition, as seen in the following diagram:



The interchange law states that the order of composition doesn't matter: we can first do vertical compositions and then the horizontal one, or first do the horizontal compositions and then the vertical one.

9.5 Universal Constructions Revisited

We've seen definitions of sums, products, exponentials, natural numbers, and lists.

The old-school approach to defining such data types is to explore their internals. This is the set-theory way: we look at how the elements of new sets are constructed from the elements of old sets. An element of a sum is either an element of the first set, or the second set. An element of a product is a pair of elements. And so on. We are looking at objects from the engineering point of view.

In category theory we take the opposite approach. We are not interested in what's inside the object or how it's implemented. We are interested in the purpose of the

object, how it can be used, and how it interacts with other objects. We are looking at objects from the user's point of view.

Both approaches have their advantages. The categorical approach came later, because you need to study a lot of examples before clear patterns emerge. But once you see the patterns, you discover unexpected connections between things, like the duality between sums and products.

Defining particular objects through their connections requires looking at possibly infinite numbers of objects with which they interact.

"Tell me your place in the Universe, and I'll tell you who you are."

Defining an object by its mappings-out or mappings-in with respect to all objects in the category is called a *universal construction*.

Why are natural transformations so important? It's because most categorical constructions involve commuting diagrams. If we can re-cast these diagrams as naturality squares, we move one level up the abstraction ladder and gain new valuable insights.

Being able to compress a lot of facts into small elegant formulas helps us see new patterns. We'll see, for instance, that natural isomorphisms between hom-sets pop up all over category theory and eventually lead to the idea of an adjunction.

But first we'll study several examples in greater detail to get some understanding of the terse language of category theory. We'll try, for instance, to decode the statement that the sum, or the coproduct of two objects, is defined by the following natural isomorphism:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X] \cong \mathcal{C}(A + B, X)$$

Picking objects

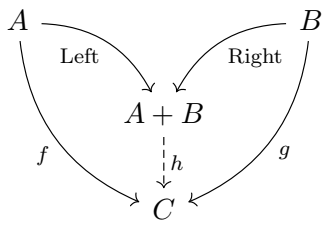
Even such a simple task as pointing at objects has a special interpretation in category theory. We have already seen that pointing at an element of a set is equivalent to selecting a function from the singleton set to it. Similarly, picking an object in a category can be replaced by selecting a functor from the single-object category. Or it can be done using a constant functor from any category.

Quite often we want to pick a pair of objects. That, too, can be accomplished by selecting a functor from a two-object stick-figure category. Similarly, picking an arrow is equivalent to selecting a functor from the "walking arrow" category, and so on.

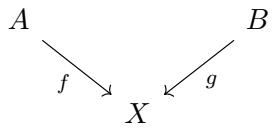
By judiciously selecting our functors and natural transformations between them, we can reformulate all the universal constructions we've seen so far.

Cospans as natural transformations

The definition of a sum requires the selection of two objects to be summed; and a third one to serve as the target of the mapping out.



This diagram can be further decomposed into two simpler shapes called *cospans*:



To construct a cospan we first have to pick a pair of objects. To do that we'll start with a two-object category **2**. We'll call its objects 1 and 2. We'll use a functor

$$D \colon \mathbf{2} \to \mathcal{C}$$

to select the objects A and B:

$$D\,1 = A$$

$$D\,2 = B$$

(D stands for "diagram", since the two objects form a very simple diagram consisting of two dots in $\mathcal{C}$.)

We'll use the constant functor

$$\Delta_X \colon \mathbf{2} \to \mathcal{C}$$

to select the object X. This functor maps both 1 and 2 to X (and the two identity arrows to id_X).

Since both functors go from **2** to $\mathcal{C}$, we can define a natural transformation α between them. In this case, it's just a pair of arrows:

$$\alpha_1 \colon D\,1 \to \Delta_X 1$$

$$\alpha_2 \colon D\,2 \to \Delta_X 2$$

These are exactly the two arrows f and g in the cospan.

Naturality condition for α is trivial, since there are no arrows (other than identities) in **2**.

There may be many cospans sharing the same three objects—meaning: there may be many natural transformations between the two functors D and Δ_X. These natural transformations form a hom-set in the functor category $[\mathbf{2}, \mathcal{C}]$, namely:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X)$$

Functoriality of cospans

Let's consider what happens when we start varying the object X in a cospan. We get a mapping from X to the set of cospans F:

$$FX = [\mathbf{2}, \mathcal{C}](D, \Delta_X)$$

This mapping turns out to be functorial in X.

To see that, consider an arrow $m \colon X \to Y$. The lifting of this arrow is a mapping between two sets of natural transformations:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X) \to [\mathbf{2}, \mathcal{C}](D, \Delta_Y)$$

This might look very abstract until you remember that natural transformations have components, and these components are just regular arrows. An element of the left-hand side is a natural transformation:

$$\mu \colon D \to \Delta_X$$

It has two components corresponding to the two objects in **2**. For instance, we have

$$\mu_1 \colon D\,1 \to \Delta_X 1$$

or, using the definitions of D and Δ:

$$\mu_1 \colon A \to X$$

This is just the arrow f in our diagram.

Similarly, the element of the right-hand side is:

$$\nu \colon D \to \Delta_Y$$

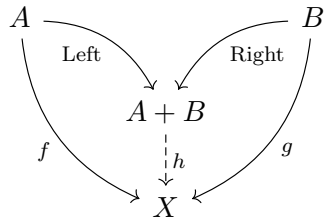
Its component at 1 is an arrow

$$\nu_1 \colon A \to Y$$

We can get from μ_1 to ν_1 simply by post-composing it with $m \colon X \to Y$. So the lifting of h is a component-by-component post-compositon $(m \circ -)$.

Sum as a universal cospan

Of all the cospans that you can build on the pair A and B, the one with the arrows we called $Left$ and $Right$ converging on $A + B$ is very special. There is a unique mapping out of it to any other cospan—a mapping that makes two triangles commute.



We are now in a position to translate this condition into a statement about natural transformations and hom-sets. The arrow h is an element of the hom-set

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

A cospan centered at X is a natural transformation, that is an element of the hom-set in the functor category:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X)$$

Both are hom-sets in their respective categories. But they are just sets, that is objects in the category **Set**. This category forms a bridge between the functor category $[\mathbf{2}, \mathcal{C}]$ and a "regular" category $\mathcal{C}$, even though, conceptually, they seem to be at very different levels of abstraction.

As Lao Tzu would say, "Sometimes a set is just a set."

Our universal construction is the bijection or the isomorphism of sets:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X) \cong \mathcal{C}(A + B, X)$$

Moreover, if we vary the object X, the two sides behave like functors from $\mathcal{C}$ to **Set**. Therefore it makes sense to ask if this mapping of functors is a natural isomorphism.

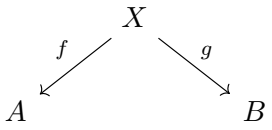
Indeed, it can be shown that the naturality condition for this isomorphism translates into commuting conditions for the triangles in the definition of the sum. So the definition of the sum can be replaced by a single equation.

Product as a universal span

An analogous argument can be made about the universal construction of the product. Again, we start with the stick-figure category **2** and the functor D. But this time we use a natural transformation going in the opposite direction

$$\alpha \colon \Delta_X \to D$$

Such a natural transformation is a pair of arrows that form a *span*:



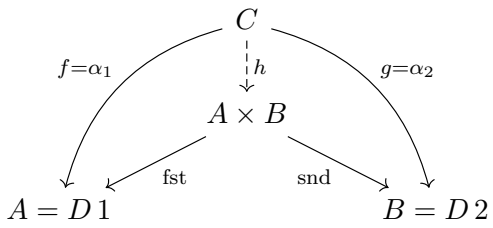
Collectively, these natural transformations form a hom-set in the functor category :

$$[\mathbf{2}, \mathcal{C}](\Delta_X, D)$$

Every element of this hom-set is in one-to-one correspondence with a unique mapping h into the product $A \times B$. Such a mapping is a member of the hom-set $\mathcal{C}(X, A \times B)$. This correspondence is expressed as the isomorphism:

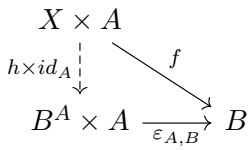
$$[\mathbf{2}, \mathcal{C}](\Delta_X, D) \cong \mathcal{C}(X, A \times B)$$

It can be shown that the naturality of this isomorphism guarantees that the triangles in this diagram commute:



Exponentials

The exponentials, or function objects, are defined by this commuting diagram:



Here, f is an element of the hom-set $\mathcal{C}(X \times A, B)$ and h is an element of $\mathcal{C}(X, B^A)$.

The isomorphism between these sets, natural in X, defines the exponential object.

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

The f in the diagram above is an element of the left-hand side, and h is the corresponding element of the right-hand side. The transformation α maps f to h. In Haskell, we call it `curry`. Its inverse, α^{-1} is known as `uncurry`.

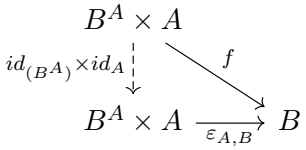
Unlike in the previous examples, here both hom-sets are in the same category, and it's easy to analyze the isomorphism in more detail. In particular, we'd like to see how the commuting condition:

$$f = \varepsilon_{A,B} \circ (h \times id_A)$$

arises from naturality.

The standard Yoneda trick is to make a substitution for X that would reduce one of the hom-sets to an endo-hom-set, that is a hom-set whose source is the same the target. This will allow us to pick a special element of that hom-set, namely the identity arrow.

In our case, substituting B^A for X will allow us to pick $h = id_{(B^A)}$.



The commuting condition in this case tells us that $f = \varepsilon_{A,B}$. In other words, we get the formula for $\varepsilon_{A,B}$ in terms of α:

$$\varepsilon_{A,B} = \alpha^{-1}(id_{(B^A)})$$

Since we recognize α^{-1} as `uncurry`, and ε as function application, we can write it in Haskell as:

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

This my be surprising at first, until you realize that the currying of `(a->b,a)->b` leads to `(a->b)->(a->b)`.

We can also encode the two sides of the main isomorphism as Haskell functors:

```
data LeftFunctor  a b x = LF ((x, a) -> b)
```

```
data RightFunctor a b x = RF (x -> (a -> b))
```

They are both contravariant functors in the type variable `x`.

```
instance Contravariant (LeftFunctor a b) where
  contramap g (LF f) = LF (f . bimap g id)
```

This says that the lifting of $g \colon X \to Y$ acting on $\mathcal{C}(Y \times A, B)$ is given by

$$(- \circ (g \times id_A)) \colon \mathcal{C}(Y \times A, B) \to \mathcal{C}(X \times A, B)$$

Similarly

```
instance Contravariant (RightFunctor a b) where
  contramap g (RF h) = RF (h . g)
```

translates to

$$(- \circ g) \colon \mathcal{C}(Y, B^A) \to \mathcal{C}(X, B^A)$$

The natural transformation α is just a thin encapsulation of `curry`; and its inverse is `uncurry`:

```
alpha :: forall a b x. LeftFunctor a b x -> RightFunctor a b x
alpha (LF f) = RF (curry f)
```

```
alpha_1 :: forall a b x. RightFunctor a b x -> LeftFunctor a b x
alpha_1 (RF h) = LF (uncurry h)
```

Using the two formulas for the lifting of $g \colon X \to Y$, here's the naturality square:

$$\begin{array}{ccc} \mathcal{C}(Y \times A, B) & \xrightarrow{(-\circ(g \times id_A))} & \mathcal{C}(X \times A, B) \\ \downarrow{\scriptstyle \alpha_Y} & & \downarrow{\scriptstyle \alpha_X} \\ \mathcal{C}(Y, B^A) & \xrightarrow{(-\circ g)} & \mathcal{C}(X, B^A) \end{array}$$

Let's now apply the Yoneda trick to it and replace Y with B^A. This also allows us to substitute g, which now goes for X to B^A, with h.

$$\begin{array}{ccc} \mathcal{C}(B^A \times A, B) & \xrightarrow{(-\circ(h \times id_A))} & \mathcal{C}(X \times A, B) \\ \downarrow{\scriptstyle \alpha_{(B^A)}} & & \downarrow{\scriptstyle \alpha_X} \\ \mathcal{C}(B^A, B^A) & \xrightarrow{(-\circ h)} & \mathcal{C}(X, B^A) \end{array}$$

We know that the hom-set $\mathcal{C}(B^A, B^A)$ contains at least the identity arrow, so we can pick the element $id_{(B^A)}$ in the lower left corner.

α^{-1} acting on it produces $\varepsilon_{A,B}$ in the upper left corner (that's the `uncurry id` trick).

Pre-composition with h acting on identity produces h in the lower right corner.

α^{-1} acting on h produces f in the upper right corner.

$$\begin{array}{ccc} \varepsilon_{A,B} & \xmapsto{(-\circ(h \times id_A))} & f \\ {\scriptstyle \alpha^{-1}}\uparrow & & \uparrow{\scriptstyle \alpha^{-1}} \\ id_{(B^A)} & \xmapsto{(-\circ h)} & h \end{array}$$

(The $\mapsto$ arrows denote the action of functions on elements of sets.)

So the selection of $id_{(B^A)}$ in the lower left corner fixes the other three corners. In particular, we can see that the upper arrow applied to $\varepsilon_{A,B}$ produces f, which is exactly the commuting condition:

$$\varepsilon_{A,B} \circ (h \times id_A) = f$$

the one that we set out to derive.

9.6 Limits and Colimits

In the previous section we defined the sum and the product using natural transformations. These were transformations between diagrams defined as functors from a very simple stick-figure category **2**, one of them being the constant functor.

Nothing prevents us from replacing the category **2** with something more complex. For instance, we could try categories that have non-trivial arrows between objects, or categories with infinitely many objects.

There is a whole vocabulary built around such constructions.

We used objects in the category **2** for indexing objects in the category $\mathcal{C}$. We can replace **2** with an arbitrary indexing category **I**. A diagram in $\mathcal{C}$ is still defined as a functor $D\colon \mathbf{I} \to \mathcal{C}$. It picks objects in $\mathcal{C}$ as well as some of the arrows between them.

As the second functor we'll still use the constant functor $\Delta_X\colon \mathbf{I} \to \mathcal{C}$.

A natural transformation that's an element of the hom-set

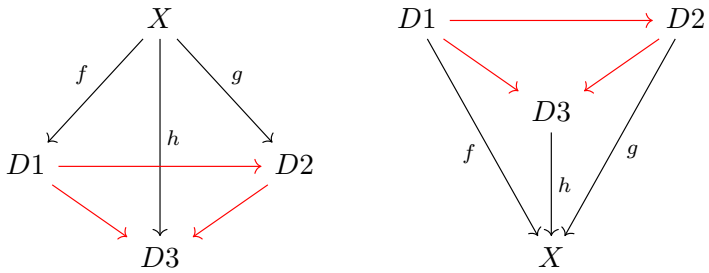
$$[\mathbf{I}, \mathcal{C}](\Delta_X, D)$$

is now called a *cone*. Its dual, an element of

$$[\mathbf{I}, \mathcal{C}](D, \Delta_X)$$

is called a *cocone*. They generalize the span and the cospan, respectively.

Diagramatically, cones and cocones look like this:



Since the indexing category may now contain arrows, the naturality conditions for these diagrams are no longer trivial. The constant functor Δ_X shrinks all vertices to one, so naturality squares turn into triangles. Naturality means that all triangles with X in their apex must now commute.

The universal cone, if it exists, is called the *limit* of the diagram D, and is written as Lim_D. Universality means that it satisfies the following isomorphism, natural in X:

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

Dually, the universal cocone is called a *colimit*, and is described by the following natural isomorphism:

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

We can now say that a product is a limit, and a sum is a colimit, of a diagram from the indexing category **2**.

Limits and colimits distill the essence of a pattern.

A limit, like a product, it is defined by its mapping-in property.

A colimit, like a sum, it is defined by its mapping out property.

There are many interesting limits and colimits, and we'll see some when we discuss algebras and coalgebras.

Exercise 9.6.1. *Show that the limit of a "walking arrow" category, that is a two-object category with an arrow connecting the two objects, has the same elements as the first object in the diagram ("elements" are the arrows from the terminal object).*

9.7 The Yoneda Lemma

A functor from some category $\mathcal{C}$ to the category of sets can be thought of as a model of this category in **Set**. Modeling, in general, is a lossy process: it discards some information. A constant functor is an extreme example: it maps the whole category to a single set and its identity function.

A hom-functor produces a model of the category as viewed from a certain vantage point. The functor $\mathcal{C}(A, -)$, for instance, offers the panorama of $\mathcal{C}$ from the vantage point of A. It organizes all the arrows emanating from A into neat packages that are connected by images of arrows that go between them, all in accordance with the original structure of the source category.

Some vantage points are better than others. For instance, the view from the initial object is quite sparse. Every object X is mapped to a singleton set corresponding to the unique mapping $0 \to X$.

The view from the terminal object is more interesting: it maps all objects to their sets of (global) elements.

The Yoneda lemma may be considered one of the most profound statements, or one of the most trivial statements in category theory. Let's start with the profound version.

Consider two models of $\mathcal{C}$ in **Set**: one given by the hom-functor $\mathcal{C}(A, -)$, that is the panoramic view of $\mathcal{C}$ from the vantage point of A; and another given by some functor $F\colon \mathcal{C} \to \mathbf{Set}$. A natural transformation between them embeds one model in the other. It turns out that the set of such natural transformations is fully determined by the value of F at A.

The set of natural transformation is the hom-set in the functor category $[\mathcal{C}, \mathbf{Set}]$, so this is the formal statement of the Yoneda lemma:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(A, -), F) \cong FA$$

The reason this works is because all the mappings involved in this theorem are bound by the requirements of preserving the structure of the category $\mathcal{C}$ and the structure of its models. In particular, naturality conditions impose a huge set of constraints on the way the mapping propagates from one point to another.

The proof of the Yoneda lemma starts with a single identity arrow and lets naturality propagate it across the whole category.

Here's the sketch of the proof. It consists of two parts: First, given a natural transformation we construct an element of FA. Second, given an element of FA we construct the corresponding natural transformation.

First, let's pick an arbitrary element on the left-hand side: a natural transformation α. Its component at X is a function

$$\alpha_X\colon \mathcal{C}(A, X) \to FX$$

We can now apply the Yoneda trick: substitute A for X and pick the identity id_A as the element of $\mathcal{C}(A, A)$. This gives us an element $\alpha_A(id_A)$ in the set FA.

Now the other way around. Take an element y of the set FA. We want to implement a natural transformation that takes an arrow h from $\mathcal{C}(A, X)$ and produces an element of FX. This is simply done by lifting the arrow h using F. We get a function

$$Fh\colon FA \to FX$$

We can apply this function to y to get an element of FX. We take this element as the action of α_X on h.

Exercise 9.7.1. *Show that the mapping*

$$\mathcal{C}(A, X) \to FX$$

defined above is a natural transformation. Hint: Vary X using some $f\colon X \to Y$.

The isomorphism in the Yoneda lemma is natural not only in A but also in F. In other words, you can "move" from the functor F to another functor G by applying an arrow in the functor category, that is a natural transformation. This is quite a leap in the levels of abstraction, but all the definitions of functoriality and naturality work equally well in the functor category, where objects are functors, and arrows are natural transformations.

Yoneda lemma in programming

Now for the trivial part: The proof of the Yoneda lemma translates directly to Haskell code. We start with the type of natural transformation between the hom-functor `a->x` and some functor `f`, and show that it's equivalent to the type of `f` acting on `a`.

```
forall x. (a -> x) -> f x.    -- is isomorphic to (f a)
```

We produce a value of the type `f a` using the standard Yoneda trick

```
yoneda :: Functor f => (forall x. (a -> x) -> f x) -> f a
yoneda g = g id
```

Here's the inverse mapping:

```
yoneda_1 :: Functor f => f a -> (forall x. (a -> x) -> f x)
yoneda_1 y = \h -> fmap h y
```

Note that we are cheating a little by mixing types and sets. The Yoneda lemma in the present formulation works with **Set**-valued functors. Again, the correct incantation is to say that we use the enriched version of the Yoneda lemma in a self-enriched category.

The Yoneda lemma has some interesting applications in programming. For instance, let's consider what happens when we apply the Yoneda lemma to the identity functor. We get the isomorphism between the type `a` (the identity functor acting on `a`) and

```
forall x. (a -> x) -> x
```

We interpret this as saying that any data type `a` can be replaced by a higher order polymorphic function. This function takes another function—called a handler, a callback, or a *continuation*—as an argument.

This is the standard continuation passing transformation that's used a lot in distributed programming, when the value of type `a` has to be retrieved from a remote

server. It's also useful as a program transformation that turns recursive algorithms into tail-recursive functions.

Continuation-passing style is difficult to work with because the composition of continuations is highly nontrivial, resulting in what programmers often call a "callback hell." Fortunately continuations form a monad, which means their composition can be automated.

The contravariant Yoneda lemma

By reversing a few arrow, the Yoneda lemma can be applied to contravariant functors as well. It works on natural transformations between the contravariant hom-functor $\mathcal{C}(-, A)$ and a contravariant functor F:

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, A), F) \cong FA$$

This is the Haskell implementation of the mapping:

```
coyoneda :: Contravariant f => (forall x. (x -> a) -> f x) -> f a
coyoneda g = g id
```

And this is the inverse transformation:

```
coyoneda_1 :: Contravariant f => f a -> (forall x. (x -> a) -> f x)
coyoneda_1 y = \h -> contramap h y
```

9.8 Yoneda Embedding

In a closed category, we have exponential objects that serve as stand-ins for hom-sets. This is obviously a thing in the category of sets, where hom-sets, being sets, are automatically objects. But in the category of categories **Cat**, hom-sets are sets of functors, and it's not immediately obvious that they can be promoted to objects—that is categories. But, as we've seen, they can! Functors between any two categories form a functor category, with natural transformations as arrows.

Because of that, it's possible to curry functors just like we curried functions. A functor from a product category can be viewed as a functor returning a functor. In other words, **Cat** is a closed (symmetric) monoidal category.

In particular, we can apply currying to the hom-functor $\mathcal{C}(A, B)$. It is a profunctor, or a functor from the product category:

$$\mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$$

But it's also a contravariant functor in A. For every A in $\mathcal{C}^{op}$ it produces a functor $\mathcal{C}(A, -)$, that is an object in the functor category $[\mathcal{C}, \mathbf{Set}]$. We can write this mapping as:

$$\mathcal{C}^{op} \to [\mathcal{C}, \mathbf{Set}]$$

Alternatively, we can focus on B and get a contravariant functor $\mathcal{C}(-, B)$. This mapping can be written as

$$\mathcal{C} \to [\mathcal{C}^{op}, \mathbf{Set}]$$

Both mappings are functorial, which means that, for instance, an arrow in $\mathcal{C}$ is mapped to a natural transformation in $[\mathcal{C}^{op}, \mathbf{Set}]$.

These **Set**-valued functor categories are common enough that they have special names. The functors in $[\mathcal{C}^{op}, \mathbf{Set}]$ are called *presheaves*, and the ones in $[\mathcal{C}, \mathbf{Set}]$ are called *co-presheaves*. (The names come from algebraic topology.)

Let's focus our attention on the following reading of the hom-functor:

$$\mathcal{Y} \colon \mathcal{C} \to [\mathcal{C}^{op}, \mathbf{Set}]$$

It takes an object X and maps it to a presheaf $\mathcal{C}(-, X)$, which can be visualized as the totality of views of X from all possible directions.

Let's also review its action on arrows. The functor $\mathcal{Y}$ lifts an arrow $f \colon X \to Y$ to a mapping of presheaves:

$$\alpha \colon \mathcal{C}(-, X) \to \mathcal{C}(-, Y)$$

The component of this natural transformation at some Z is a function between hom-sets:

$$\alpha_Z \colon \mathcal{C}(Z, X) \to \mathcal{C}(Z, Y)$$

which is simply implemented as the post-composition $(f \circ -)$.

Such a functor $\mathcal{Y}$ can be thought of as creating a model of $\mathcal{C}$ in the presheaf category. But this is no run-of-the-mill model—it's an *embedding* of one category inside another. This particular one is called the *Yoneda embedding*.

First of all, every object of $\mathcal{C}$ is mapped to a different object (presheaf) in $[\mathcal{C}^{op}, \mathbf{Set}]$. We say that it's "injective on objects." But that's not all: every arrow in $\mathcal{C}$ is mapped to a different arrow. We say that the embedding functor is *faithful*. If that weren't enough, the mapping of hom-sets is also surjective, meaning that every arrow between objects in $[\mathcal{C}^{op}, \mathbf{Set}]$ comes from some arrow in $\mathcal{C}$. We say that the functor is *full*. Altoghether, the embedding is *fully faithful*.

The latter fact is the direct consequence of the Yoneda lemma. We know that, for any functor $F \colon \mathcal{C}^{op} \to \mathbf{Set}$, we have a natural isomorphism:

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, X), F) \cong FX$$

In particular, we can substitute another hom-functor $\mathcal{C}(-, Y)$ for F:

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, X), \mathcal{C}(-, Y)) \cong \mathcal{C}(X, Y)$$

The left-hand side is the hom-set in the presheaf category and the right-hand side is the hom-set in $\mathcal{C}$. They are isomorphic, which proves that the embedding is fully faithful.

Let's have a closer look at this isomorphism. Let's pick an element of the set $\mathcal{C}(X, Y)$—an arrow f. The isomorphism maps it to a natural transformation whose component at Z is a function:

$$\mathcal{C}(Z, X) \to \mathcal{C}(Z, Y)$$

This mapping is implemented as post-composition $(f \circ -)$.

In Haskell, we would write it as:

```
toNat :: (x -> y) -> (forall z. (z -> x) -> (z -> y))
toNat f = \h -> f . h
```

In fact, this syntax works too:

```
toNat f = (f . )
```

The inverse mapping is:

```
fromNat :: (forall z. (z -> x) -> (z -> y)) -> (x -> y)
fromNat alpha = alpha id
```

(Notice the use of the Yoneda trick again.)

This isomorphism maps identity to identity and composition to composition. That's because it's implemented as post-composition, and post-composition preserves both identity and composition. We've seen this in the chapter on isomorphisms:

$$((f \circ g) \circ -) = (f \circ -) \circ (g \circ -)$$

Because it preserves composition and identity, this isomorphism also preserves *isomorphisms*. So if X is isomorphic to Y then the presheaves $\mathcal{C}(-, X)$ and $\mathcal{C}(-, Y)$ are isomorphic, and vice versa.

This is exactly the result that we've been using all along to prove numerous isomorphisms in previous chapters.

9.9 Representable Functors

Objects in a co-presheaf category are functors that assign sets to objects in $\mathcal{C}$. Some of these functors work by picking a reference object A and assigning, to all objects X, their hom-sets $\mathcal{C}(A, X)$. Such functors, and all the functors isomorphic to those, are called *representable*. The whole functor is "represented" by a single object A.

In a closed category, the functor which assigns the set of elements of X^A to every object X is represented by A, because the set of elements of X^A is isomorphic to $\mathcal{C}(A, X)$:

$$\mathcal{C}(1, X^A) \cong \mathcal{C}(1 \times A, X) \cong \mathcal{C}(A, X)$$

Seen this way, the representing object A is like a logarithm of a functor.

The analogy goes deeper: just like a logarithm of a product is a sum of logarithms, a representing object for a product data type is a sum. For instance, the functor that squares its argument using a product, $Fx = x \times x$, is represented by 2, which is the sum $1 + 1$.

Representable functors play a very special role in the category of **Set**-valued functors. Notice that the Yoneda embedding maps objects of $\mathcal{C}$ to representable presheaves. It maps an object X to a presheaf represented by X:

$$\mathcal{Y} \colon X \mapsto \mathcal{C}(-, X)$$

We can find the entire category $\mathcal{C}$, objects and morphisms, embedded inside the presheaf category as representable functors. The question is, what else is there in the presheaf category "in between" representable functors?

Just like rational numbers are dense among real numbers, so representables are "dense" among (co-) presheaves. Every real number may be approximated by rational numbers. Every presheaf is a colimit of representables (and every co-presheaf, a limit). We'll come back to this topic when we talk about (co-) ends.

The guessing game

The idea that objects can be described by the way they interact with other objects is sometimes illustrated by playing imaginary guessing games. One category theorist picks a secret object in a category, and the other has to guess which object it is (up to isomorphism, of course).

The guesser is allowed to point at objects, and use them as "probes" into the secret object. The opponent is supposed to respond, each time, with a set: the set of arrows from the probing object A to the secret object X. This, of course, is the hom-set $\mathcal{C}(A, X)$.

The totality of these answers, as long as the opponent is not cheating, will define a presheaf $F\colon \mathcal{C} \to \mathbf{Set}$, and the object they are hiding is its representing object.

But how do we know they are not cheating? To test that, we have to be able to ask questions about arrows. For every arrow we select, they should give us a function between two sets—the sets they gave us for its endpoints. We can then check if all identity arrows are mapped to identity functions, and whether compositions of arrows map to compositions of functions. In other words, we'll be able to verify that F is a functor.

However, a clever enough opponent may still fool us. The presheaf they are revealing to us may describe a fantastical object—a figment of their imagination—and we won't be able to tell. It turns out that such imaginary objects are often as interesting as the real ones.

Representable functors in programming

In Haskell, we define a class of representable functors using two functions that witness the isomorphism: `tabulate` turns a function into a lookup table, and `index` uses the representing type `Key` to index into it.

```
class Representable f where
  type Key f :: Type
  tabulate :: (Key f -> a) -> f a
  index    :: f a -> (Key f -> a)
```

Algebraic data types that use sums are not representable—there is no formula for taking a logarithm of a sum. List type is defined as a sum, so it's not representable.

However, an infinite stream is. Conceptually, such a stream is like an infinite tuple, which is technically a product. A stream is represented by the type of natural numbers. In other words, an infinite stream is equivalent to a mapping out of natural numbers.

```
data Stream a = Stm a (Stream a)
```

Here's the instance definition:

```
instance Representable Stream where
  type Key Stream = Nat
  tabulate g = tab Z
    where
      tab n = Stm (g n) (tab (S n))
  index stm = \n -> ind n stm
    where
```

```
    ind Z (Stm a _) = a
    ind (S n) (Stm _ as) = ind n as
```

Representable types are useful in implementing memoization of functions.

Exercise 9.9.1. *Implement the* `Representable` *instance for* `Pair`*:*

```
data Pair x = Pair x x
```

Exercise 9.9.2. *Is the constant functor that maps everything to the terminal object representable? Hint: what's the logarithm of 1?*

In Haskell, such a functor could be implemented as:

```
data Unit a = U
```

Implement the instance of `Representable` *for it.*

Exercise 9.9.3. *The list functor is not representable. But can it be considered a sum or representables?*

9.10 2-category Cat

In the category of categories, **Cat**, the hom-sets are not just sets. Each of them can be promoted to a functor category, with natural transformations playing the role of arrows. This kind of structure is called a 2-category.

In the language of 2-categories, objects are called 0-cells, arrows between them are called 1-cells, and arrows between arrows are called 2-cells.

The obvious generalization of that picture would be to have 3-cells that go between 2-cells and so on. An n-category has cells going up to the n-th level.

But why not have arrows all the way down? Enter infinity categories. Far from being a curiosity, ∞-categories have practical applications. For instance they are used in algebraic topology to describe points, paths between points, surfaces swiped by paths, volumes swiped by surfaces, and so on, ad infinitum.

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 $(- \times B)$ applied to A. On the right we have the functor $(-)^B$ applied to 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.

In components, this isomorphism tells us that, given a mapping $\phi \in \mathcal{C}(L_B A, C)$, there is a unique mapping $\phi^T \in \mathcal{C}(A, R_B C)$ and vice versa. These mappings are sometimes called the *transpose* of each other—the nomenclature taken from matrix algebra.

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 **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 **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 **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.

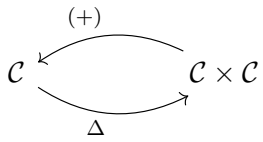
We can spot a pair of functors here as well. On the left we have the functor that takes a pair of objects $\langle A, B\rangle$ and produces their sum $A + B$

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

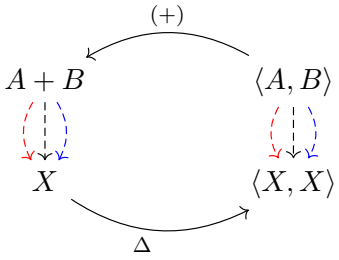
On the right, we have the diagonal functor Δ going in the opposite direction

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

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



and an isomorphism between the hom-sets:



The product adjunction

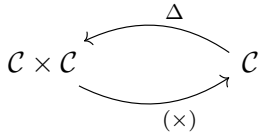
We can apply the same reasoning to the definition of a product. This time we have a natural isomorphism between pairs of arrows and and a mapping into the product.

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

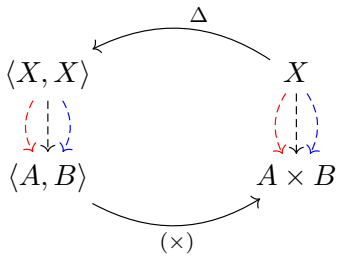
Replacing pairs of arrows with arrows in the product category we get:

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

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

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

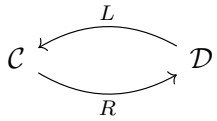
and the right functor:

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

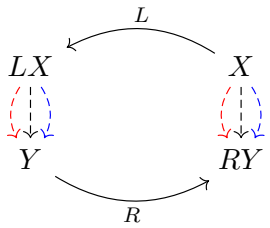
The adjunction $L \dashv R$ is defined as a natural isomorphism between two hom-sets.

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

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.

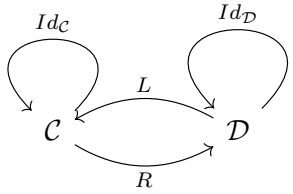
We have a problem with functors, though. On the one hand, they are objects in the functor category, so isomorphisms are the way to go; on the other hand, they are arrows in **Cat** so maybe it's okay to compare them for equality?

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:

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

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:

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

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

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

We get a family of arrows:

$$\varepsilon_Y \colon L(RY) \to Y$$

which form another natural transformation called the *counit* of the adjunction:

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

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.

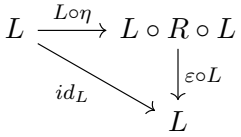
Similarly, ε can be used to eliminate the other composite $L \circ R$.

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).

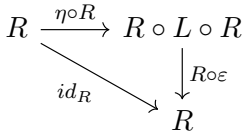
The first triangle identity is the condition that this chain of transformations result in the identity natural transformation. Pictorially:



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.

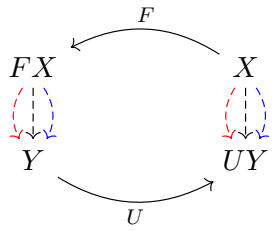
10.7 Free-Forgetful Adjunctions

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*.

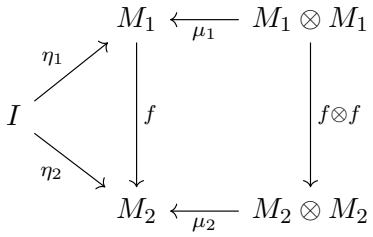


A classic example of a free/forgetful adjunction is the construction of the free monoid.

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 $\mathbf{Mon}(\mathbf{Set})$ 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

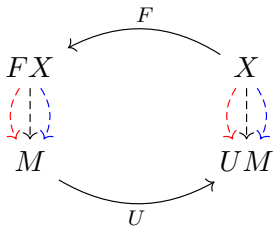
$$F \colon \mathbf{Set} \to \mathbf{Mon}(\mathbf{Set})$$

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`.

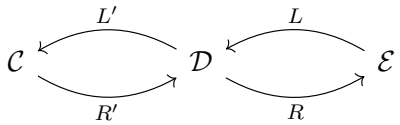
It's easy to see that an empty list is mapped to the monoidal unit. It's not too hard to see that a concatenation of two lists is mapped to the monoidal product of the results. So, indeed, `foldMap` produces a monoid morphism.

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.*

10.8 The Category of Adjunctions

We can define composition of adjunctions by taking advantage of the composition of functors that define them. Two adjunctions, $L \dashv R$ and $L' \dashv R'$, are composable if they share the category in the middle:



By composing the functors we get a new adjunction $(L' \circ L) \dashv (R \circ R')$.

Indeed, let's consider the hom-set:

$$\mathcal{C}(L'(Le), c)$$

Using the $L' \dashv R'$ adjunction, we can transpose L' to the right, where it becomes R':

$$\mathcal{D}(Le, R'c)$$

and using $L \dashv R$ we can similarly transpose L:

$$\mathcal{E}(e, R(R'c))$$

Combining these two isomorphisms, we get the composite adjunction:

$$\mathcal{C}((L' \circ L)e, c) \cong \mathcal{E}(e, (R \circ R')c)$$

Because functor composition is associative, the composition of adjunctions is also associative. It's easy to see that a pair of identity functors forms a trivial adjunction that serves as the identity with respect to composition of adjunctions. Therefore we can define a category **Adj**(**Cat**) in which objects are categories and arrows are adjunctions (by convention, pointing in the direction of the left adjoint).

Adjunctions can be defined purely in terms of functors and natural transformations, that is 1-cells and 2-cells in the 2-category **Cat**. There is nothing special about **Cat**, and in fact adjunctions can be defined in any 2-category. Moreover, the category of adjunctions is itself a 2-category.

Chapter 11

Dependent Types

We've seen types that depend on other types. They are defined using type constructors with type parameters, like `Maybe` or `[]`. Most programming languages have some support for generic data types—data types parameterized by other data types.

Categorically, such types are modeled as functors [1].

A natural generalization of this idea is to have types that are parameterized by values. For instance, it's often advantageous to encode the length of a list in its type. A list of length zero would have a different type than a list of length one, and so on.

Types parameterized by values are called *dependent types*. There are languages like Idris or Agda that have full support for dependent types. It's also possible to implement dependent types in Haskell, but support for them is still rather patchy.

The reason for using dependent types in programming is to make programs provably correct. In order to do that, the compiler must be able to check the assumptions made by the programmer.

Haskell, with its strong type system, is able to uncover a lot of bugs at compile time. For instance, it won't let you write `a <> b` (infix notation for `mappend`), unless you provide the `Monoid` instance for the type of your variables.

However, within Haskell's type system, there is no way to express or, much less enforce, the unit and associativity laws for the monoid. For that, the instance of the `Monoid` type class would have to carry with itself proofs of equality (not actual code):

```
assoc :: m <> (n <> p) = (m <> n) <> p
lunit :: mempty <> m = m
runit :: m <> mempty = m
```

Dependent types, and equality types in particular, pave the way towards this goal.

The material in this chapter is more advanced, and not used in the rest of the book, so you may safely skip it on first reading.

11.1 Dependent Vectors

We'll start with the standard example of a counted list, or a vector:

[1] A type constructor that has no `Functor` instance can be thought of as a functor from a discrete category—a category with no arrows other than identities

```
data Vec n a where
    VNil  :: Vec Z a
    VCons :: a -> Vec n a -> Vec (S n) a
```

The compiler will recognize this definition as dependently typed if you include the following language pragmas:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
```

The first argument to the type constructor is a natural number `n`. Notice: this is a value, not a type. The type checker is able to figure this out from the usage of `n` in the two data constructors. The first one creates a vector of the type `Vec Z a`, and the second creates a vector of the type `Vect (S n) a`, where `Z` and `S` are defined as the constructors of natural numbers:

```
data Nat = Z | S Nat
```

We can be more explicit about the parameters if we use the pragma:

```
{-# LANGUAGE KindSignatures #-}
```

and import the library:

```
import Data.Kind
```

We can then specify that `n` is a `Nat`, whereas `a` is a `Type`:

```
data Vec (n :: Nat) (a :: Type) where
    VNil  :: Vec Z a
    VCons :: a -> Vec n a -> Vec (S n) a
```

Using one of these definitions we can, for instance, construct a vector (of integers) of length zero:

```
emptyV :: Vec Z Int
emptyV = VNil
```

It has a different type than a vector of length one:

```
singleV :: Vec (S Z) Int
singleV = VCons 42 VNil
```

and so on.

We can now define a dependently typed function that returns the first element of a vector:

```
headV :: Vec (S n) a -> a
headV (VCons a _) = a
```

This function is guaranteed to work exclusively with non-zero-length vectors. These are the vectors whose size matches `(S n)`, which cannot be `Z`. If you try to call this function with `emptyV`, the compiler will flag the error.

Another example is a function that zips two vectors together. Encoded in its type signature is the requirement that the two vectors be of the same size `n` (the result is also of the size `n`):

```
zipV :: Vec n a -> Vec n b -> Vec n (a, b)
zipV (VCons a as) (VCons b bs) = VCons (a, b) (zipV as bs)
```

```
zipV VNil VNil = VNil
```

Exercise 11.1.1. *Implement the function* `tailV` *that returns the tail of the non-zero-length vector. Try calling it with* `emptyV`*.*

11.2 Dependent Types Categorically

The easiest way to visualize dependent types is to think of them as families of types indexed by elements of a set. In the case of counted vectors, the indexing set would be the set of natural numbers $\mathbb{N}$.

The zeroth type would be the unit type `()` representing an empty vector. The type corresponding to `(S Z)` would be `a`; then we'd have a pair `(a, a)`, a triple `(a, a, a)` and so on, with higher and higher powers of `a`.

If we want to talk about the whole family as one big set, we can take the sum of all these types. For instance, the sum of all powers of a is the familiar list type, a.k.a, a free monoid:

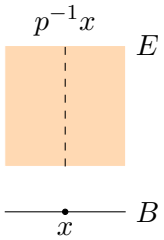
$$List(a) = \coprod_{n:\mathbb{N}} a^n$$

Fibrations

Although intuitively easy to visualize, this point of view doesn't generalize nicely to category theory, where we don't like mixing sets with objects. So we turn this picture on its head and instead of talking about injecting family members into the sum, we consider a mapping that goes in the opposite direction.

This, again, we can first visualize using sets. We have one big set E describing the whole family, and a function p called the projection, or a *display map*, that goes from E down to the indexing set B (also called the *base*).

This function will, in general, map multiple elements to one. We can then talk about the inverse image of a particular element $x \in B$ as the set of elements that get mapped down to it by p. This set is called the *fiber* and is written $p^{-1}x$ (even though, in general, p is not invertible in the usual sense). Seen as a collection of fibers, E is often called a *fiber bundle* or just a bundle.



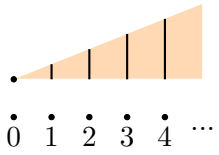
Now forget about sets. A *fibration* in an arbitrary category is a pair of objects E and B and an arrow $p\colon E \to B$.

So this is really just an arrow, but the context is everything. When an arrow is called a fibration, we use the intuition from sets, and imagine its source E as a collection of fibers, with p projecting each fiber down to a single point in the base B.

We will therefore model type families as fibrations. For instance, our counted-vector family can be represented as a fibration whose base is the type of natural numbers. The whole family is a sum (coproduct) of consecutive powers (products) of A:

$$List(A) = A^0 + A^1 + A^2 + ... = \coprod_{n\colon \mathbb{N}} A^n$$

with the zeroth power—the initial object—representing a vector of size zero.



The projection $p\colon List(A) \to \mathbb{N}$ is the familiar *length* function.

Slice categories

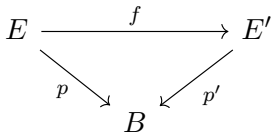
In category theory we like to describe things in bulk—defining internal structure of things by structure-preserving maps between them. Such is the case with fibrations.

If we fix the base object B and consider all possible source objects in the category $\mathcal{C}$, and all possible projections down to B, we get a *slice category* $\mathcal{C}/B$ (also known as an over-category).

An object in the slice category is a pair $\langle E, p\rangle$, with $p\colon E \to B$. An arrow between two objects $\langle E, p\rangle$ and $\langle E', p'\rangle$ is an arrow $f\colon E \to E'$ that commutes with the projections, that is:

$$p' \circ f = p$$

Again, the best way to visualize this is to notice that such an arrow maps fibers of p to fibers of p'. It's a "fiber-preserving" mapping between bundles.

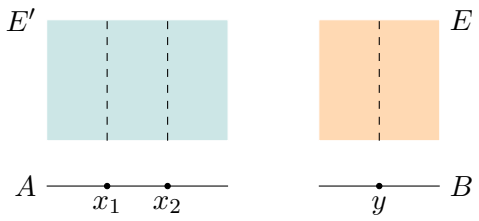


Our counted vectors can be seen as objects in the slice category $\mathcal{C}/\mathbb{N}$ given by pairs $\langle List(A), length\rangle$.

Pullbacks

Let's start with a particular fibration $p\colon E \to B$ and ask ourselves the question: what happens when we change the base from B to some A that is related to it through a mapping $f\colon A \to B$. Can we "pull the fibers back" along f?

Again, let's think about sets first. Imagine picking a fiber in E over some point $y \in B$ that is in the image of f. We can plant this fiber over all points in A that are in the inverse image $f^{-1}y$. If multiple points in A are mapped to the same point in B, we just duplicate the corresponding fiber. This way, every point in A will have a fiber sticking out of it. The sum of all these fibers will form a new bundle E'.



We have thus constructed a new fibration with the base A. Its projection $p' : E' \to A$ maps each point in a given fiber to the point over which this fiber was planted. There is also an obvious mapping $g : E' \to E$ that maps fibers to their corresponding fibers.

By construction, this new fibration $\langle E', p' \rangle$ satisfies the condition:

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

which can be represented as a commuting square:

$$\begin{array}{ccc} E' & \xrightarrow{g} & E \\ {\scriptstyle p'}\downarrow & & \downarrow{\scriptstyle p} \\ A & \xrightarrow{f} & B \end{array}$$

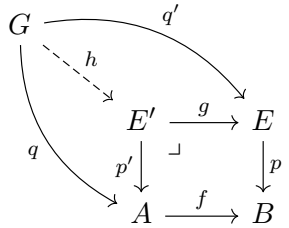
In **Set**, we can explicitly construct E' as a *subset* of the cartesian product $A \times E$ with $p' = \pi_1$ and $g = \pi_2$ (the two cartesian projections). An element of E' is a pair $\langle b, e \rangle$, such that:

$$f(b) = p(e)$$

This commuting square is the starting point for the categorical generalization. However, even in **Set** there are many different fibrations over A that make this diagram commute. We have to pick the universal one. Such a universal construction is called a *pullback*, or a *fibered product*.

A pullback of $p : E \to B$ along $f : A \to B$ is an object E' together with two arrows $p' : E' \to A$ and $g : E' \to E$ that makes the above diagram commute, and that satisfies the universal condition.

The universal condition says that, for any other candidate object G with two arrows $q' : G \to E$ and $q : G \to A$ such that $p \circ q' = f \circ q$, there is a unique arrow $h : G \to E'$ that makes the two triangles commute:



The angle symbol in the corner of the square is used to mark pullbacks.

If we look at the pullback through the prism of fibrations, E is a bundle over B, and we are constructing a new bundle E' out of the fibers taken from E. Where we plant these fibers over A is determined by (the inverse image of) f. This procedure makes E' a bundle over both A and B, the latter with the projection $p \circ g = f \circ p'$.

G in this picture is some other bundle over A with the projection q. It is simultaneously a bundle over B with the projection $f \circ q = p \circ q'$. The unique mapping h maps the fibers of G given by q^{-1} to fibers of E' given by p'.

All mappings in this picture work on fibers. Some of them rearrange fibers over new bases—that's what a pullback does. This is analogous to what natural transformations do to containers. Others modify individual fibers—the mapping $h\colon G \to E'$ works like this. This is analogous to what `fmap` does to containers. The universal condition then tells us that q' can be factored into a transformation of fibers h, followed by the rearrangement g.

It's worth noting that picking the terminal object as the pullback target gives us automatically the definition of the categorical product:

$$\begin{array}{ccc} B \times E & \xrightarrow{\pi_2} & E \\ {\scriptstyle \pi_1}\downarrow & \lrcorner & \downarrow{\scriptstyle !} \\ B & \xrightarrow{!} & 1 \end{array}$$

Alternatively, we can think of this picture as a generalization of the diagonal functor. Normally, the diagonal functor ΔE would duplicate E. Here, you can imagine π_1 fibrating $B \times E$ into as many copies of E as there are elements in B. We'll use this analogy when we talk about the dependent sum and product.

Conversely, a single fiber can be extracted from a fibration by pulling it back to the terminal object. In this case the mapping $x\colon 1 \to B$ picks an element of the base, and the pullback along it extracts a single fiber F:

$$\begin{array}{ccc} F & \xrightarrow{g} & E \\ {\scriptstyle !}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ 1 & \xrightarrow{x} & B \end{array}$$

The arrow g injects this fiber back into E. By varying x we can pick different fibers in E.

Exercise 11.2.1. *Show that the pullback with the terminal object as the target is the product.*

Exercise 11.2.2. *Show that a pullback can be defined as a limit of the diagram from a stick-figure category with three objects:*

$$A \to B \leftarrow C$$

Exercise 11.2.3. *Show that a pullback in $\mathcal{C}$ with the target B is a product in the slice category $\mathcal{C}/B$. Hint: Define two projections as morphisms in the slice category. Use universality of the pullback to show the universality of the product.*

Base-change functor

We used a cartesian closed category as a model for programming. To model dependent types, we need to impose an additional condition: We require the category to be *locally cartesian closed*. This is a category in which all slice categories are cartesian closed.

In particular, such categories have all pullbacks, so it's always possible to change the base of any fibration. Base change induces a mapping between slice categories that is functorial.

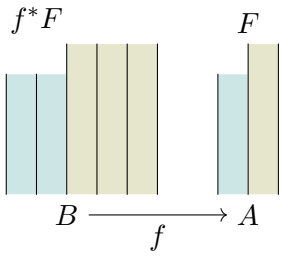
Given two slice categories $\mathcal{C}/B$ and $\mathcal{C}/A$ and an arrow between bases $f\colon B \to A$ the base-change functor $f^*\colon \mathcal{C}/A \to \mathcal{C}/B$ maps a fibration $\langle F, p\rangle$ to the fibration $f^*\langle F, p\rangle = \langle f^*F, f^*p\rangle$, which is given by the pullback:

$$\begin{array}{ccc} f^*F & \xrightarrow{g} & F \\ {\scriptstyle f^*p}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ B & \xrightarrow{f} & A \end{array}$$

Notice that the functor f^* goes in the opposite direction to the arrow f.

To visualize the base-change functor let's consider how it works on sets. We have the intuition that the fibration p decomposes F into fibers over each point of A.

But here we have another fibration f that similarly decomposes B. Let's call these fibers "patches." For instance, if A is just a two-element set, then the fibration given by f splits B into two patches. The pullback takes a fiber from F and plants it over the whole patch in B. The resulting set f^*F looks like a patchwork, where each patch is planted with clones of one fiber from F.



Since we have a function from B to A that may map many elements to one, the fibration over B has finer grain than the coarser fibration over A. The simplest, least-effort way to turn the fibration of F over A to a fibration over B, is to spread the existing fibers over the patches defined by (the inverse of) f. This is the essence of the universal construction of the pullback.

You may also think of A as providing an *atlas* that enumerates all the patches in the *base* B.

In particular, if A is a singleton set (the terminal object), then we have only one fiber (the whole of F) and the bundle f^*F is a cartesian product $B \times F$. Such bundle is called a *trivial bundle*.

A non-trivial bundle is not a product, but it can be *locally* decomposed into products. Just as B is a sum of patches, so f^*F is a sum of products of these patches and the corresponding fibers of F.

In a locally cartesian closed category, the base change functor has both the left and the right adjoints. The left adjoint is called the dependent sum, and the right adjoint is called the dependent product (or dependent function).

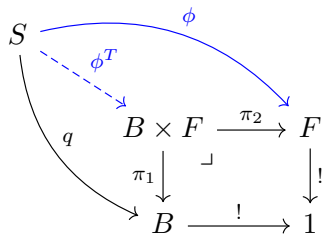
11.3 Dependent Sum

In type theory, the dependent sum, or the sigma type $\Sigma_{x:B}T(x)$, is defined as a type of pairs in which the type of the second component depends on the value of the first component. Our counted vector type can be thought of as a dependent sum. An element of this type is a natural number `n` paired with an element `(x_1, x_2, ... x_n)` of the n-tuple `(a, a, ... a)`.

The introduction rule for the dependent sum assumes that there is a family of types $T(x)$ indexed by elements of the base type B. Then an element of $\Sigma_{x:B}T(x)$ is constructed from a pair of elements $x\colon B$ and $y\colon T(x)$.

Categorically, dependent sum is modeled as the left adjoint of the base-change functor.

To see this, let's first revisit the definition of a pair, which is an element of a product. We've noticed before that a product can be written as a pullback from the terminal object. Here's the universal construction for the product/pullback (the notation anticipates the target of this construction):



We have also seen that the product can be defined using an adjunction. We can spot this adjunction in our diagram: for every pair of arrows $\langle \phi, q\rangle$ there is a unique arrow ϕ^T that makes the triangles commute.

Notice that, if we keep q fixed, we get a one-to-one correspondence between the arrows ϕ and ϕ^T. This is the adjunction we're interested in.

We can now put our fibrational glasses on and notice that $\langle S, q\rangle$ and $\langle B \times F, \pi_1\rangle$ are two fibrations over the same base B. The commuting triangle makes ϕ^T a morphism in the slice category $\mathcal{C}/B$, or a fiber-wise mapping. In other words ϕ^T is a member of the hom-set:

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} B \times F \\ \pi_1 \end{matrix} \right\rangle\right)$$

Since ϕ is a member of the hom-set $\mathcal{C}(S, F)$, we can rewrite the one-to-one correspondence between ϕ and ϕ^T as an isomorphism of hom-sets:

$$\mathcal{C}(S, F) \cong (\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} B \times F \\ \pi_1 \end{matrix} \right\rangle\right)$$

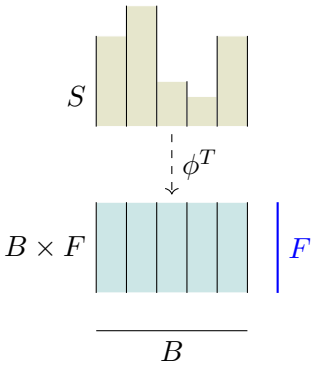
In fact, it's an adjunction in which the left functor is the forgetful functor $L\colon \mathcal{C}/B \to \mathcal{C}$ that maps $\langle S, q\rangle$ to S, thus forgetting the fibration.

If you squint at this adjunction hard enough, you can see the outlines of the definition of S as a categorical sum (coproduct).

Firstly, on the left you have a mapping out of S. Think of S as the sum of fibers that are defined by the fibration $\langle S, q\rangle$.

Secondly, recall that the fibration $\langle B \times F, \pi_1\rangle$ can be though of as a generalization of the diagonal functor, producing many copies of F planted over B. Then the right

hand side looks like a bunch of arrows, each mapping a different fiber of S to the same fiber F.

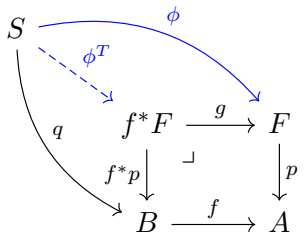


For comparison, this is the definition of a coproduct of two fibers, with $S = F_1 + F_2$:

$$\mathcal{C}(F_1 + F_2, F) \cong (\mathcal{C} \times \mathcal{C})(\langle F_1, F_2 \rangle, \Delta F)$$

Seen in this light, a dependent sum is just a sum of many fibers. In **Set** it's a tagged union. Each individual set is a fiber of S under q.

We can generalize our diagram by replacing the terminal object with an arbitrary base A. We now have a fibration $\langle F, p \rangle$, and we get the pullback square that defines the base-change functor f^*:



The universality of the pullback results in the following isomorphism of hom-sets:

$$(\mathcal{C}/A)\left(\left\langle \begin{matrix} S \\ f \circ q \end{matrix} \right\rangle, \left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right) \cong (\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, f^* \left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right)$$

Here, ϕ is an element of the left-hand side and ϕ^T is the corresponding element of the right-hand side.

We interpret this isomorphism as the adjunction between the base change functor f^* on the right and the dependent sum functor on the left.

$$(\mathcal{C}/A)\left(\Sigma_f \left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right) \cong (\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, f^* \left\langle \begin{matrix} F \\ p \end{matrix} \right\rangle\right)$$

The dependent sum is thus defined by this formula:

$$\Sigma_f \left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle = \left\langle \begin{matrix} S \\ f \circ q \end{matrix} \right\rangle$$

This says that, if S is fibered over B using q, and there is a mapping f from B to A, then S is automatically fibered over A, the projection being the composition $f \circ q$.

We've seen before that, in **Set**, f defines patches within B. Fibers of F are replanted in these patches to form f^*F. Locally—that is within each patch—f^*F looks like a cartesian product. Locally, the mapping ϕ^T is a product of fiber-wise mappings. So, locally, S is a sum of fibers. Globally, these patches of fibers are summed together into one whole.

Existential quantification

In the *propositions as types* interpretation, type families correspond to families of propositions. The dependent sum type $\Sigma_{x:B}\, T(x)$ corresponds to the proposition: There exists an x for which $T(x)$ is true:

$$\exists_{x:B}\, T(x)$$

Indeed, a term of the type $\Sigma_{x:B}\, T(x)$ is a pair of an element $x\colon B$ and an element $y\colon T(x)$—which shows that $T(x)$ is inhabited for some x.

11.4 Dependent Product

In type theory, the dependent product, or dependent function, or pi-type $\Pi_{x:B}T(x)$, is defined as a function whose return type depends on the value of its argument.

It's called a function, because you can evaluate it. Given a dependent function $f\colon \Pi_{x:B}T(x)$, you may apply it to an argument $x\colon B$ to get a value $f(x)\colon T(x)$.

Dependent product in Haskell

A simple example is a function that constructs a vector of a given size and fills it with copies of a given value:

```
replicateV :: a -> SNat n -> Vec n a
replicateV _ SZ  = VNil
replicateV x (SS n) = VCons x (replicateV x n)
```

At the time of this writing, Haskell's support for dependent types is limited, so the implementation of dependent functions requires the use of singleton types. In this case, the number that is the argument to `replicateV` is passed as a singleton natural:

```
data SNat n where
  SZ :: SNat Z
  SS :: SNat n -> SNat (S n)
```

(Note that `replicateV` is a function of two arguments, so it can be either considered a dependent function of a pair, or a regular function returning a dependent function.)

Dependent product of sets

Before we describe the categorical model of dependent functions, it's instructive to consider how they work on sets. A dependent function selects one element from each set $T(x)$.

You may visualize this selection as a large tuple—or an element of the cartesian product that you get by multiplying together all the sets in the family. This is the meaning of the product notation, $\Pi_{x:B}T(x)$. For instance, in the trivial case of B a two-element set $\{1, 2\}$, a dependent function is just a cartesian product $T(1) \times T(2)$.

In our example, `replicateV` picks a particular counted vector for each value of `n`. Counted vectors are equivalent to tuples so, for `n` equal zero, `replicateV` returns an empty tuple `()`; for `n` equals one it returns a single value `x`; for `n` equals two, it returns a homogenous pair `(x, x)`; etc.

The function `replicateV`, evaluated at some `x :: a`, is equivalent to an infinite tuple of tuples:

$$((), x, (x, x), (x, x, x), ...)$$

which is a specific element of the more general type:

$$((), a, (a, a), (a, a, a), ...)$$

Dependent product categorically

In order to build a categorical model of dependent functions, we need to change our perspective from a family of types to a fibration. We start with a bundle E/B fibered by the projection $p\colon E \to B$. A dependent function is called a *section* of this bundle.

If you visualize the bundle as a bunch of fibers sticking out from the base B, a section is like a haircut: it cuts through each fiber to produce a corresponding value. In physics, such sections are called fields—with spacetime as the base.

Just like we talked about a function object representing a set of functions, we can talk about an object $S(E)$ that represents a set of sections of a given bundle E.

Just like we defined function application as a mapping out of the product:

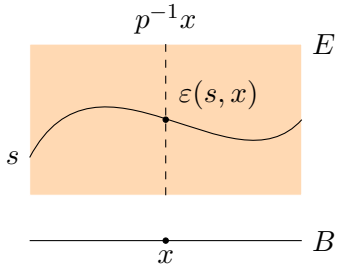
$$\varepsilon_{B,C}\colon C^B \times B \to C$$

we can define the dependent function application as a mapping:

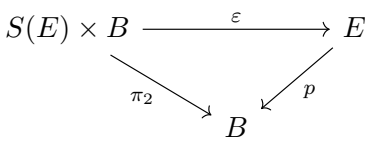
$$\varepsilon\colon S(E) \times B \to E$$

We can visualize it as picking a section s in S and an element x of the base B and producing a value in the bundle E. (In physics, this would correspond to measuring a field at a particular point in spacetime.)

But this time we have to insist that this value be in the correct fiber. If we project the result of applying ε to (s, x), we should get the same x.

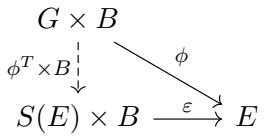


In other words, this diagram should commute:



This makes ε a morphism in the slice category $\mathcal{C}/B$.

And just like the exponential object was universal, so is the object of sections. The universality condition has the same form: for any other object G with an arrow $\phi\colon G\times B \to E$ there is a unique arrow $\phi^T : G \to S(E)$ that makes the following diagram commute:



The difference is that now both ε and ϕ are now morphisms in the slice category $\mathcal{C}/B$.

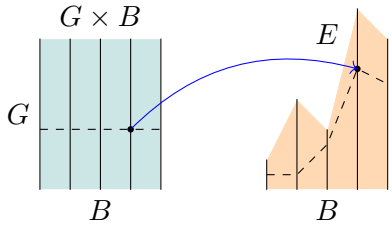
The one-to-one correspondence between ϕ and ϕ^T forms an adjunction:

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} G\times B \\ \pi_2 \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \cong \mathcal{C}\left(G, S(E)\right)$$

which we can use as the definition of the object of sections $S(E)$.

Recall that we can interpret the product $G \times B$ as a generalization of the diagonal functor. It takes copies of G and plants them as identical fibers over each element of B.

In **Set**, a single morphism on the left-hand side is equivalent to a family of functions, one per fiber. Any given $x \in G$ produces a horizontal slice of $G \times B$. Our family of functions maps this slice to the corresponding fibers of E thus creating a section of E.



The adjunction tells us that this family of mappings uniquely determines a function from G to $S(E)$. Every $x \in G$ is thus mapped to a different element s of $S(E)$. Therefore elements of $S(E)$ are in one-to-one correspondence with sections of E .

These are all set-theoretical intuitions. We can generalize them by first noticing that the right hand side of the adjunction can be easily expressed as a hom-set in the slice category $\mathcal{C}/1$ over the terminal object.

Indeed, there is one-to-one correspondence between objects X in $\mathcal{C}$ and objects $\langle X, !\rangle$ in $\mathcal{C}/1$. Arrows in $\mathcal{C}/1$ are arrows of $\mathcal{C}$ with no additional constraints. We therefore have:

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} G\times B \\ \pi_2 \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \cong (\mathcal{C}/1)\left(\left\langle \begin{matrix} G \\ ! \end{matrix} \right\rangle, \left\langle \begin{matrix} S(E) \\ ! \end{matrix} \right\rangle\right)$$

The next step is to "blur the focus" by replacing the terminal object with a more general base A.

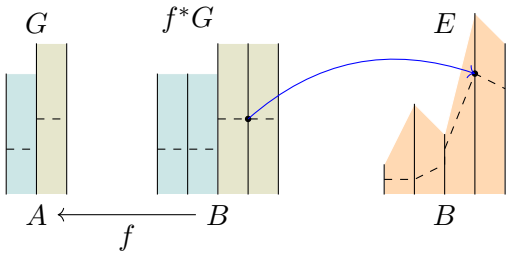
The right-hand side becomes a hom-set in the slice category $\mathcal{C}/A$ and G gets fibrated by some $q\colon G \to A$.

We can then replace the product $G \times B$ with a more general pullback of q along some $f\colon B \to A$.

$$\begin{array}{ccc} G \times B & \xrightarrow{\pi_1} & G \\ \downarrow{\scriptstyle \pi_2} & \lrcorner & \downarrow{\scriptstyle !} \\ B & \xrightarrow{!} & 1 \end{array} \longrightarrow \begin{array}{ccc} f^*G & \xrightarrow{g} & G \\ \downarrow{\scriptstyle f^*q} & \lrcorner & \downarrow{\scriptstyle q} \\ B & \xrightarrow{f} & A \end{array}$$

The result is that, instead of a bunch of G fibers over B, we get a pullback f^*G that is populated by groups of fibers from the fibration $q\colon G \to A$. This way, A serves as an atlas that enumerates all the uniform patches.

Imagine, for instance, that A is a two-element set. The fibration q will split G into two fibers. These will be our generic fibers. These fibers are now replanted over the two patches in B to form f^*G. The replanting is guided by f^{-1}.



The adjunction that defines the dependent function type is therefore:

$$(\mathcal{C}/B)\left(f^*\left\langle \begin{matrix} G \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \cong (\mathcal{C}/A)\left(\left\langle \begin{matrix} G \\ q \end{matrix} \right\rangle, \Pi_f \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right)$$

where $\Pi_f E$ is a suitable rearrangement of the object of sections $S(E)$. The adjunction is a mapping between morphisms in their respective slice categories:

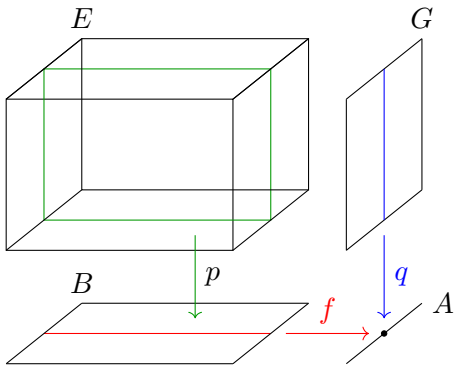
$$\begin{array}{ccc} f^*G & \xrightarrow{\phi} & E \\ & {\scriptstyle f^*q}\searrow \quad \swarrow{\scriptstyle p} & \\ & B & \end{array} \qquad \begin{array}{ccc} G & \xrightarrow{\phi^T} & \Pi_f E \\ & {\scriptstyle q}\searrow \quad \swarrow{\scriptstyle \Pi_f\, p} & \\ & A & \end{array}$$

To gain some intuition into this adjunction, let's consider how it works on sets.

The right hand side operates in a coarsely grained fibration over the atlas A. It's a family of functions, one function per patch. For every patch we get a function from the "thick fiber" of G (drawn in blue below) to the "thick fiber" of $\Pi_f E$.

The left hand side operates in a more finely grained fibration over B. These fibers are grouped into small bundles over patches. Once we pick a patch (drawn in red below), we get a family of functions from that patch to the corresponding patch in E—a section of a small bundle in E. So, patch-by-patch, we get small sections of E.

The adjunction tells us that the elements of the "thick fiber" of $\Pi_f E$ correspond to small sections of E over the same patch.



The following exercises shed some light on the role played by f. It can be seen as localizing the sections of E by restricting them to "neighborhoods" defined by f^{-1}.

Exercise 11.4.1. *Consider what happens when A is a two-element set $\{0, 1\}$ and f maps the whole of B to one element, say 1. How would you define the function on the right-hand side of the adjunction? What should it do to the fiber over 0?*

Exercise 11.4.2. *Let's pick G to be a singleton set 1, and let $x\colon 1 \to A$ be a fibration that selects an element in A. Using the adjunction, show that:*

- *f^*1 has two types of fibers: singletons over the elements of $f^{-1}(x)$ and empty sets otherwise.*
- *A mapping $\phi\colon f^*1 \to E$ is equivalent to a selection of elements, one from each fiber of E over the elements of $f^{-1}(x)$. In other words, it's a partial section of E over the subset $f^{-1}(x)$ of B.*
- *A fiber of $\Pi_f E$ over a given x is such a partial section.*
- *What happens when A is also a singleton set?*

Universal quantification

The logical interpretation of the dependent product $\Pi_{x:B}\, T(x)$ is a universally quaniti-fied proposition. An element of $\Pi_{x:B}\, T(x)$ is a section—the proof that it's possible to select an element from each member of the family $T(x)$. It means that none of them is empty. In other words, it's a proof of the proposition:

$$\forall_{x:B}\, T(x)$$

11.5 Equality

Our first experience in mathematics involves equality. We learn that

$$1 + 1 = 2$$

and we don't think much of it afterwards.

But what does it mean that $1 + 1$ is equal to 2? Two is a number, but one plus one is an expression, so they are not the same thing. There is some mental processing that we have to perform before we pronounce these two things equal.

Contrast this with the statement $0 = 0$, in which both sides of equality are *the same thing*.

It makes sense that, if we are to define equality, we'll have to at least make sure that everything is equal to itself. We call this property *reflexivity*.

Recall our definition of natural numbers:

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

This is how we can define equality for natural numbers:

```
equal :: Nat -> Nat -> Bool
equal Z Z = True
equal (S m) (S n) = equal m n
equal _ _ = False
```

We are recursively stripping S's in each number until one of them reaches Z. If the other reaches Z at the same time, we pronounce the numbers we started with to be equal, otherwise they are not.

Equational reasoning

Notice that, when defining equality in Haskell, we were already using the equal sign. For instance, the equal sign in:

```
equal Z Z = True
```

tells us that wherever we see the expression `equal Z Z` we can replace it with `True` and vice versa.

This is the principle of substituting equals for equals, which is the basis for *equational reasoning* in Haskell. We can't encode proofs of equality directly in Haskell, but we can use equational reasoning to reason about Haskell programs. This is one of the main advantages of pure functional programming. You can't perform such substitutions in imperative languages, because of side effects.

If we want to prove that $1 + 1$ is 2, we have to first define addition. The definition can either be recursive in the first or in the second argument. This one recurses in the second argument:

```
add :: Nat -> Nat -> Nat
add n Z = n
add n (S m) = S (add n m)
```

We encode $1 + 1$ as:

```
add (S Z) (S Z)
```

We can now use the definition of `add` to simplify this expression. We try to match the first clause, and we fail, because `S Z` is not the same as `Z`. But the second clause matches. In it, `n` is an arbitrary number, so we can substitute `S Z` for it, and get:

```
add (S Z) (S Z) = S (add (S Z) Z)
```

In this expression we can perform another substitution of equals using the first clause of the definition of `add` (again, with `n` replaced by `S Z`):

```
add (S Z) Z = (S Z)
```

We arrive at:

```
add (S Z) (S Z) = S (S Z)
```

We can clearly see that the right-hand side is the encoding of 2. But we haven't shown that our definition of equality is reflexive so, in principle, we don't know if:

```
eq (S (S Z)) (S (S Z))
```

yields `True`. We have to use step-by-step equational reasoning again:

```
equal (S (S Z) (S (S Z)) =
{- second clause of the definition of equal -}
equal (S Z) (S Z) =
{- second clause of the definition of equal -}
equal Z Z =
{- first clause of the definition of equal -}
True
```

We can use this kind of reasoning to prove statements about concrete numbers, but we run into problems when reasoning about generic numbers—for instance, showing that something is true for all `n`. Using our definition of addition, we can easily show that `add n Z` is the same as `n`. But we can't prove that `add Z n` is the same as `n`. The latter proof requires the use of induction.

We end up distinguishing between two kinds of equality. One is proven using substitutions, or rewriting rules, and is called *definitional equality*. You can think of it as macro expansion or inline expansion in programming languages. It also involves β-reductions: performing function application by replacing formal parameters by actual arguments, as in:

```
(\x -> x + x) 2 =
{- beta reduction -}
2 + 2
```

The second more interesting kind of equality is called *propositional equality* and it may require actual proofs.

Equality vs isomorphism

We said that category theorists prefer isomorphism over equality—at least when it comes to objects. It is true that, within the confines of a category, there is no way to differentiate between isomorphic objects. In general, though, equality is stronger than isomorphism. This is a problem, because it's very convenient to be able to substitute equals for equals, but it's not always clear that one can substitute isomorphic for isomorphic.

Mathematicians have been struggling with this problem, mostly trying to modify the definition of isomorphism—but a real breakthrough came when they decided to simultaneously weaken the definition of equality. This led to the development of *homotopy type theory*, or HoTT for short.

Roughly speaking, in type theory, specifically in Martin-Löf theory of dependent types, equality is represented as a type, and in order to prove equality one has to construct an element of that type—in the spirit of the Curry-Howard interpretation.

Furthermore, in HoTT, the proofs themselves can be compared for equality, and so on ad infinitum. You can picture this by considering proofs of equality not as points but as some abstract paths that can be morphed into each other; hence the language of homotopies.

In this setting, instead of isomorphism, which involves strict equalities of arrows:

$$f \circ g = id$$

$$g \circ f = id$$

one defines an *equivalence*, in which these equalities are treated as types.

The main idea of HoTT is that one can impose the *univalence axiom* which, roughly speaking, states that equalities are equivalent to equivalences, or symbolically:

$$(A = B) \cong (A \cong B)$$

Notice that this is an axiom, not a theorem. We can either take it or leave it and the theory is still valid (at least we think so).

Equality types

Suppose that you want to compare two terms for equality. The first requirement is that both terms be of the same type. You can't compare apples with oranges. Don't get confused by some programming languages allowing comparisons of unlike terms: in every such case there is an implicit conversion involved, and the final equality is always between same-type values.

For every pair of values there is, in principle, a separate type of proofs of equality. There is a type for $0 = 0$, there is a type for $1 = 1$, and there is a type for $1 = 0$; the latter hopefully uninhabited.

Equality type, a.k.a., identity type, is therefore a dependent type: it depends on the two values that we are comparing. It's usually written as Id_A, where A is the type of both values, or using an infix notation as $x =_A y$ (equal sign with the subscript A).

For instance, the type of equality of two zeros is written as $Id_{\mathbb{N}}(0, 0)$ or:

$$0 =_{\mathbb{N}} 0$$

Notice: this is not a statement or a term. It's a *type*, like `Int` or `Bool`. You can define a value of this type if you have an introduction rule for it.

Introduction rule

The introduction rule for the equality type is the dependent function:

$$refl_A : \Pi_{x:A} Id_A(x, x)$$

which can be interpreted in the spirit of propositions as types as the proof of the statement:

$$\forall_{x:A}\ x = x$$

This is the familiar reflexivity: it shows that, for all x of type A, x is equal to itself. You can apply this function to some concrete value x of type A, and it will produce a new value of type $Id_A(x, x)$.

We can now prove that $0 = 0$. We can execute $refl_{\mathbb{N}}(0)$ to get a value of the type $0 =_{\mathbb{N}} 0$. This value is the proof that the type is inhabited, and therefore corresponds to a true proposition.

This is the only introduction rule for equality, so you might think that all proofs of equality boil down to "they are equal because they are the same." Surprisingly, this is not the case.

β-reduction and η-conversion

In type theory we have this interplay of introduction and elimination rules that essentially makes them the inverse of each other.

Consider the definition of a product. We introduce it by providing two values, $x : A$ and $y : B$ and we get a value $p : A \times B$. We can then eliminate it by extracting two values using two projections. But how do we know if these are the same values that we used to construct it? This is something that we have to postulate (in the categorical model this follows from the universal construction). We call it the computation rule or the β-reduction rule.

Conversely, if we are given a value $p : A \times B$, we can extract the two components using projections, and then use the introduction rule to recompose it. But how do we know that we'll get the same p? This too has to be postulated. This is sometimes called the uniqueness condition, or the η-conversion rule.

The equality type also has the elimination rule, which we'll discuss shortly, but we don't impose the uniqueness condition. It means that it's possible that there are some equality proofs that were not obtained using *refl*.

This is exactly the weakening of the notion of equality that makes HoTT interesting to mathematicians.

Induction principle for natural numbers

Before formulating the elimination rule for equality, it's instructive to first discuss a simpler elimination rule for natural numbers. We've already seen such rule describing primitive recursion. It allowed us to define recursive functions by specifying a value *init* and a function *step*.

Using dependent types, we can generalize this rule to define the *dependent elimination rule* that is equivalent to the principle of mathematical induction.

The principle of induction can be described as a device to prove, in one fell swoop, whole families of propositions indexed by natural numbers. For instance, the statement that `add Z n` is equal to `n` is really an infinite number of propositions, one per each value of `n`.

We could, in principle, write a program that would meticulously verify this statement for a very large number of cases, but we'd never be sure if it holds in general. There are some conjectures about natural numbers that have been tested this way using computers but, obviously, they can never exhaust an infinite set of cases.

Roughly speaking, we can divide all mathematical theorems into two groups: the ones that can be easily formulated and the ones whose formulation is complex. They can be further subdivided into the ones whose proofs are simple, and the ones that are hard or impossible to prove. For instance, the famous Fermat's Last Theorem was ex-

tremely easy to formulate, but its proof required some massively complex mathematical machinery.

Here, we are interested in theorems about natural numbers that are both easy to formulate and easy to prove. We'll assume that we know how to generate a family of propositions or, equivalently, a dependent type $T(n)$, where n is a natural number.

We'll also assume that we have a value:

$$\mathit{init}\colon T(Z)$$

or, equivalently, the proof of the zeroth proposition; and a dependent function:

$$\mathit{step}\colon \Pi_{n:\mathbb{N}}\ (T(n) \to T(Sn))$$

This function is interpreted as generating a proof of the $(n+1)$st proposition from the proof of the nth proposition.

The *dependent elimination rule* for natural numbers tells us that, given such *init* and *step*, there exists a dependent function:

$$f\colon \Pi_{n:\mathbb{N}}\, T(n)$$

This function is interpreted as providing the proof that $T(n)$ is true for all n.

Moreover, this function, when applied to zero reproduces *init*:

$$f(Z) = \mathit{init}$$

and, when applied to the successor of n, is consistent with taking a *step*:

$$f(Sn) = (\mathit{step}(n))(f(n))$$

(Here, $\mathit{step}(n)$ produces a function, which is then applied to the value $f(n)$.) These are the two *computation rules* for natural numbers.

Notice that the induction principle is not a theorem about natural numbers. It's part of the *definition* of the type of natural numbers.

Not all dependent mappings out of natural numbers can be decomposed into *init* and *step*, just as not all theorems about natural numbers can be proven inductively. There is no η-conversion rule for natural numbers.

Equality elimination rule

The elimination rule for equality type is somewhat analogous to the induction principle for natural numbers. There we used *init* to ground ourselves at the start of the journey, and *step* to make progress. The elimination rule for equality requires a more powerful grounding, but it doesn't have a *step*. There really is no good analogy for how it works, other than through a leap of faith.

The idea is that we want to construct a mapping *out* of the equality type. But since equality type is itself a two-parameter family of types, the mapping out should be a dependent function. The target of this function is another family of types:

$$T(x, y, p)$$

that depends on the pair of values that are being compared $x, y\colon A$ and the proof of equality $p\colon \mathit{Id}(x, y)$.

The function we are trying to construct is:

$$f\colon \Pi_{x,y:A}\Pi_{p:Id(x,y)}\, T(x,y,p)$$

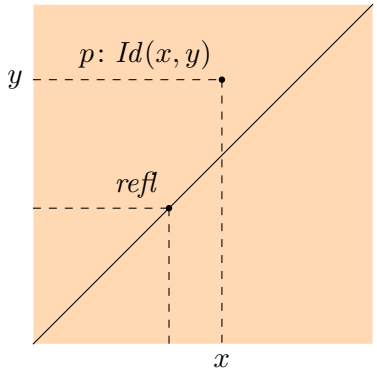
It's convenient to think of it as generating a proof that for all points x and y, and for every proof that the two are equal, the proposition $T(x,y,p)$ is true. Notice that, potentially, we have a different proposition for *every proof* that the two points are equal.

The least that we can demand from $T(x,y,p)$ is that it should be true when x and y are literally the same, and the equality proof is the obvious *refl*. This requirement can be expressed as a dependent function:

$$t\colon \Pi_{x:A}\, T\,(x,x,\mathit{refl}(x))$$

Notice that we are not even considering proofs of $x = x$, other than those given by reflexivity. Do such proofs exist? We don't know and we don't care.

So this is our grounding, the starting point of a journey that should lead us to defining our f for all pairs of points and all proofs of equality. The intuition is that we are defining f as a function on a plane (x,y), with a third dimension given by p. To do that, we're given something that's defined on the diagonal (x,x), with p restricted to *refl*.



You would think that we need something more, some kind of a *step* that would move us from one point to another. But, unlike with natural numbers, there is no *next* point or *next* equality proof to jump to. All we have at our disposal is the function t and nothing else.

Therefore we postulate that, given a type family $T(x,y,p)$ and a function:

$$t\colon \Pi_{x:A}\, T\,(x,x,\mathit{refl}(x))$$

there exists a function:

$$f\colon \Pi_{x,y:A}\Pi_{p:Id(x,y)}\, T(x,y,p)$$

such that (computation rule):

$$f(x,x,\mathit{refl}(x)) = t(x)$$

Notice that the equality in the computation rule is *definitional equality*, not a type.

Equality elimination tells us that it's alway possible to extend the function t, which is defined on the diagonal, to the whole 3-d space.

This is a very strong postulate. One way to understand it is to argue that, within the framework of type theory—which is formulated using the language of introduction and elimination rules, and the rules for manipulating those—it's *impossible* to define a type family $T(x, y, p)$ that would *not* satisfy the equality elimination rule.

The closest analogy that we've seen so far is the result of parametricity, which states that, in Haskell, all polymorphic functions between endofunctors are automatically natural transformations. Another example, this time from calculus, is that any analytic function defined on the real axis has a unique extension to the whole complex plane.

The use of dependent types blurs the boundary between programming and mathematics. There is a whole spectrum of languages, with Haskell barely dipping its toes in dependent types while still firmly established in commercial usage; all the way to theorem provers, which are helping mathematicians formalize mathematical proofs.

Chapter 12

Algebras

The essence of algebra is the formal manipulation of expressions. But what is an expression, and how do we manipulate it?

The first things to observe about algebraic expressions like $2(x + y)$ or $ax^2 + bx + c$ is that there are infinitely many of them. There is a finite number of rules for making them, but these rules can be used in infinitely many combinations. This suggests that the rules are used *recursively*.

In programming, expressions are virtually synonymous to (parsing) trees. Consider this simple example of an arithmetic expression:

```
data Expr = Val Int | Plus Expr Expr
```

It's a recipe for building trees. We start with little trees using the `Val` constructor. We then plant these seedlings into nodes, and so on.

```
e2 = Val 2
e3 = Val 3
e5 = Plus e2 e3
e7 = Plus e5 e2
```

Such recursive definitions work perfectly well in a programming language. The problem is that every new recursive data structure would require its own library of functions that operate on it.

From type-theory point of view, we've been able to define recursive types, such as natural numbers or lists, by providing, in each case, specific introduction and elimination rules. What we need is something more general, a procedure for generating arbitrary recursive types from simpler pluggable components.

There are two orthogonal concerns when it comes to recursive data structures. One is the machinery of recursion. The other is the pluggable components to be used by recursion.

We know how to work recursion: We assume that we know how to construct small trees. Then we use the recursive step to plant those trees into nodes to make bigger trees.

Category theory tells us how to formalize this imprecise description.

12.1 Algebras from Endofunctors

The idea of planting smaller trees into nodes requires that we formalize what it means to have a data structure with holes—a "container for stuff." This is exactly what functors are for. Because we want to use these functors recursively, they have to be *endo*-functors.

For instance, the endofunctor from our earlier example would be defined by the following data structure, where `x` marks the spots:

```
data ExprF x = ValF Int | PlusF x x
```

Information about all possible shapes of expressions is abstracted into a single functor.

The other important piece of information is the recipe for evaluating expressions. This, too, can be encoded using the same endofunctor.

Thinking recursively, let's assume that we know how to evaluate all subtrees of a larger expression. Then the remaining step is to plug these results into the top level node and evaluate it.

For instance, suppose that the `x`'s in the functor were replaced by integers—the results of evaluation of the subtrees. It's pretty obvious what we should do in the last step. If the top of the tree is a leaf `ValF` (which means there were no subtrees to evaluate) we'll just return the integer stored in it. If it's a `PlusF` node, we'll add the two integers in it. This recipe can be encoded as:

```
eval :: ExprF Int -> Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

We have made some seemingly obvious assumptions based on our experience. For instance, since the node was called `PlusF` we assumed that we should add the two numbers. But multiplication or subtraction would work equally well.

Since the leaf `ValF` contained an integer, we assumed that the expression should evaluate to an integer. But there is an equally plausible evaluator that pretty-prints the expression by converting it to a string, and using concatenation instead of addition:

```
pretty :: ExprF String -> String
pretty (ValF n)    = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

In fact there are infinitely many evaluators, some sensible, others less so, but we shouldn't be judgmental. Any choice of the target type and any choice of the evaluator should be equally valid. This leads to the following definition:

An *algebra* for an endofunctor F is a pair (A, α). The object A is called the *carrier* of the algebra, and the evaluator $\alpha\colon FA \to A$ is called the *structure map*.

In Haskell, given the functor `f` we define:

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

Notice that the evaluator is *not* a polymorphic function. It's a specific choice of a function for a specific type `a`. There may be many choices of the carrier types and there be many different evaluators for a given type. They all define separate algebras.

We have previously defined two algebras for `ExprF`. This one has `Int` as a carrier:

```
eval :: Algebra ExprF Int
eval (ValF n)   = n
eval (PlusF m n) = m + n
```

and this one has `String` as a carrier:

```
pretty :: Algebra ExprF String
pretty (ValF n)   = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

12.2 Category of Algebras

Algebras for a given endofunctor F form a category. An arrow in that category is an algebra morphism, which is a structure-preserving arrow between their carrier objects.

Preserving structure in this case means that the arrow must commute with the two structure maps. This is where functoriality comes into play. To switch from one structure map to another, we have to be able to lift an arrow that goes between their carriers.

Given an endofunctor F, an *algebra morphism* between two algebras (A, α) and (B, β) is an arrow $f : A \to B$ that makes this diagram commute:

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{f} & B \end{array}$$

In other words, the following equation must hold:

$$f \circ \alpha = \beta \circ Ff$$

The composition of two algebra morphisms is again an algebra morphism, which can be seen by pasting together two such diagrams. The identity arrow is also an algebra morphism, because

$$id_A \circ \alpha = \alpha \circ F(id_A)$$

(a functor maps identity to identity).

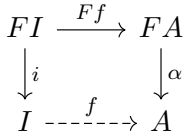
The commuting condition in the definition of an algebra morphism is very restrictive. Consider for instance a function that maps an integer to a string. In Haskell there is a `show` function (actually, a method of the `Show` class) that does it. It is *not* an algebra morphism from `eval` to `pretty`.

Exercise 12.2.1. *Show that `show` is not an algebra morphism. Hint: Consider what happens to a `PlusF` node.*

Initial algebra

The initial object in the category of algebras is called the *initial algebra* and, as we'll see, it plays a very important role.

By definition, the initial algebra (I, i) has a unique algebra morphism f from it to any other algebra (A, α). Diagrammatically:



This unique morphism is called a *catamorphism* for the algebra (A, α).

Exercise 12.2.2. *Let's define two algebras for the following functor:*

```
data FloatF x = Num Float | Op x x
```

The first algebra:

```
addAlg :: Algebra FloatF Float
addAlg (Num x) = log x
addAlg (Op x y) = x + y
```

The second algebra:

```
mulAlg :: Algebra FloatF Float
mulAlg (Num x) = x
mulAlg (Op x y) = x * y
```

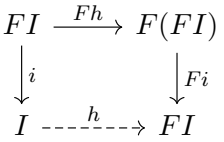
Make a convincing argument that `log` *(logarithm) is an algebra morphism between these two. (*`Float` *is a built-in floating-point number type.)*

12.3 Lambek's Lemma and Fixed Points

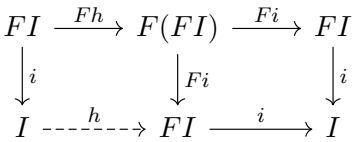
Lambek's lemma says that the structure map i of the initial algebra is an isomorphism.

The reason for it is the self-similarity of algebras. You can lift any algebra (A, α) using F, and the result $(FA, F\alpha)$ is also an algebra with the structure map $F\alpha \colon F(FA) \to FA$.

In particular, if you lift the initial algebra (I, i), you get an algebra with the carrier FI and the structure map $Fi \colon F(FI) \to FI$. It follows then that there must be a unique algebra morphism from the initial algebra to it:



This h is the inverse of i. To see that, let's consider the composition $i \circ h$. It is the arrow at the bottom of the following diagram



This is a pasting of the original diagram with a trivially commuting diagram. Therefore the whole rectangle commutes. We can interpret this as $i \circ h$ being an algebra morphism

from (I, i) to itself. But there already is such an algebra morphism—the identity. So, by uniqueness of the mapping out from the initial algebra, these two must be equal:

$$i \circ h = id_I$$

Knowing that, we can now go back to the previous diagram, which states that:

$$h \circ i = Fi \circ Fh$$

Since F is a functor, it maps composition to composition and identity to identity. Therefore the right hand side is equal to:

$$F(i \circ h) = F(id_I) = id_{FI}$$

We have thus shown that h is the inverse of i, which means that i is an isomorphism. In other words:

$$FI \cong I$$

We interpret this identity as stating that I is a fixed point of F (up to isomorphism). The action of F on I "doesn't change it."

There may be many fixed points, but this one is the *least fixed point* because there is an algebra morphism from it to any other fixed point. The least point of an endofunctor F is denoted μF, so we have:

$$I = \mu F$$

Fixed point in Haskell

Let's consider how the definition of the fixed point works with our original example:

```
data ExprF x = ValF Int | PlusF x x
```

Its fixed point is a data structure defined by the property that `ExprF` acting on it reproduces it. If we call this data structure `Expr`, the fixed point equation becomes (in pseudo-Haskell):

```
Expr = ExprF Expr
```

Expanding `ExprF` we get:

```
Expr = ValF Int | PlusF Expr Expr
```

Compare this with the recursive definition (actual Haskell):

```
data Expr = Val Int | Plus Expr Expr
```

We get a recursive data structure as a solution to the fixed-point equation.

In Haskell, we can define a fixed point data structure for any functor (or even just a type constructor). As we'll see later, this doesn't always give us the carrier of the initial algebra. It only works for those functors that have the "leaf" component.

Let's call `Fix f` the fixed point of a functor `f`. Symbolically, the fixed-point equation can be written as:

$$f(\text{Fix} f) \cong \text{Fix} f$$

or, in code,

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

The data constructor `In` is exactly the structure map of the initial algebra whose carrier is `Fix f`. Its inverse is:

```
out :: Fix f -> f (Fix f)
out (In x) = x
```

The Haskell standard library contains a more idiomatic definition:

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

To create terms of the type `Fix f` we often use "smart constructors." For instance, with the `ExprF` functor, we would define:

```
val :: Int -> Fix ExprF
val n = In (ValF n)

plus :: Fix ExprF -> Fix ExprF -> Fix ExprF
plus e1 e2 = In (PlusF e1 e2)
```

and use it to generate expression trees like this one:

```
e9 :: Fix ExprF
e9 = plus (plus (val 2) (val 3)) (val 4)
```

12.4 Catamorphisms

Our goal, as programmers, is to be able to perform a computation over a recursive data structure—to "fold" it. We now have all the ingredients.

The data structure is defined as a fixed point of a functor. An algebra for this functor defines the operation we want to perform. We've seen the fixed point and the algebra combined in the following diagram:

$$\begin{array}{ccc} FI & \xrightarrow{Ff} & FA \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle \alpha} \\ I & \xrightarrow{f} & A \end{array}$$

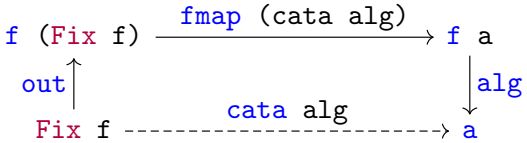
that defines the catamorphism f for the algebra (A, α).

The final piece of information is the Lambek's lemma, which tells us that i could be inverted because it's an isomorphism. It means that we can read this diagram as:

$$f = \alpha \circ Ff \circ i^{-1}$$

and interpret it as a recursive definition of f.

Let's redraw this diagram using Haskell notation. The catamorphism depends on the algebra so, for the algebra with the carrier `a` and the evaluator `alg`, we'll have the catamorphism `cata alg`.



By simply following the arrows, we get this recursive definition:

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

Here's what's happening: `cata alg` is applied to some `Fix f`. Every `Fix f` is obtained by applying `In` to a functorful of `Fix f`. The function `out` "strips" this data constructor.

We can evaluate the functorful of `Fix f` by `fmap`'ing `cata alg` over it. This is a recursive application. The idea is that the trees inside the functor are smaller than the original tree, so the recursion eventually terminates. It terminates when it hits the leaves.

After this step, we are left with a functorful of values, and we apply the evaluator `alg` to it, to get the final result.

The power of this approach is that all the recursion is encapsulated in one data type and one library function: We have the definition of `Fix` and the catamorphism. The client of the library has only to provide the *non-recursive* pieces: the functor and the algebra. These are much easier to deal with.

Examples

We can immediately apply this construction to our earlier examples. You can check that:

```
cata eval e9
```

evaluates to 9 and

```
cata pretty e9
```

evaluates to the string `"2 + 3 + 4"`.

Sometimes we want to display the tree on multiple lines with indentation. This requires passing a depth counter to recursive calls. There is a clever trick that uses a function type as a carrier:

```
pretty' :: Algebra ExprF (Int -> String)
pretty' (ValF n) i = indent i ++ show n
pretty' (PlusF f g) i = f (i + 1) ++ "\n" ++
                        indent i ++ "+" ++ "\n" ++
                        g (i + 1)
```

The auxiliary function `indent` replicates the space character:

```
indent n = replicate (n * 2) ' '
```

The result of:

```
cata pretty' e9 0
```

when printed, looks like this:

```
    2
  +
    3
+
  4
```

Let's try defining algebras for other familiar functors. The fixed point of the `Maybe` functor:

```
data Maybe x = Nothing | Just x
```

after some renaming, is equivalent to the type of natural numbers

```
data Nat = Z | S Nat
```

An algebra for this functor consists of a choice of the carrier `a` and an evaluator:

```
alg :: Maybe a -> a
```

The mapping out of `Maybe` is determined by two things: the value corresponding to `Nothing` and a function `a->a` corresponding to `Just`. In our discussion of the type of natural numbers we called these `init` and `step`. We can now see that the elimination rule for `Nat` is the catamorphism for this algebra.

The list type that we've seen previously is equivalent to a fixed point of the following functor:

```
data ListF a x = NilF | ConsF a x
```

An algebra for this functor is a mapping out

```
alg :: ListF a c -> c
alg NifF = init
alg (ConsF a c) = step (a, c)
```

which is determined by the value `init` and the function `step`:

```
init :: c
step :: (a, c) -> c
```

A catamorphism for such an algebra is the list recursor:

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
```

where `(List a)` is can be identified with the fixed point `Fix (ListF a)`.

We've seen before a recursive function that reverses a list. It was implemented by appending elements to the end of a list, which is very inefficient. It's easy to rewrite this function using a catamorphism, but the problem remains.

Prepending elements, on the other hand, is cheap. A better algorithm would traverse the list, accumulating elements in a last-in-first-out stack, and then pop them one-by-one and prepend them to a new list.

The stack regimen can be implemented by using composition of closures: each closure is a function that remembers its environment. Here's the algebra whose carrier is a function type:

```
revAlg :: Algebra (ListF a) ([a]->[a])
revAlg NilF = id
revAlg (ConsF a f) = \as -> f (a : as)
```

At each step, this algebra creates a function that will apply the previous function `f` to the result of prepending the current element `a` to its argument. It's this element that's remembered by the closure. The catamorphism for this algebra accumulates a stackful of such closures. To reverse a list, we apply the result of the catamorphism for this algebra to an empty list:

```
reverse :: Fix (ListF a) -> [a]
reverse as = (cata revAlg as) []
```

This trick is at the core of the fold-left funcion, `foldl`. Care should be taken when using it, because of the danger of stack overflow.

Lists are so common that their eliminators (called "folds") are included in the standard library. But there are infinitely many possible recursive data structures, each generated by its own functor, and we can use the same catamorphism on all of them.

12.5 Initial Algebra from Universality

Another way of looking at the initial algebra, at least in **Set**, is to view it as a collection of catamorphisms that, as a whole, hint at the existence of an underlying object. Instead of seeing μF as a set of trees, we can look at it as a set of functions from algebras to their carriers.

In a way, this is just another manifestation of the Yoneda lemma: every data structure can be described either by mappings in or mappings out. The mappings in, in this case, are the constructors of the recursive data structure. The mappings out are all the catamorphisms that can be applied to it.

First, let's make the polymorphism in the definition of `cata` explicit:

```
cata :: Functor f => forall a. Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

and then flip the arguments. We get:

```
cata' :: Functor f => Fix f -> forall a. Algebra f a -> a
cata' (In x) = \alg -> alg (fmap (flip cata' alg) x)
```

The function `flip` reverses the order of arguments to a function:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f b a = f a b
```

This gives us a mapping from `Fix f` to a set of polymorphic functions.

Conversely, given a polymorphic function of the type:

```
forall a. Algebra f a -> a
```

we can reconstruct `Fix f`:

```
uncata :: Functor f => (forall a. Algebra f a -> a) -> Fix f
uncata alga = alga In
```

In fact, these two functions, `cata'` and `uncata`, are the inverse of each other, establishing the isomorphism between `Fix f` and the type of polymorphic functions:

```
data Mu f = Mu (forall a. Algebra f a -> a)
```

Folding over `Mu f` is easy, since `Mu` carries in itself its own set of catamorphisms:

```
cataMu :: Algebra f a -> (Mu f -> a)
cataMu alg (Mu h) = h alg
```

You might be wondering how one can construct terms of the type `Mu f` for, let's say lists. It can be done using recursion:

```
fromList :: forall a. [a] -> Mu (ListF a)
fromList as = Mu h
  where h :: forall x. Algebra (ListF a) x -> x
        h alg = go as
          where
            go [] = alg NilF
            go (n: ns) = alg (ConsF n (go ns))
```

To compile this code you have to use the language pragma:

```
{-# language ScopedTypeVariables #-}
```

which puts the type variable `a` in the scope of the `where` clause.

Exercise 12.5.1. *Write a test that takes a list of integers, converts it to the `Mu` form, and calculates the sum using `cataMu`.*

12.6 Initial Algebra as a Colimit

In general, there is no guarantee that the initial object in the category of algebras exists. But if it exists, Lambek's lemma tells us that it's a fixed point of the endofunctor for these algebras. The construction of this fixed point is a little mysterious, since it involves tying the recursive knot.

Loosely speaking, the fixed point is reached after we apply the functor infinitely many times. Then, applying it once more wouldn't change anything. This idea can be made precise, if we take it one step at a time. For simplicity, let's consider algebras in the category of sets, which has all the nice properties.

We've seen, in our examples, that building instances of recursive data structures always starts with the leaves. The leaves are the parts in the definition of the functor that don't depend on the type parameter: the `NilF` of the list, the `ValF` of the tree, the `Nothing` of the `Maybe`, etc.

We can tease them out if we apply our functor F to the initial object—the empty set 0. Since the empty set has no elements, the instances of the type $F0$ are leaves only.

Indeed, the only inhabitant of the type `Maybe Void` is constructed using `Nothing`. The only inhabitants of the type `ExprF Void` are `ValF n`, where `n` is an `Int`.

In other words, $F0$ is the "type of leaves" for the functor F. Leaves are trees of depth one. For the `Maybe` functor there's only one—the type of leaves for this functor is a singleton:

```
m1 :: Maybe Void
m1 = Nothing
```

In the second iteration, we apply F to the leaves from the previous step and get trees of depth at most two. Their type is $F(F0)$.

For instance, these are all the terms of the type `Maybe(Maybe Void)`:

```
m2, m2' :: Maybe (Maybe Void)
m2 = Nothing
m2' = Just Nothing
```

We can continue this process, adding deeper and deeper trees at each step. In the n-th iteration, the type F^n0 (n-fold application of f to the initial object) describes all

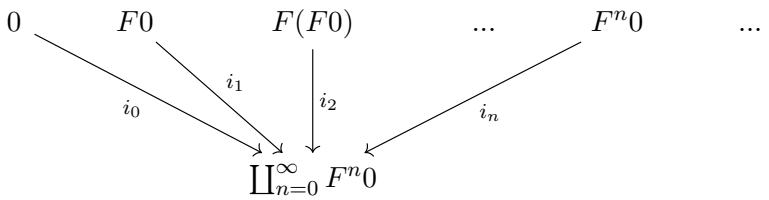
trees of depth up to n. However, for every n, there are still infinitely many trees of depth greater than n that are not covered.

If we knew how to define $F^\infty 0$, it would cover all possible trees. But the next best thing we could try is to gather all these partial trees into one infinite sum type. Just like we have defined sums of two types, we can define sums of many types, including infinitely many.

An infinite sum (a coproduct):

$$\coprod_{n=0}^{\infty} F^n 0$$

is just like a finite sum, except that it has infinitely many constructors i_n:



It has the universal mapping-out property, just like the sum of two types, only with infinitely many cases. (Obviously, we can't express it in Haskell.)

To construct a tree of depth n, we would first select it from $F^n 0$ and use the n-th constructor i_n to inject it into the sum.

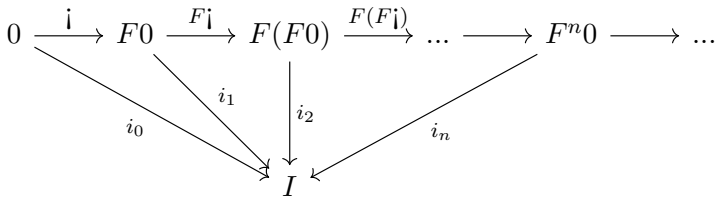
There is only one problem: the same tree shape can also be constructed using any of the $F^m 0$, with $m > n$.

Indeed, we've seen the leaf `Nothing` appear in `Maybe Void` and `Maybe(Maybe Void)`. In fact it shows up in any nonzero power of `Maybe` acting on `Void`.

Similarly, `Just Nothing` shows up in all powers starting with two.

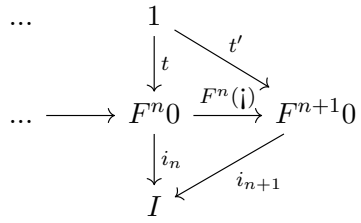
`Just(Just(Nothing))` shows up in all powers starting with three, and so on...

But there is a way to get rid of all these duplicates. The trick is to replace the sum by the colimit. Instead of a diagram consisting of discrete objects, we can construct a chain. Let's call this chain Γ, and its colimit $I = Colim\,\Gamma$.



It's almost the same as the sum, but with additional arrows at the base of the cocone. These arrows are the liftings of the unique arrow ¡ that goes from the initial object to $F0$ (we called it `absurd` in Haskell). The effect of these arrows is to collapse the set of infinitely many copies of the same tree down to just one representative.

To see that, consider for instance a tree of depth n. It can be first found as an element of $F^n 0$, that is to say, as an arrow $t\colon 1 \to F^n 0$. It is injected into the colimit I as the composite $i_n \circ t$.



The same shape of a tree is also found in $F^{n+1}0$, as the composite $t' = F^n(¡) \circ t$. It is injected into the colimit as the composite $i_{n+1} \circ t' = i_{n+1} \circ F^n(¡) \circ t$.

This time, however, we have the commuting triangle as the face of the cocone:

$$i_{n+1} \circ F^n(¡) = i_n$$

which means that

$$i_{n+1} \circ t' = i_{n+1} \circ F^n(¡) \circ t = i_n \circ t$$

The two copies of the tree have been identified in the colimit. You can convince yourself that this procedure removes all duplicates.

We can prove directly that $I = Colim\,\Gamma$ is the initial algebra. There is however one assumptions that we have to make: the functor F must preserve the colimit. The colimit of $F\Gamma$ must be equal to FI.

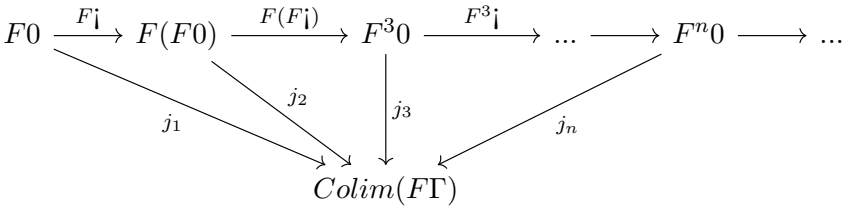
$$\mathrm{Colim}(F\Gamma) \cong FI$$

Fortunately, this assumption holds in **Set**.

Here's the sketch of the proof: First we'll construct an arrow $I \to FI$ and then an arrow in the opposite direction. We'll skip the proof that they are the inverse of each other.

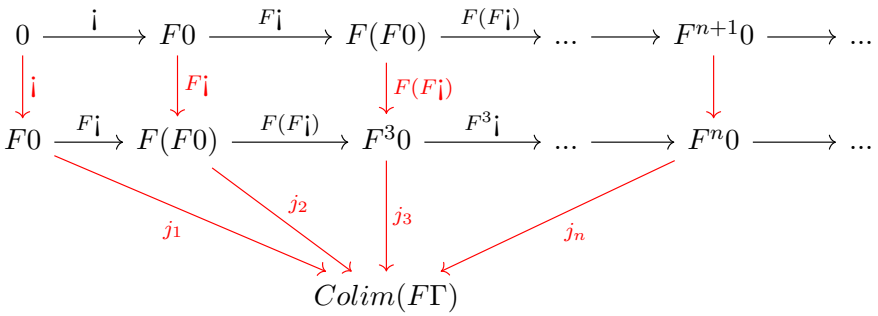
We start with the universality of the colimit. If we can construct a cocone from the chain Γ to $Colim(F\Gamma)$ then, by universality, there must be an arrow from I to $ColimF\Gamma$. And the latter, by our assumption, is isomorphic to FI. So we'll have a mapping $I \to FI$.

To construct this cocone, notice that $Colim(F\Gamma)$ is, by definition, the apex of a cocone $F\Gamma$.



The diagram $F\Gamma$ is the same as Γ, except that it's missing the naked initial object at the start of the chain.

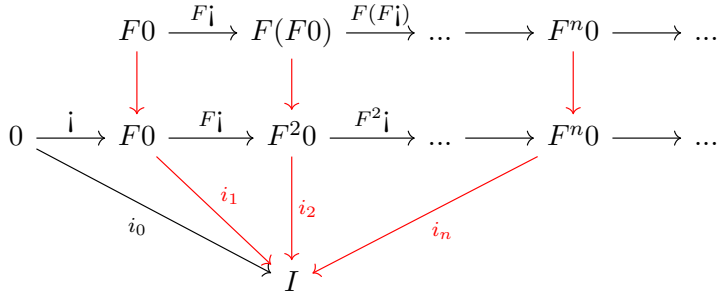
The spokes of the cocone we are looking for are marked in red in the diagram below:



And since $I = Colim\Gamma$ is the apex of the universal cocone based on Γ, there must be a unique mapping out of it to $Colim(F\Gamma)$, which is FI:

$$I \to FI$$

Next, notice that the chain $F\Gamma$ is a sub-chain of Γ, so it can be embedded in it. It means that we can construct a cocone from $F\Gamma$ to the apex I by going through (a sub-chain of) Γ.

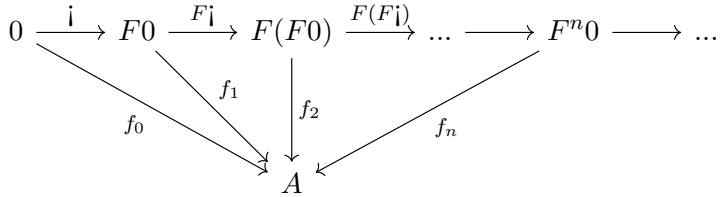


From the universality of $Colim(F\Gamma)$ it follows that there is a mapping out

$$Colim(F\Gamma) \cong FI \to I$$

This shows that I is a carrier of an algebra. In fact, it can be shown that the two mappings are the inverse of each other, as we would expect from the Lambek's lemma.

To show that this is indeed the initial algebra, we have to construct a mapping out of it to an arbitrary algebra $(A, \alpha\colon FA \to A)$. Again, we can use universality, if we can construct a cocone from Γ to A.



The zero'th spoke of this cocone goes from 0 to A, so it's just $f_0 = ¡$.
The first one, $F0 \to A$, is $f_1 = \alpha \circ Ff_0$, because $Ff_0\colon F0 \to FA$.
The third one, $F(F0) \to A$ is $f_2 = \alpha \circ Ff_1$. And so on...

The unique mapping from I to A is then our catamorphism. With some more diagram chasing, it can be shown that it's indeed an algebra morphism.

Notice that this construction only works if we can "prime" the process by creating the leaves of the functor. If, on the other hand, $F0 \cong 0$, then there are no leaves, and all further iterations will just reproduce the 0.

Chapter 13

Coalgebras

Coalgebras are just algebras in the opposite category. End of chapter!

Well, maybe not... As we've seen before, the category in which we're working is not symmetric with respect to duality. In particular, if we compare the terminal and the initial objects, their properties are not symmetric. Our initial object has no incoming arrows, whereas the terminal one, besides having unique incoming arrows, has lots of outgoing arrows.

Since initial algebras were constructed starting from the initial object, we might expect terminal coalgebras—their duals, generated from the terminal object—not to be just their mirror images, but to add their own interesting twists.

We've seen that the main application of algebras was in processing recursive data structures: in folding them. Dually, the main application of coalgebras is in generating, or unfolding, of recursive, tree-like, data structures. The unfolding is done using an anamorphism.

We use catamorphisms to chop trees, we use anamorphisms to grow them.

We cannot produce information from nothing so, in general, both a catamorphism and an anamorphism reduce the amount of information that's contained in their input.

After you sum a list of integers, it's impossible to recover the original list.

By the same token, if you grow a recursive data structure using an anamorphism, the seed must contain all the information that ends up in the tree. The advantage is that the information is now stored in a form that's more convenient for further processing.

13.1 Coalgebras from Endofunctors

A coalgebra for an endofunctor F is a pair consisting of a carrier A and a structure map: an arrow $A \to FA$.

In Haskell, we define:

```
type Coalgebra f a = a -> f a
```

We often think of the carrier as the type of a seed from which we'll grow the data structure, be it a list or a tree.

For instance, here's a functor that can be used to create a binary tree, with integers stored at the nodes:

```
data TreeF x = LeafF | NodeF Int x x
  deriving (Show, Functor)
```

We don't even have to define the instance of `Functor` for it—the `deriving` clause tells the compiler to generate the canonical one for us (together with the `Show` instance to allow conversion to `String`, if we want to display it).

A coalgebra is a function that takes a seed of the carrier type and produces a functor-ful of new seeds. These new seeds can then be used to generate the subtrees, recursively.

Here's a coalgebra for the functor `TreeF` that takes a list of integers as a seed:

```
split :: Coalgebra TreeF [Int]
split [] = LeafF
split (n : ns) = NodeF n left right
  where
    (left, right) = partition (<= n) ns
```

If the seed is empty, it generates a leaf; otherwise it creates a new node. This node stores the head of the list and fills the node with two new seeds. The library function `partition` splits a list using a user-defined predicate, here `(<= n)`, less-than-or-equal to `n`. The result is a pair of lists: the first one satisfying the predicate; and the second, not.

You can convince yourself that a recursive application of this coalgebra creates a binary sorted tree. We'll use this coalgebra later to implement a sort.

13.2 Category of Coalgebras

By analogy with algebra morphisms, we can define coalgebra morphisms as the arrows between carriers that satisfy a commuting condition.

Given two coalgebras (A, α) and (B, β), the arrow $f\colon A \to B$ is a coalgebra morphism if the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ FA & \xrightarrow{Ff} & FB \end{array}$$

The interpretation is that it doesn't matter if we first map the carriers and then apply the coalgebra β, or first apply the coalgebra α and then apply the arrow to its contents, using the lifting Ff.

Coalgebra morphisms can be composed, and the identity arrow is automaticaly a coalgebra morphism. It's easy to see that coalgebras, just like algebras, form a category.

This time, however, we are interested in the terminal object in this category—a *terminal coalgebra*. If a terminal coalgebra (T, j) exists, it satisfies the dual of the Lambek's lemma.

Exercise 13.2.1. *Lambek's lemma: Show that the structure map j of the terminal coalgebra (T, j) is an isomorphism. Hint: The proof is dual to the one for the initial algebra.*

As a consequence of the Lambek's lemma, the carrier of the terminal algebra is a fixed point of the endofunctor in question.

$$FT \cong T$$

with j and j^{-1} serving as the witnesses of this isomorphism.

It also follows that (T, j^{-1}) is an algebra; just as (I, i^{-1}) is a coalgebra, assuming that (I, i) is the initial algebra.

We've seen before that the carrier of the initial algebra is a fixed point. In principle, there may be many fixed points for the same endofunctor. The initial algebra is the least fixed point and the terminal coalgebra the greatest fixed point.

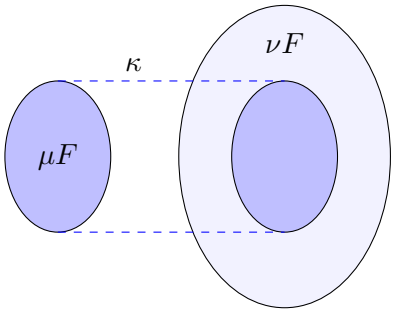
The greatest fixed point of an endofunctor F is denoted by νF, so we have:

$$T = \nu F$$

We can also see that there must be a unique algebra morphism (a catamorphism) from the initial algebra to the terminal coalgebra. That's because the terminal coalgebra is an algebra.

Similarly, there is a unique coalgebra morphism from the initial algebra (which is also a coalgebra) to the terminal coalgebra. In fact, it can be shown that it's the same underlying morphism $\kappa \colon \mu F \to \nu F$ in both cases.

In the category of sets, the carrier set of the initial algebra is a subset of the carrier set of the terminal coalgebra, with the function κ embedding the former in the latter.



We'll see later that in Haskell the situation is more subtle, because of lazy evaluation. But, at least for functors that have the leaf component—that is, their action on the initial object is non-trivial—Haskell's fixed point type works as a carrier for both the initial algebra and the terminal coalgebra.

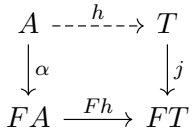
```
data Fix f where
  In :: f (Fix f) -> Fix f
```

Exercise 13.2.2. *Show that, for the identity functor in* **Set**, *every object is a fixed point, the empty set is the least fixed point, and the singleton set is the greatest fixed point. Hint: The least fixed point must have arrows going to all other fixed points, and the greatest fixed point must have arrows coming from all other fixed points.*

Exercise 13.2.3. *Show that the empty set is the carrier of the initial algebra for the identity functor in* **Set**. *Dually, show that the singleton set is this functor's terminal coalgebra. Hint: Show that the unique arrows are indeed (co-) algebra morphisms.*

13.3 Anamorphisms

The terminal coalgebra (T, j) is defined by its universal property: there is a unique coalgebra morphism h from any coalgebra (A, α) to (T, j). This morphism is called the *anamorphism*. Being a coalgebra morphism, it makes the following diagram commute:



Just like with algebras, we can use the Lambek's lemma to "solve" for `h`:

$$h = j^{-1} \circ Fh \circ \alpha$$

Since the terminal coalgebra (just like the initial algebra) is a fixed point of a functor, the above recursive formula can be translated directly to Haskell as:

```
ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coa = In . fmap (ana coa) . coa
```

Here's the interpretation of this formula: Given a seed of type `a`, we first act on it with the coalgebra `coa`. This gives us a functorful of seeds. We expand these seeds by recursively applying the anamorphism using `fmap`. We then apply the constructor `In` to get the final result.

As an example, we can apply the anamorphism to the `split` coalgebra we defined earlier: `ana split` takes a list of integers and creates a sorted tree.

We can then use a catamorphsims to fold this tree into a sorted list. We define an algebra:

```
toList :: Algebra TreeF [Int]
toList LeafF = []
toList (NodeF n ns ms) = ns ++ [n] ++ ms
```

that concatenates the left list with the singleton pivot and the right list. To sort a list we combine the anamorphism with the catamorphism:

```
qsort = cata toList . ana split
```

This gives us a (very inefficient) implementation of quicksort. We'll come back to it in the next section.

Infinite data structures

When studying algebras we relied on data structures that had a leaf component—or functors that, when acting on the initial object, would produce a result different from the initial object. When constructing recursive data structures we had to start somewhere, and that meant constructing the leaves first.

With coalgebras, we are free to drop this requirement. We no longer have to construct recursive data structures "by hand"—we have anamorphisms to do that for us. An endofunctor that has no leaves is perfectly acceptable: its coalgebras are going to generate infinite data structures.

Infinite data structures are representable in Haskell because of its laziness. Things are evaluated on the need-to-know basis. Only those parts of an infinite data structure that are explicitly demanded are calculated; the evaluation of the rest is kept in suspended animation.

To implement infinite data structures in strict languages, one must resort to representing values as functions—something Haskell does behind the scenes (these functions are called *thunks*).

Let's look at a simple example: an infinite stream of values. To generate it, we first define a functor that looks very much like the one we used to generate lists, except that it lacks the leaf component—the empty list constructor. You may recognize it as a product functor, with the first component fixed—it describes the stream's payload:

```
data StreamF a x = StreamF a x
  deriving Functor
```

An infinite stream is the fixed point of this functor.

```
type Stream a = Fix (StreamF a)
```

Here's a simple coalgebra that uses a single integer `n` as a seed:

```
step :: Coalgebra (StreamF Int) Int
step n = StreamF n (n+1)
```

It stores the seed as a payload, and seeds the next budding stream with `n + 1`.

The anamorphism for this coalgebra, seeded with zero, generates the stream of all natural numbers.

```
allNats :: Stream Int
allNats = ana step 0
```

In a non-lazy language this anamorphism would run forever, but in Haskell it's instantaneous. The incremental price is paid only when we want to retrive some of the data, for instance, using these accessors:

```
head :: Stream a -> a
head (In (StreamF a _)) = a

tail :: Stream a -> Stream a
tail (In (StreamF _ s)) = s
```

13.4 Hylomorphisms

The type of the output of an anamorphism is a fixed point of a functor, which is the same type as the input to a catamorphism. In Haskell, they are both described by the same data type, `Fix f`. Therefore it's possible to compose them together, as we've done when implementing quicksort. In fact, we can combine a coalgebra with an algebra in one recursive function called a *hylomorphism*:

```
hylo :: Functor f => Algebra f b -> Coalgebra f a -> a -> b
hylo alg coa = alg . fmap (hylo alg coa) . coa
```

We can rewrite quicksort as a hylomorphism:

```
qsort = hylo toList split
```

Notice that there is no trace of the fixed point in the definition of the hylomorphism. Conceptually, the coalgebra is used to build (unfold) the recursive data structure from the seed, and the algebra is used to fold it into a value of type `b`. But because of Haskell's laziness, the intermediate data structure doesn't have to be materialized in full in memory. This is particularly important when dealing with very large intermediate trees. Only the branches that are currently being traversed are evaluated and, as soon as they have been processed, they are passed to the garbage collector.

Hylomorphisms in Haskell are a convenient replacement for recursive backtracking algorithms, which are very hard to implement correctly in imperative languages. We take advantage of the fact that designing a data structure is easier than following complicated flow of control and keeping track of our place in a recursive algorithm.

This way, data structures can be used to visualize complex flows of control.

The impedance mismatch

We've seen that, in the category of sets, the initial algebras don't necessarily coincide with terminal coalgebras. The identity functor, for instance, has the empty set as the carrier of the initial algebra and the singleton set as the carrier of its terminal coalgebra.

We have other functors that have no leaf components, such as the stream functor. The initial algebra for such a functor is the empty set as well.

In **Set**, the initial algebra is the subset of the terminal coalgebra, and hylomorphisms can only be defined for this subset. It means that we can use a hylomorphism only if the anamorphism for a particular coalgebra lands us in this subset. In that case, because the embedding of initial algebras in terminal coalgebras is injective, we can find the corresponding element in the initial algebra and apply the catamorphism to it.

In Haskell, however, we have one type, `Fix f`, combining both, the initial algebra and the terminal coalgebra. This is where the simplistic interpretation of Haskell types as sets of values breaks down.

Let's consider this simple stream algebra:

```
add :: Algebra (StreamF Int) Int
add (StreamF n sum) = n + sum
```

Nothing prevents us from using a hylomorphism to calculate the sum of all natural numbers:

```
sumAllNats :: Int
sumAllNats = hylo add step 1
```

It's a perfectly well-formed Haskell program that passes the type checker. So what value does it produce when we run it? (Hint: It's not $-1/12$.) The answer is: we don't know, because this program never terminates. It runs into infinite recursion and eventually exhausts the computer's resources.

This is the aspect of real-life computations that mere functions between sets cannot model. Some computer function may never terminate.

Recursive functions are formally described by *domain theory* as limits of partially defined functions. If a function is not defined for a particular value of the argument, it is said to return a bottom value $\bot$. If we include bottoms as special elements of every type (these are then called *lifted* types), we can say that our function `sumAllNats` returns a

bottom of the type `Int`. In general, catamorphisms for infinite types don't terminate, so we can treat them as returning bottoms.

It should be noted, however, that the inclusion of bottoms complicates the categorical interpretation of Haskell. In particular, many of the universal constructions that rely on uniqueness of mappings no longer work as advertised.

The "bottom" line is that Haskell code should be treated as an illustration of categorical concepts rather than a source of rigorous proofs.

13.5 Terminal Coalgebra from Universality

The definition of an anamorphism can be seen as an expression of the universal property of the terminal coalgebra. Here it is, with the universal quantification made explicit:

```
ana :: Functor f => forall a. Coalgebra f a -> (a -> Fix f)
ana coa = In . fmap (ana coa) . coa
```

It says that, given any coalgebra, there is a mapping from its carrier to the carrier of the terminal coalgebra, `Fix f`. We know, from the Lambek's lemma, that this mapping is in fact a coalgebra morphism.

Let's uncurry this definition:

```
ana :: Functor f => forall a. (a -> f a, a) -> Fix f
ana (coa, x) = In (fmap (curry ana coa) (coa x))
```

In this form, we can take it as the definition of the carrier for the terminal coalgebra. We can replace `Fix f` with the type we are defining—let's call it `Nu f`. The type signature:

```
forall a. (a -> f a, a) -> Nu f
```

tells us that we can construct an element of `Nu f` from a pair `(a -> f a, a)`. It looks just like a data constructor, except that it's polymorphic in `a`.

Data types with a polymorphic constructor are called *existential* types. To construct an element of an existential type, we have the option of picking the most convenient type—the type for which we have the data required by the constructor.

For instance, we can construct a term of the type `Nu (StreamF Int)` by picking `Int` as the convenient type, and providing the pair:

```
nuArgs :: (Int -> StreamF Int Int, Int)
nuArgs =  (\n -> StreamF n (n+1) , 0)
```

The clients of an existential data type have no idea what type was used in its construction. All they know is that such a type *exists*—hence the name. If they want to use an existential type, they have to do it in a way that is not sensitive to the choice that was made in its construction. In practice, it means that an existential type must carry with itself both the producer and the consumer of the hidden value.

This is indeed the case in our example: the producer is just the value of type `a`, and the consumer is the function `a -> f a`.

Naively, all that the clients could do with this pair, without any knowledge of what the type `a` was, is to apply the function to the value. But if `f` is a functor, they can do much more. They can repeat the process by applying the lifted function to the contents of `f a`, and so on. They end up with all the information that's contained in the infinite stream.

There are several ways of defining existential data types in Haskell. We can use the uncurried version of the anamorphism directly as the data constructor:

```
data Nu f where
  Nu :: forall a f. (a -> f a, a) -> Nu f
```

Notice that, in Haskell, if we explicitly quantify one type, all other type variables must also be quantified: here, it's the type constructor `f` (however, `Nu f` is not existential in `f`, since it's an explicit parameter).

We can also omit the quantification altogether:

```
data Nu f where
  Nu :: (a -> f a, a) -> Nu f
```

This is because type variables that are not arguments to the type constructor are automatically treated as existentials.

We can also use the more traditional form:

```
data Nu f = forall a. Nu (a -> f a, a)
```

(This one requires the quantification of `a`.)

Finally, although this syntax is not allowed in Haskell, it's often more intuitive to write it like this:

```
data Nu f = Nu (exists a. (a -> f a, a))
```

(Later we'll see that existential data types correspond to coends in category theory.)

The constructor of `Nu f` is literally the (uncurried) anamorphism:

```
anaNu :: Coalgebra f a -> a -> Nu f
anaNu coa a = Nu (coa, a)
```

If we are given a stream in the form of `Nu (Stream a)`, we can access its element using accessors functions. This one extracts the first element:

```
head :: Nu (StreamF a) -> a
head (Nu (unf, s)) =
  let (StreamF a _) = unf s
  in a
```

and this one advances the stream:

```
tail :: Nu (StreamF a) -> Nu (StreamF a)
tail (Nu (unf, s)) =
  let (StreamF _ s') = unf s
  in Nu (unf, s')
```

You can test them on an infinite stream of integers:

```
allNats = Nu nuArgs
```

13.6 Terminal Coalgebra as a Limit

In category theory we are not afraid of infinitities—we make sense of them.

At face value, the idea that we could construct a terminal coalgebra by applying the functor F infinitely many times to some object, let's say the terminal object 1, makes no sense. But the idea is very convincing: Applying F one more time is like adding one

to infinity—it's still infinity. So, naively, this is a fixed point of F:

$$F(F^\infty 1) \cong F^{\infty+1} 1 \cong F^\infty 1$$

To turn this loose reasoning into a rigorous proof, we have to tame the infinity, which means we have to define some kind of a limiting procedure.

As an example, let's consider the product functor:

$$F_A X = A \times X$$

Its terminal coalgebra is an infinite stream. We'll approximate it it by starting with the terminal object 1. The next step is:

$$F_A 1 = A \times 1 \cong A$$

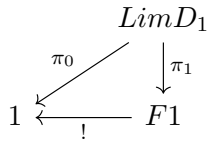
which we could imagine is a stream of length one. We can continue with:

$$F_A(F_A 1) = A \times (A \times 1) \cong A \times A$$

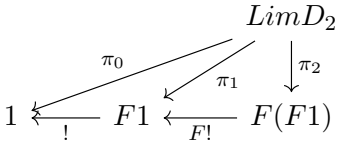
a stream of length two, and so on.

This looks promising, but what we need is one object that would combine all these approximations. We need a way to glue the next approximation to the previous one.

Recall, from an earlier exercise, the limit of the "walking arrow" diagram. This limit has the same elements as the starting object in the diagram. In particular, consider the limit in this diagram:

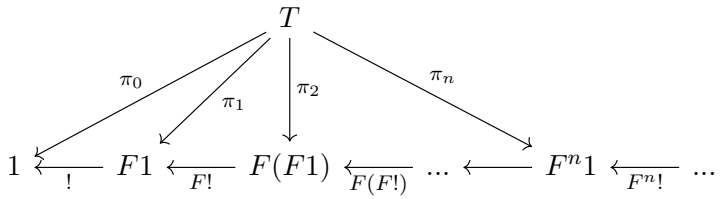


(! is the unique morphism targeting the terminal object 1). This limit has the same elements as $F1$. Similarly, this limit:



has the same elements as $F(F1)$.

We can continue extending this diagram to infinity. The limit of the infinite chain is the fixed point carrier of the terminal coalgebra.



The proof of this fact can be obtained from the analogous proof for initial algebras by reversing the arrows.

Chapter 14

Monads

What do a wheel, a clay pot, and a wooden house have in common? They are all useful because of the emptiness in their center.

Lao Tzu says: "The value comes from what is there, but the use comes from what is not there."

What does the `Maybe` functor, the list functor, and the reader functor have in common? They all have emptiness in their center.

When monads are explained in the context of programming, it's hard to see the common pattern when you focus on the functors. To understand monads you have to look inside functors and in between functions.

14.1 Programming with Side Effects

So far we've been talking about programming in terms of computations that were modeled mainly on functions between sets (with the exception of non-termination). In programming, such functions are called *total* and *pure*.

A total function is defined for all values of its arguments.

A pure function is implemented purely in terms of its arguments—it has no access to, much less having the ability to modify, its environment.

Most real-world programs, though, have to interact with the external world: they read and write files, process network packets, prompt users for data, etc. Most programming languages solve this problem by allowing side effect. A side effect is anything that breaks the totality or the purity of a function.

Unfortunately, this shotgun approach adopted by imperative languages makes reasoning about programs extremely hard. When composing effectful computations one has to carefully reason about the composition of effects, on a case-by-case basis. To make things even harder, most effects are hidden inside the implementation of a function and all the functions it's calling, recursively.

The solution adopted by purely functional languages, like Haskell, is to encode side effects in the return type of a pure function. Amazingly, this is possible for all relevant effects.

The idea is that, instead of a computation of the type `a->b` with side effects, we use a function `a -> f b`, where the type constructor `f` encodes the appropriate effect. At

this point there are no conditions imposed on `f`. It doesn't even have to be a `Functor`, much less a monad. This will come later, when we talk about effect composition.

Below is the list of common effects and their pure-function versions.

Partiality

In imperative languages, partiality is often encoded using exceptions. When a function is called with the "wrong" value for its argument, it throws an exception. In some languages, the type of exception is encoded in the signature of the function using special syntax.

In Haskell, a partial computation can be implemented by a function returning the result inside the `Maybe` functor. Such a function, when called with the "wrong" argument, returns `Nothing`, otherwise is wraps the result in the `Just` constructor.

If we want to encode more information about the type of the failure, we can use the `Either` functor, with the `Left` traditionally passing the error data (often a simple `String`); and `Right` encapsulating the real return, if available.

The caller of a `Maybe`-valued function cannot easily ignore the exceptional condition. In order to extract the value, they have to pattern-match the result and decide how to deal with `Nothing`. This is in contrast to the "poor-man's `Maybe`" of some imperative languages where the error condition is encoded using a null pointer.

Logging

Sometimes a computation has to log some values in some external data structure. Logging or auditing is a side effect that's particularly dangerous in concurrent programs, where multiple threads might try to access the same log simultaneously.

The simple solution is for a function to return the computed value paired with the item to be logged. In other words, a logging computation of the type `a -> b` can be replaced by a pure function:

```
a -> Writer w b
```

where the `Writer` functor is a thin encapsulation of the product:

```
newtype Writer w a = Writer (a, w)
```

with `w` being the type of the log.

The caller of this function is then responsible for extracting the value to be logged. This is a common trick: make the function provide all the data, and let the caller deal with the effects.

Environment

Some computations need read-only access to some external data stored in the environment. The read-only environment, instead of being secretly accessed by a computation, can be simply passed to a function as an additional argument. If we have a computation `a -> b` that needs access to some environment `e`, we replace it with a function `(a, e) -> b`. At first, this doesn't seem to fit the pattern of encoding side effects in the return type. However, such a function can always be curried to the form:

```
a -> (e -> b)
```

The return type of this function can be encoded in the reader functor, itself parameterized by the environment type `e`:

```
newtype Reader e a = Reader (e -> a)
```

This is an example of a delayed side effect. The function:

```
a -> Reader e a
```

doesn't want to deal with effects so it delegates the responsibility to the caller. You may think of it as producing a script to be executed at a later time. The function `runReader` plays the role of a very simple interpreter of this script:

```
runReader :: Reader e a -> e -> a
runReader (Reader h) e = h e
```

State

The most common side effect is related to accessing and potentially modifying some shared state. Unfortunately, shared state is the notorious source of concurrency errors. This is a serious problem in object oriented languages where stateful objects can be transparently shared between many clients. In Java, such objects may be provided with individual mutexes at the cost of impaired performance and the risk of deadlocks.

In functional programming we make state manipulations explicit: we pass the state as an additional argument and return the modified state paired with the return value. We replace a stateful computation `a -> b` with

```
(a, s) -> (b, s)
```

where `s` is the type of state. As before, we can curry such a function to get it to the form:

```
a -> (s -> (b, s))
```

This return type can be encapsulated in the following functor:

```
newtype State s a = State (s -> (a, s))
```

The caller of such a function is supposed to retrieve the result and the modified state by providing the initial state and calling the helper function, the interpreter, `runState`:

```
runState :: State s a -> s -> (a, s)
runState (State h) s = h s
```

Notice that, modulo constructor unpacking, `runState` is bona fide function application.

Nondeterminism

Imagine performing a quantum experiment that measures the spin of an electron. Half of the time the spin will be up, and half of the time it will be down. The result is non-deterministic. One way to describe it is to use the many-worlds interpretation: when we perform the experiment, the Universe splits into two universes, one for each result.

What does it mean for a function to be non-deterministic? It means that it will return different results every time it's called. We can model this behavior using the many-worlds interpretation: we let the function return *all possible results* at once. In practice, we'll settle for a (possibly infinite) list of results:

We replace a non-deterministic computation `a -> b` with a pure function returning a functor-ful of results—this time it's the list functor:

```
a -> [b]
```

Again, it's up to the caller to decide what to do with these results.

Input/Output

This is the trickiest side effect because it involves interacting with the external world. Obviously, we cannot model the whole world inside a computer program. So, in order to keep the program pure, the interaction has to happen outside of it. The trick is to let the program generate a script. This script is then passed to the runtime to be executed. The runtime is the effectful virtual machine that runs the program.

This script itself sits inside the opaque, predefined `IO` functor. The values hidden in this functor are not accessible to the program: there is no `runIO` function. Instead, the `IO` value produced by the program is executed, at least conceptually, *after* the program is finished.

In reality, because of Haskell's laziness, the execution of I/O is interleaved with the rest of the program. Pure functions that comprise the bulk of your program are evaluated on demand—the demand being driven by the execution of the `IO` script. If it weren't for I/O, nothing would ever be evaluated.

The `IO` object that is produced by a Haskell program is called `main` and its type signature is:

```
main :: IO ()
```

It's the `IO` functor containing the unit—meaning: there is no useful value other than the input/output script.

We'll talk about how `IO` actions are created soon.

Continuation

We've seen that, as a consequences of the Yoneda lemma, we can replace a value of type `a` with a function that takes a handler for that value. This handler is called a continuation. Calling a handler is considered a side effect of a computation. In terms of pure functions, we encode it as:

```
a -> Cont r b
```

where `Cont r` is the following functor:

```
newtype Cont r a = Cont ((a -> r) -> r)
```

It's the responsibility of the caller of this function to provide the continuation and retrieve the result:

```
runCont :: Cont r a -> (a -> r) -> r
runCont (Cont f) k = f k
```

This is the `Functor` instance for `Cont r`:

```
instance Functor (Cont r) where
  -- f :: a -> b
  -- k :: b -> r
  fmap f c = Cont (\k -> runCont c (k . f))
```

Notice that this is a covariant functor because the type `a` is in a doubly negative position.

14.2 Composing Effects

Now that we know how to make one giant leap using a function that produces both a value and a side effect, the next problem is to figure out how to decompose this leap into smaller human-sized steps. Or, conversely, how to combine two smaller steps into one larger step.

The way effectful computations are composed in imperative languages is to use regular function composition for the values and let the side effects combine themselves separately.

When we represent effectful computations as pure functions, we are faced with the problem of composing two functions of the form

```
g :: a -> f b
h :: b -> f c
```

In all cases of interest the type constructor `f` happens to be a `Functor`, so we'll assume that in what follows.

The naive approach would be to unpack the result of the first function, pass the value to the next function, then compose the effects of both functions on the side, and combine them with the result of the second function. This is not always possible, even for cases that we have studied so far, much less for an arbitrary type constructor.

It's instructive to see how we could do it for the `Maybe` functor. If the first function returns `Just`, we pattern match it to extract the contents and call the next function with it.

But if the first function returns `Nothing`, we have no value with which to call the second function. We have to short-circuit it, and return `Nothing` directly. So composition is possible, but it means modifying flow of control by skipping the second call based on the side effect of the first call.

For some functors the composition of side effects is possible, for others it's not. How can we characterize those "good" functors?

For a functor to encode composable side effects we must at least be able to implement the following polymorphic higher-order function:

```
composeWithEffects :: Functor f =>
       (b -> f c) -> (a -> f b) -> (a -> f c)
```

This is very similar to regular function composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

so it's natural to ask if there is a category in which the former defines a composition of arrows. Let's see what more is needed to construct such a category.

Objects in this new category are the same Haskell types as before. But an arrow from `a` to `b` is implemented as a Haskell function:

```
g :: a -> f b
```

Our `composeWithEffects` can then be used to implement the composition of such arrows.

To have a category, we require that this composition be associative. We also need an identity arrow for every object `a`. This is an arrow from `a` to `a`, so it corresponds to a Haskell function:

```
idWithEffects :: a -> f a
```

It must behave like identity with respect to `composeWithEffects`.

We have just defined a monad! After some renaming and rearranging, we can write it as a typeclass:

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

The infix operator `<=<` replaces the function `composeWithEffects`. The `return` function is the identity arrow in our new category. (This is not the definition of the monad you'll find in the Haskell's `Prelude` but, as we'll see soon, it's equivalent to it.)

As an exercise, let's define the `Monad` instance for `Maybe`. The "fish" operator `<=<` composes two functions:

```
f :: a -> Maybe b
g :: b -> Maybe c
```

into one function of the type `a -> Maybe c`. The unit of this composition, `return`, encloses a value in the `Just` constructor.

```
instance Monad Maybe where
  g <=< f = \a -> case f a of
                    Nothing -> Nothing
                    Just b -> g b
  return = Just
```

You can easily convince yourself that category laws are satisfied. In particular `return <=< g` is the same as `g` and `f <=< return` is the same as `f`. The proof of associativity is also pretty straightforward: If any of the functions returns `Nothing`, the result is `Nothing`; otherwise it's just a straightforward function composition, which is associative.

The category that we have just defined is called the *Kleisli category* for the monad `m`. The functions `a -> m b` are called the *Kleisli arrows*. They compose using `<=<` and the identity arrow is called `return`.

All functors from the previous section are `Monad` instances. If you look at them as type constructors, or even functors, it's hard to see any similarities between them. The thing they have in common is that they can be used to implement *composable* Kleisli arrows.

As Lao Tze would say: Composition is something that happens *between* things. While focusing our attention on things, we often lose sight of what's in the gaps.

14.3 Alternative Definitions

The definition of a monad using Kleisli arrows has the advantage that the monad laws are simply the associativity and the unit laws of a category. There are two other equivalent definitions of a monad, one preferred by mathematicians, and one by programmers.

First, let's notice that, when implementing the fish operator, we are given two functions as arguments. The only thing a function is useful for is to be applied to an argument. When we apply the first function `f :: a -> m b` we get a value of the type `m b`. At this point we would be stuck, if it weren't for the fact that `m` is a functor. Functoriality lets us apply the second function `g :: b -> m c` to `m b`. Indeed the lifting of `g` by `m` is of the type:

```
m b -> m (m c)
```

This is almost the result we are looking for, if we could only flatten `m(m c)` to `m c`. This flattening is called `join`. In other words, if we are given:

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

we can implement <=<:

```
g <=< f = \a -> join (fmap g (f a))
```

or, using point free notation:

```
g <=< f = join . fmap g . f
```

Conversely, `join` can be implemented in terms of <=<:

```
join = id <=< id
```

This may not be immediately obvious, until you realize that the rightmost `id` is applided to `m (m a)`, and the leftmost is applied to `m a`. We interpret a Haskell function:

```
m (m a) -> m (m a)
```

as an arrow in the Kleisli category that goes from `m(m a)` to `m a`. Similarly, the function:

```
m a -> m a
```

implements a Kleisli arrow from `m a` to `a`. Their Kleisli composition produces a Kleisli arrow from `m (m a)` to `a`, or a Haskell function:

```
m (m a) -> m a
```

This leads us to the equivalent definition of a monad in terms of `join` and `return`:

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

This is still not the definition you will find in the standard Haskell `Prelude`. Since the fish operator is a generalization of the dot operator, using it is equivalent to point-free programming. It lets us compose arrows without naming intermediate values. Although some consider point-free programs more elegant, most programmers find them difficult to follow.

But function composition is really done in two steps: We apply the first function, then apply the second function to the result. Explicitly naming the intermediate result is often helpful in understanding of what's going on.

To do the same with Kleisli arrows, we have to know how to apply the second Kleisli arrow to a named monadic value—the result of the first Kleisli arrow. The function that does that is called *bind* and is written as an infix operator:

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

Obviously, we can implement Kleisli composition in terms of bind:

```
g <=< f = \a -> (f a) >>= g
```

Conversely, bind can be implemented in terms of the Kleisli arrow:

```
ma >>= k = (k <=< id) ma
```

This leads us to the following definition:

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

This is almost the definition you'll find in the `Prelude`, except for the additional constraint. This constraint states the fact that every instance of `Monad` is also an instance of `Applicative`. We will postpone the discussion of applicatives to the section on monoidal functors.

We can also implement `join` using bind:

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

Here, `id` goes from `m a` to `m a` or, as a Kleisli arrow, from `m a` to `a`.

Interestingly, a `Monad` defined using bind is automatically a functor. The lifting function for it is called `liftM`

```
liftM :: Monad m => (a -> b) -> (m a -> m b)
liftM f ma = ma >>= (return . f)
```

14.4 Monad Instances

We are now ready to define monad instances for the functors we used to describe side effects. This will allow us to compose side effects.

Partiality

We've already seen the version of the `Maybe` monad implemented using Kleisli composition. Here's the more familiar implementation using bind:

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

Logging

In order to compose functions that produce logs, we need a way to combine individual log entries. This is why the writer monad:

```
newtype Writer w a = Writer (a, w)
```

requires the type of the log to be an instance of `Monoid`. This allows us to append logs and to create an empty log.

```
instance Monoid w => Monad (Writer w) where
  (Writer (a, w)) >>= k = let (Writer (b, w')) = k a
                          in Writer (b, mappend w w')
  return a = Writer (a, mempty)
```

The `let` clause is used for introducing local bindings. Here, the result of applying `k` is pattern matched, and the local variables `b` and `w'` are initialized. The `let`/`in` construct is an expression whose value is given by the content of the `in` clause.

Environment

The reader monad is a thin encapsulation of a function from the environment to the return type:

```
newtype Reader e a = Reader (e -> a)
```

Here's the `Monad` instance:

```
instance Monad (Reader e) where
  ma >>= k = Reader (\e -> let a = runReader ma e
                           in runReader (k a) e)
  return a = Reader (\e -> a)
```

The implementation of bind for the reader monad creates a function that takes the environment as its argument. This environment is used twice, first to run `ma` to get the value of `a`, and then to evaluate the value produced by `k a`.

The implementation of `return` ignores the environment.

Exercise 14.4.1. *Define the `Functor` and the `Monad` instance for the following data type:*

```
newtype E e a = E (e -> Maybe a)
```

Hint: You may use this handy function:

```
runE :: E e a -> e -> Maybe a
runE (E f) e = f e
```

State

Like reader, the state monad is a function type:

```
newtype State s a = State (s -> (a, s))
```

Its bind is similar, except that the result of `k` acting on `a` is now run with the modified state `s'`.

```
instance Monad (State s) where
  st >>= k = State (\s -> let (a, s') = runState st s
                          in runState (k a) s')

  return a = State (\s -> (a, s))
```

Applying bind to identity gives us the definition of `join`:

```
join :: State s (State s a) -> State s a
join mma = State (\s -> let (ma, s') = runState mma s
                        in runState ma s')
```

Notice that we are essentially passing the result of the first `runState` to the second `runState`, except that we have to uncurry the second one so it can accept a pair:

```
join mma = State (\s -> (uncurry runState) (runState mma s))
```

In this form, it's easy to convert it to point-free notation:

```
join mma = State (uncurry runState . runState mma)
```

There are two basic Kleisli arrows (the first one, conceptually, coming from the terminal object `()`) with which we can construct an arbitrary stateful computation. The first one retrieves the current state:

```
get :: State s s
get = State (\s -> (s, s))
```

and the second one modifies it:

```
set :: s -> State s ()
set s = State (\_ -> ((), s))
```

Nondeterminism

For the list monad, let's consider how we could implement `join`. It must turn a list of lists into a single list. This can be done by concatenating all the inner lists using the library function `concat`. From there, we can derive the implementation of bind.

```
instance Monad [] where
  as >>= k = concat (fmap k as)
  return a = [a]
```

`return` constructs a singleton list.

What in imperative languages do using nested loops we can do in Haskell using the list monad. Think of `as` as aggregating the result of running the inner loop and `k` as the code that runs in the outer loop.

In many ways, Haskell's list behaves more like what is called an iterator or a generator in imperative languages. Because of laziness, the elements of the list are rarely stored in memory all at once. You may also think of a list as a coroutine that produces, on demand, elements of a sequence.

Continuation

The implementation of bind for the continuation monad:

```
newtype Cont r a = Cont ((a -> r) -> r)
```

requires some backward thinking, because of the inherent inversion of control—the "don't call us, we'll call you" principle.

The result of bind is supposed to be a function that takes, as an argument, a continuation `k` of the type:

```
k :: b -> r
```

We have to put together such a function given the two ingredients at our disposal:

```
ma :: Cont r a
fk :: a -> Cont r b
```

We'd like to run `ma`, but for that we need a continuation that would accept an `a`. We can build such a continuation: given an `a` it would call `fk`. The result is a `b`-expecting continuation `Cont r b`. But that's exactly the continuation `k` we have at our disposal. Taken together, this convoluted process produces the following implementation:

```
instance Monad (Cont r) where
  ma >>= fk = Cont (\k -> runCont ma (\a -> runCont (fk a) k))
  return a = Cont (\k -> k a)
```

As we mentioned earlier, composing continuations is not for the faint of heart. However, it has to be implemented only once—in the definition of the continuation monad. From there on, the `do` notation will make the rest relatively easy.

Input/Output

The `IO` monad's implementation is baked into the language. The basic I/O primitives are available through the library. They are either in the form of Kleisli arrows, or `IO` objects (conceptually, Kleisli arrows from the terminal object `()`).

For instance, the following object contains a command to read a line from the standard input:

```
getLine :: IO String
```

There is no way to extract the string from it, since it's not there yet; but the program can process it through a further series of Kleisli arrows.

The `IO` monad is the ultimate procrastinator: the composition of its Kleisli arrows piles up task after task to be executed later by the Haskell runtime.

To output a string followed by a newline, you can use this Kleisli arrow:

```
putStrLn :: String -> IO ()
```

Combining the two, you may construct a simple `main` object:

```
main :: IO ()
main = getLine >>= putStrLn
```

which echos a string you type.

14.5 Do Notation

It's worth repeating that the sole purpose of monads in programming is to let us decompose one big Kleisli arrow into multiple smaller ones.

This can be either done directly, in a point-free style, using Kleisli composition `<=<`; or by naming intermediate values and binding them to Kleisli arrows using `>>=`.

Some Kleisli arrows are defined in libraries, others are reusable enough to warrant out-of-line implementation but, in practice, the majority are implemented as single-shot inline lambdas.

Here's a simple example:

```
main :: IO ()
main =
  getLine >>= \s1 ->
    getLine >>= \s2 ->
      putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

which uses an ad-hoc Kleisli arrow of the type `String->IO ()` defined by the lambda expression:

```
\s1 ->
    getLine >>= \s2 ->
      putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

The body of this lambda is further decomposed using another ad-hoc Kleisli arrow:

```
\s2 -> putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

Such constructs are so common that there is special syntax called the `do` notation that cuts through a lot of boilerplate. The above code, for instance, can be written as:

```
main = do
  s1 <- getLine
  s2 <- getLine
  putStrLn ("Hello " ++ s1 ++ " " ++ s2)
```

The compiler will automatically convert it to a series of nested lambdas. The line `s1<-getLine` is usually read as: "`s1` *gets* the result of `getLine`."

Here's another example: a function that uses the list monad to generate all possible pairs of elements taken from two lists.

```
pairs :: [a] -> [b] -> [(a, b)]
pairs as bs = do
  a <- as
  b <- bs
  return (a, b)
```

Notice that the last line in a `do` block must produce a monadic value—here this is accomplished using `return`.

As mentioned before, the `do` notation makes an easy task of otherwise very cumbersome composition of continuations.

Most imperative languages lack the abstraction power to generically define a monad and instead they attempt to hard-code some of the more common monads. For instance, they implement exceptions as an alternative to the `Either` monad, or concurrent tasks as an alternative to the continuation monad. Some, like C++, introduce coroutines that mimic Haskell's `do` notation.

Exercise 14.5.1. *Implement the following function that works for any monad:*

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

Hint: Use `do` notation to extract the function and the argument. Use `return` to return the result.

Exercise 14.5.2. *Rewrite the `pairs` function using the bind operators and lambdas.*

14.6 Monads Categorically

In category theory monads first arose in the study of algebras. In particular, we can use the bind operator to implement the very important operation of substitution.

Substitution

Consider this simple expression type. It's parameterized by the type `x` that we can use for naming variables:

```
data Ex x = Val Int
          | Var x
          | Plus (Ex x) (Ex x)
 deriving (Functor, Show)
```

We can, for instance, construct an expression $(2 + a) + b$:

```
ex :: Ex Char
ex = Plus (Plus (Val 2) (Var 'a')) (Var 'b')
```

We can implement the `Monad` instance for `Ex`:

```
instance Monad Ex where
  Val n >>= k = Val n
  Var x >>= k = k x
  Plus e1 e2 >>= k =
    let x = e1 >>= k
        y = e2 >>= k
    in (Plus x y)

  return x = Var x
```

Now suppose that you want to make a substitution by replacing a with $x_1 + 2$ and b with x_2 (for simplicity, let's not worry about other letters of the alphabet). This substitution is represented by the Kleisli arrow `sub` (as you can see, we are even able to change the type of variable names from `Char` to `String`):

```
sub :: Char -> Ex String
sub 'a' = Plus (Var "x1") (Val 2)
sub 'b' = Var "x2"
```

When we bind it to `ex`

```
ex' :: Ex String
ex' = ex >>= sub
```

we get, as expected, a tree corresponding to $(2 + (x_1 + 2)) + x_2$.

Monad as a monoid

Let's analyze the definition of a monad that uses `join`:

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

We have an endofunctor `m` and two polymorphic functions.

In category theory, the functor that defines the monad is traditionally denoted by T (probably because monads were initially called "triples"). The two polymorphic functions become natural transformations. The first one, corresponding to `join`, maps the "square" of T—or a composition of T with itself—to T:

$$\mu \colon T \circ T \to T$$

(Of course, only *endo*-functors can be squared this way.)

The second, corresponding to `return`, maps the identity functor to T:

$$\eta \colon Id \to T$$

Compare this with our earlier definition of a monoid in a monoidal category:

$$\mu \colon M \otimes M \to M$$
$$\eta \colon I \to M$$

The similarity is striking. This is why we often call the natural transformation μ *monadic multiplication*. But in what category can the composition of functors be considered a tensor product?

Enter the category of endofunctors. Objects in this category are endofunctors and arrows are natural transformations.

But there's more structure to that category. Any two endofunctors can be composed, so maybe composition could be considered a tensor product? After all, the only condition we impose on a tensor product is that it's functorial in both arguments. That is, given a pair of arrows:

$$\alpha \colon T \to T'$$
$$\beta \colon S \to S'$$

we can lift it to the mapping of the tensor product:

$$\alpha \otimes \beta \colon T \otimes S \to T' \otimes S'$$

In the category of endofunctors, the arrows are natural transformations so, if we replace $\otimes$ with $\circ$, the lifting would be a mapping:

$$\alpha \circ \beta \colon T \circ T' \to S \circ S'$$

But this is just the horizontal composition of natural transformations (now you understand why it's denoted by a circle).

The unit object in this monoidal category is the identity endofunctor, and unit laws are satisfied "on the nose," meaning

$$\mathrm{Id} \circ T = T = T \circ \mathrm{Id}$$

We don't need any unitors. We don't need any associators either, since functor composition is automatically associative.

A monoidal category in which unit and associativity laws are equalities is called a *strict* monoidal category.

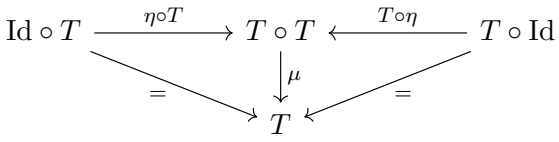
Notice, however, that composition is not symmetric, so this is not a symmetric monoidal category.

So, all said, a monad is a monoid in the monoidal category of endofunctors.

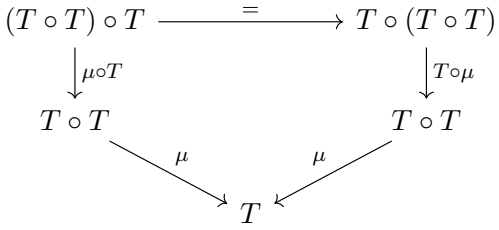
A monad (T, η, μ) consists of an object in the category of endofunctors—meaning an endofunctor T; and two arrows—meaning natural transformations:

$$\eta \colon \mathrm{Id} \to T$$
$$\mu \colon T \circ T \to T$$

For this to be a monoid, these arrows must satisfy monoidal laws. Here are the unit laws (with unitors replaced by strict equalities):



and this is the associativity law:



We used the whiskering notation for horizontal composition of $\mu \circ T$ and $T \circ \mu$.

These are the monad laws in terms of μ and η. They can be directly translated to the laws for `join` and `return`. They are also equivalent to the laws of the Kleisli category built from arrows $A \to TB$.

14.7 Monoidal Functors

We've seen several examples of monoidal cateogries. Such categories are equipped with some kind of binary operation, e.g., a cartesian product, a sum, composition (in the category of endofunctors), etc. They also have a special object that serves as a unit with respect to that binary operation. Unit and associativity laws are satisfied either on the nose (in strict monoidal categories) or up to isomorphism.

Every time we have more than one instance of some structure, we may ask ourselves the question: is there a whole category of such things? In this case: do monoidal categories form their own category? For this to work we would have to define arrows between monoidal categories.

A *monoidal functor* F from a monoidal category $(\mathcal{C}, \otimes, I)$ to another monoidal category $(\mathcal{D}, \oplus, J)$ maps tensor product to tensor product and unit to unit—all up to isomorphism:

$$FA \oplus FB \cong F(A \otimes B)$$
$$J \cong FI$$

Here, on the left-hand side we have the tensor product and the unit in the target category, and on the right their counterparts in the source category.

If the two monoidal categories in question are not strict, that is the unit and associativity laws are satisfied only up to isomorphism, there are additional coherency conditions that ensure that unitors are mapped to unitors and associators are mapped to associators.

The category of monoidal categories with monoidal functors as arrows is called **MonCat**. In fact it's a 2-category, since one can define structure-preserving natural transformations between monoidal functors.

Lax monoidal functors

One of the perks of monoidal categories is that they allow us to define monoids. You can easily convince yourself that monoidal functors map monoids to monoids. It turns out that you don't need the full power of monoidal functors to accomplish this task. Let's consider what the minimal requirements are for a functor to map a monoid to a monoid.

Let's start with a monoid (M, μ, η) in the monoidal category $(\mathcal{C}, \otimes, I)$. Consider a functor F that maps M to FM. We want FM to be a monoid in the target monoidal category $(\mathcal{D}, \oplus, J)$. For that we need to find two mappings:

$$\eta' \colon J \to FM$$
$$\mu' \colon FM \oplus FM \to FM$$

satisfying monoidal laws.

Since M is a monoid, we do have at our disposal the liftings of the original mappings:

$$F\eta \colon FI \to FM$$
$$F\mu \colon F(M \otimes M) \to FM$$

What we are missing, in order to implement η' and μ', are two additional arrows:

$$J \to FI$$
$$FM \oplus FM \to F(M \otimes M)$$

A monoidal functor would provide such arrows as halves of the defining isomorphisms. However, for what we're trying it accomplish, we don't need the other halves, so we can relax the isomorphisms.

A *lax monoidal functor* is a functor equipped with a one-way morphism and a natural transformation:

$$\phi_I \colon J \to FI$$
$$\phi_{AB} \colon FA \oplus FB \to F(A \otimes B)$$

satisfying the appropriate untiality and associativity conditions.

The simplest example of a lax monoidal functor is an endofunctor that (half-) preserves the usual cartesian product. We can define it in Haskell as a typeclass:

```
class Monoidal f where
  unit  :: f ()
  (>*<) :: f a -> f b -> f (a, b)
```

Corresponding to ϕ_{AB} we have an infix operator which, according to Haskell conventions, it's written in its curried form.

Exercise 14.7.1. *Implement the* `Monoidal` *instance for the list functor.*

Functorial strength

There is another way a functor may interact with monoidal structure, one that hides in plain sight when we do programming. We take it for granted that functions have access to the environment. Such functions are called closures.

For instance, here's a function that captures a variable `a` from the environment and pairs it with its argument:

```
\x -> (a, x)
```

This definition makes no sense in isolation, but it does when the environment contains the variable `a`, e.g.,

```
pairWith :: Int -> (String -> (Int, String))
pairWith a = \x -> (a, x)
```

The function returned by calling `pairWith 5` "encloses" the 5 from its environment.

Now consider the following modification, which returns a singleton list that contains the closure:

```
pairWith' :: Int -> [String -> (Int, String)]
pairWith' a = [\x -> (a, x)]
```

As a programmer you'd be very surprised if this didn't work. But what we do here is highly nontrivial: we are smuggling the environment inside the list functor. According to our model of lambda calculus, a closure is a morphism from the product of the environment and the function argument: here `(Int, String)`.

The property that lets us smuggle the environment under a functor is called *functorial strength* or *tensorial strength* and can be implemented in Haskell as:

```
strength :: Functor f => (e, f a) -> f (e, a)
strength (e, as) = fmap (e, ) as
```

The notation `(e, )` is called a *tuple section* and is equivalent to the partial application of the pair constructor: `(,) e`.

In category theory, strength for an endofunctor F is defined as a natural transformation:

$$\sigma \colon A \otimes F(B) \to F(A \otimes B)$$

There are some additional conditions which ensure that it works nicely with the unitors and the associator of the monoidal category in question.

The fact that we were able to implement `strength` for any functor means that, in Haskell, every functor is strong. This is the reason why we don't have to worry about accessing the environment from inside a functor.

Even more importantly, every monad in Haskell is strong by virtue of being a functor. This is also why every monad is automatically `Monoidal`.

```
instance Monad m => Monoidal m where
  unit = return ()
  ma >*< mb = do
    a <- ma
    b <- mb
    return (a, b)
```

(Warning: to compile this code you'll need to turn a few compiler extensions.) If you desugar this code to use monadic bind and lambdas, you'll notice that the final `return` needs access to both `a` and `b`, which are defined in outer environments. This would be impossible without a monad being strong.

In category theory, though, not every endofunctor in a monoidal category is strong. The reason it works in Haskell is that the category we're working with is cartesian closed. For now, the magic incantation is that such a category is self-enriched, so every endofunctor is canonically enriched. In Haskell this boils down to the fact that we can `fmap` a partially applied pair constructor `(a, )`.

Applicative functors

In programming, the idea of applicative functors arose from the following question: A functor lets us lift a function of one variable. How can we lift a function of two or more variables?

By analogy with `fmap`, we'd like to have a function:

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

A function of two arguments—here, in its curried form—is a function of one argument returning a function. So, assuming that `f` is a functor, we can `fmap` the first argument of `liftA2`:

```
a -> (b -> c)
```

over the second argument `(f a)` to get:

```
f (b -> c)
```

The problem is, we don't know how to apply `f (b -> c)` to the remaining argument `f b`.

The class of functors that let us do that is called `Applicative`. It turns out that, once we know how to lift a two-argument function, we can lift functions of any number of arguments, except zero. A zero-argument function is just a value, so lifting it means implementing a function:

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

Here's the Haskell definition:

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

The application of a functor-ful of functions to a functor-ful of arguments is defined as an infix operator that is customarily called "splat."

There is also an infix version of `fmap`:

```
(<$>) :: (a -> b) -> f a -> f b
```

which can be used in this terse implementation of `liftA2`:

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 g as bs = g <$> as <*> bs
```

Both operators bind to the left, which makes this syntax mimic regular function application.

An applicative functor must also satisfy a set of laws:

```
pure id <*> v = v                            -- Identity
pure f <*> pure x = pure (f x)               -- Homomorphism
u <*> pure y = pure ($ y) <*> u              -- Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- Composition
```

Exercise 14.7.2. *Implement `liftA3`, a function that lifts a 3-argument function using an applicative functor.*

Closed functors

If you squint at the definition of the splat operator:

```
(<*>) :: f (a -> b) -> (f a -> f b)
```

you may see it as mapping a function object to a function object.

This becomes clearer if you consider a functor between two categories, both of them closed. You may start with a function object B^A in the source category and apply the functor F to it:

$$F(B^A)$$

Alternatively, you may map the two objects A and B and construct a function object between them in the target category:

$$(FB)^{FA}$$

If we demand that the two ways be isomorphic, we get a strict *closed functor*. But, as was the case with monoidal functors, we are more interested in the lax version, which is equipped with a one-way natural transformation:

$$F(B^A) \to (FB)^{FA}$$

If F is an endofunctor, this translates directly into the definition of the splat operator.

The full definition of a lax closed functor includes the mapping of the monoidal unit and some coherence conditions.

In a closed monoidal category, the exponential is related to the cartesian product through an adjunction. It's no surprise then that, in such a category, lax monoidal and lax closed endofunctors are the same.

We can easily express this in Haskell:

```
instance (Functor f, Monoidal f) => Applicative f where
  pure a = fmap (const a) unit
  fs <*> as = fmap apply (fs >*< as)
```

where `const` is a function that ignores its second argument:

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

and `apply` is the uncurried function application:

```
apply :: (a -> b, a) -> b
apply (f, a) = f a
```

The other way around we have:

```
instance Applicative f => Monoidal f where
  unit = pure ()
  as >*< bs = (,) <$> as <*> bs
```

In the latter, we used the pair constructor `(,)` as a two-argument function.

Monads and applicatives

Since, in a cartesian closed category, every monad is lax monoidal, it is automatically applicative. We can show it directly by implementing `ap`, which has the same type signature as the splat operator:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap fs as = do
    f <- fs
    a <- as
    return (f a)
```

This connection is expressed in the Haskell definition of a `Monad` by having `Applicative` as its superclass:

```
class Applicative m => Monad m where
    (>>=)       :: forall a b. m a -> (a -> m b) -> m b
    return      :: a -> m a
    return      = pure
```

Notice the default implementation of `return` as `pure`.

The converse is not true: not every `Applicative` is a `Monad`. The standard example is the `Applicative` instance for a list functor that uses zipping:

```
instance Applicative [] where
  pure = repeat
  fs <*> as = zipWith apply fs as
```

Of course, the list functor is also a monad, so there is another `Applicative` instance based on that. Its splat operator applies every function to every argument.

In programming, monad is more powerful than applicative. That's because monadic code lets you examine the contents of a monadic value and branch depending on it. This is true even for the `IO` monad which otherwise provides no means of extracting the value, as in this example:

```
main :: IO ()
main = do
  s <- getLine
  if s == "yes"
```

```
  then putStrLn "Thank you!"
  else putStrLn "Next time."
```

Of course, the inspection of the value is postponed until the runtime interpreter of `IO` gets hold of this code.

Applicative composition using the splat operator doesn't allow for one part of the computation to inspect the result of the other. This a limitation that can be turned into an advantage. The absence of dependencies makes it possible to run the computations in parallel. Haskell's parallel libraries use applicative programming extensively.

On the other hand, monads let us use the very convenient `do` syntax, which is arguably more readable than the applicative syntax. Fortunately, there is a language extension `ApplicativeDo`, which instructs the compiler to selectively use applicative constructs in interpreting `do` blocks, whenever there are no dependencies.

Exercise 14.7.3. *Verify `Applicative` laws for the zip instance of the list functor.*

Chapter 15

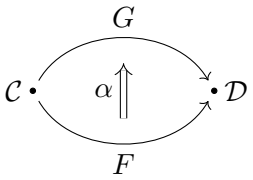
Monads from Adjunctions

15.1 String Diagrams

A line partitions a plane. We can think of it as either dividing a plane or as connecting two halves of the plane.

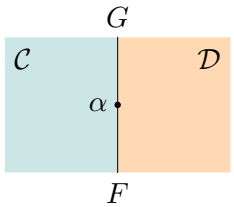
A dot partitions a line. We can think of it as either separating two half-lines or as joining them together.

This is a diagram in which two categories are represented as dots, two functors as arrows, and a natural transformation as a double arrow.



But the same idea can be represented by drawing categories as areas of a plane, functors that connect them as lines between areas, and natural transformations as dots that join line segments.

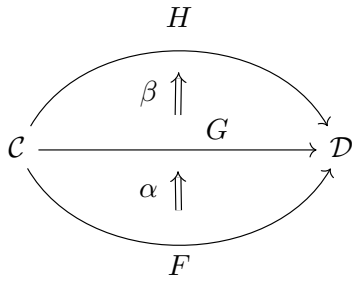
The idea is that a functor always goes between a pair of categories, therefore it can be drawn as a boundary between them. A natural transformation always goes between a pair of functors, therefore it can be drawn as a dot joining two halves of a line.



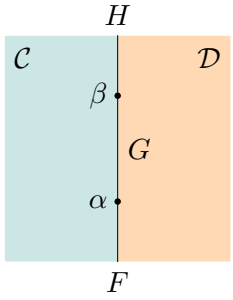
This is an example of a *string diagram*. You read such a digram bottom-up, left-to-right (think of the (x, y) system of coordinates).

The bottom of this diagram shows a functor F that goes from $\mathcal{C}$ to $\mathcal{D}$. The top of the diagram shows a functor G that goes between the same two categories. The transition happens in the middle, where a natural transformation α maps F to G.

So far it doesn't seem like we gain a lot by using this new visual representation. But let's apply it to something more interesting: vertical composition of natural transformations:

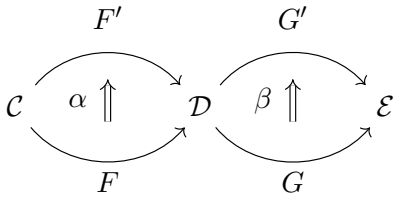


The corresponding string diagram shows the two categories and three functors between them joined by two natural transformations.



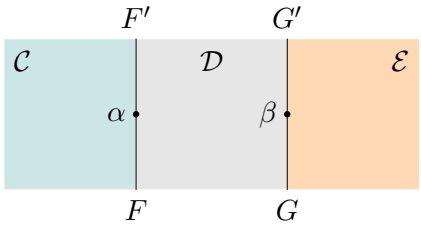
As you can see, you can reconstruct the original diagram from the string diagram by scanning it bottom-to-top.

Let's continue with the horizontal composition of natural transformations:



This time we have three categories, so we'll have three areas.

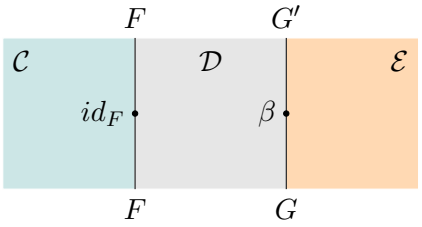
The bottom of the string diagram corresponds to the composition of functors $G \circ F$ (in this order). The top corresponds to $G' \circ F'$. One natural transformation, α, connects F to F'; the other, β, connects G to G'.



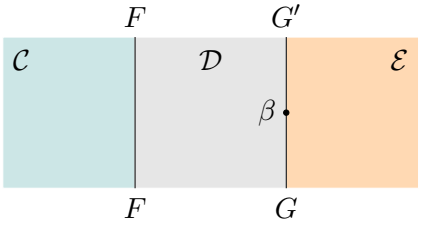
Drawing two parallel vertical lines, in this new system, corresponds to functor composition. You can think of the horizontal composition of natural transformations as

happening along the imaginary horizontal line in the middle of the diagram. But what if somebody was sloppy in drawing the diagram, and one of the dots was a little higher than the other? As it turns out, the exact positioning of the dots doesn't matter, due to the interchange law.

But first, let's illustrate whiskering: horizontal composition in which one of the natural transformation is the identity. We can draw it like this:

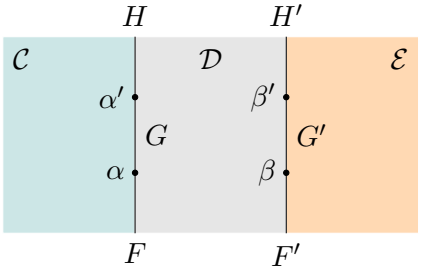


But, really, the identity can be inserted at any point on a vertical line, so we don't even have to draw it. The following diagram represents the whiskering of $\beta \circ F$.



Similarly, you can easily imagine the diagram for $\alpha \circ G$.

Here's the string diagram that corresponds to the interchange law:

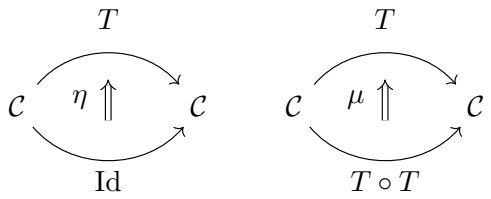


This diagram is purposefully ambiguous. Are we supposed to first do vertical composition of natural transformations and then the horizontal one? Or should we compose $\beta \circ \alpha$ and $\beta' \circ \alpha'$ horizontally, and then compose the results vertically? The interchange law says that it doesn't matter: the result is the same.

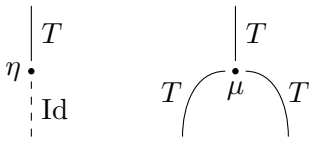
Now try to replace a pair of natural transformations in this diagram with identities. If you replace α' and β', you get the horizontal composition of $\beta \circ \alpha$. If you replace α' and β, and rename β' to β, you get the diagram in which α is shifted down with respect to β, and so on. The interchange law tells us that all these diagrams are equal. We are free to slide natural transformations like beads on a string.

String diagrams for the monad

A monad is defined as an endofunctor equipped with two natural transformations, as illustrated by the following diagrams:



Since we are dealing with just one category, when translating these diagrams to string diagrams, we can dispose of the naming (and shading) of categories, and just draw the strings alone.

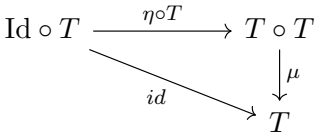


In the first diagram, it's customary to skip the dashed line corresponding to the identity functor. The η dot can be used to freely inject a T line into a diagram. Two T lines can be joined by the μ dot.

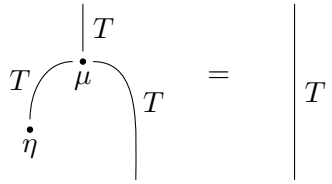
String diagrams are especially useful in expressing monad laws. For instance, we have the left identity law:

$$\mu \circ (\eta \circ T) = id$$

which can be visualized as a commuting diagram:

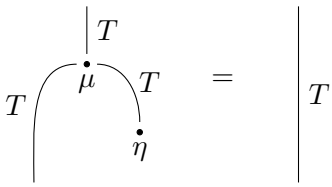


The corresponding string diagrams representat the equality of the two paths through this diagram:

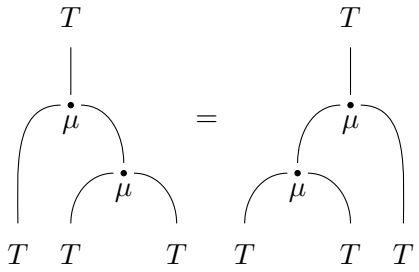


You can think of this equality as the result of pulling on the top and bottom strings resulting in the η appendage being retracted into the straight line.

There is a symmetric right identity law:



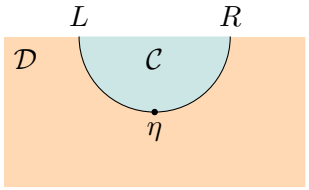
Finally, this is the associativity law in terms of string diagrams:



String diagrams for the adjunction

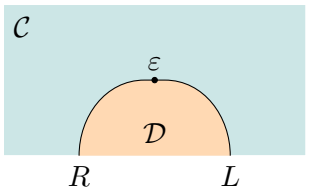
As we discussed before, an adjunction is a relation between a pair of functors, $L\colon \mathcal{D} \to \mathcal{C}$ and $R\colon \mathcal{C} \to \mathcal{D}$. It can be defined by a pair of natural transformations, the unit η and the counit ε, satisfying triangular identities.

The unit of the adjunction can be illustrated by a "cup"-shaped diagram:



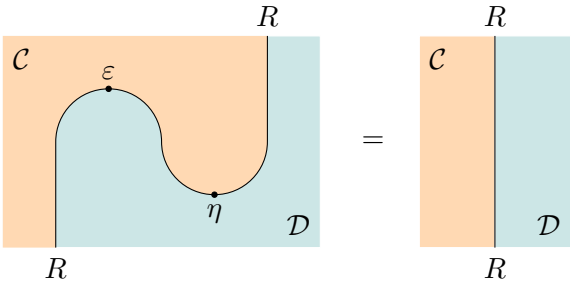
The identity functor at the bottom of the diagram is omitted from the picture. The η dot turns the identity functor below it to the composition $R \circ L$ above it.

Similarly, the counit can be visualized as a "cap"-shaped string diagram with the implicit identity functor at the top:



Triangle identities can be easily expressed using string diagrams. They also make intuitive sense, as you can imagine pulling on the string from both sides to straighten the curve.

For instance, this is the first triangle identity.



Reading the left diagram bottom-to-top produces a series of mappings:

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

This must be equal to the right-hand-side, which may be interpreted as the (invisible) identity natural transformation on R

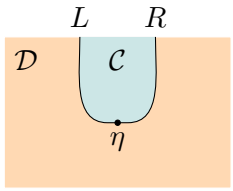
Exercise 15.1.1. *Draw the string diagrams for the second triangle identity.*

15.2 Monads from Adjunctions

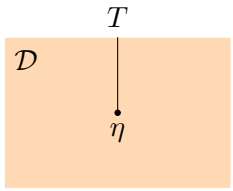
You might have noticed that the same symbol η is used for the unit of the adjunction and for the unit of the monad. This is *not* a coincidence.

At first sight it might seem like we are comparing apples to oranges: an adjunction is defined with two functors between two categories and a monad is defined by one endofunctor operating on a single category. However, the composition of the two functors going in opposite directions is an endofunctor, and the unit of the adjunction maps the identity endofunctor to the endofunctor $R \circ L$.

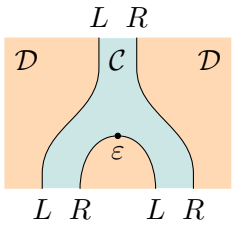
Compare this diagram:



with the one defining the monadic unit:



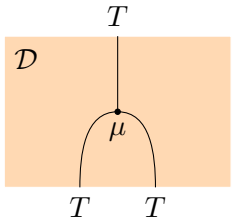
It turns out that $T = R \circ L$ is in fact a monad, with the multiplication μ defined by the following diagram:



Reading this diagram bottom-to-top, we get the following transformation (imagine slicing it horizontally at the dot):

$$R \circ L \circ R \circ L \xrightarrow{R \circ \varepsilon \circ L} R \circ L$$

Compare this with the definition of monadic μ:



This gives us the definition of μ for the monad $R \circ L$ as the double-whiskering of ε:

$$\mu = R \circ \varepsilon \circ L$$

To complete the picture, we can use string diagrams to derive monadic laws using triangle identities. The trick is to replace all strings in monadic laws by pairs of parallel strings and then rearrange them according to the rules.

To summarize, every adjunction $L \dashv R$ with the unit η and counit ε defines a monad $(R \circ L, \eta, R \circ \varepsilon \circ L)$.

We'll see later that, dually, the other composition, $L \circ R$ defines a comonad.

Exercise 15.2.1. *Draw string diagrams to illustrate monadic laws (unit and associativity) for the monad derived from an adjunction.*

15.3 Examples of Monads from Adjunctions

We'll go through several examples of adjunctions that generate some of the monads that we use in programming. Most examples involve functors that leave the category of Haskell types and functions, even though the round trip that generates the monad ends up being an endofunctor. This is why it's often impossible to express such an adjunction in Haskell.

To additionally complicate things, there is a lot of bookkeeping related to explicit naming of data constructors, which is necessary for type inference to work. This may sometimes obscure the simplicity on the underlying formulas.

Free monoid and the list monad

The list monad corresponds to the free monoid adjunction we've seen before. The unit of this adjunction, $\eta_X \colon X \to U(FX)$, injects the elements of the set X as the generators of the free monoid FX and U produces the underlying set. We represent the free monoid as a list, and its generators are singleton lists. The unit maps elements of X to such singletons:

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

To implement the counit, $\varepsilon_M \colon F(UM) \to M$, we take a monoid M, forget its multiplication, and use its set of elements as generators for the free monoid. The counit is a monoid morphism from that free monoid back to M. It turns out to be a special case of a catamorphism.

Recall the Haskell implementation of a list catamorphism:

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

We can interpret it as taking a function from `a` to the underlying set of a monoid `m` and producing a monoid morphism from the free monoid generated by `a` to `m`. We get the counit ε_M by applying it to the identity, `foldMap id`, or

```
epsilon = foldr mappend mempty
```

Monad multiplication is given by the whiskering of the counit:

$$\mu = U \circ \varepsilon \circ F$$

You can easily convince yourself that left whiskering doesn't do much here, since it's just a lifting of a monoid morphism by the forgetful functor. It keeps the function while forgetting about its special property of preserving structure.

The right whiskering by F is more interesting. It means taking the component of ε at a free monoid generated from the set X. This free monoid is defined by:

```
mempty = []
mappend = (++)
```

which gives us the definition of μ or `join`:

```
join = foldr (++) []
```

As expected, this is the same as `concat`.

The currying adjunction and the state monad

The state monad is generated by the adjunction between two functors that define the exponential object. The left functor is defined by a product with some fixed object S:

$$L_S A = A \times S$$

We can implement it as a Haskell type:

```
newtype L s a = L (a, s)
```

The right functor is the exponentiation, parameterized by the same object S:

$$R_S C = C^S$$

In Haskell, it's a thinly encapsulated function type:

```
newtype R s c = R (s -> c)
```

The monad is given by the composition of these two functors. On objects:

$$(R_S \circ L_S) A = (A \times S)^S$$

In Haskell we would write it as:

```
newtype St s a = St (R s (L s a))
```

If you expand this definition, it's easy to recognize in it the `State` functor:

```
newtype State s a = State (s -> (a, s))
```

The unit of the adjunction $L_S \dashv R_S$ is:

$$\eta_A \colon A \to (A \times S)^S$$

which can be implemented in Haskell as:

```
eta :: a -> R s (L s a)
eta a = R (\s -> L (a, s))
```

You may recognized in it a thinly veiled version of `return` for the state monad:

```
return :: a -> State s a
return a = State (\s -> (a, s))
```

Here's the component of the counit of this adjunction at C:

$$\varepsilon_C \colon C^S \times S \to C$$

It can be implemented in Haskell as:

```
epsilon :: L s (R s a) -> a
epsilon (L ((R f), s))= f s
```

which, after stripping data constructors, is equivalent to `apply`, or the uncurried version of `runState`.

Monad multiplication μ is given by the whiskering of ε from both sides:

$$\mu = R_S \circ \varepsilon \circ L_S$$

This is how it translates to Haskell:

```
mu :: R s (L s (R s (L s a))) -> R s (L s a)
mu = fmap epsilon
```

Here, whiskering on the right doesn't do anything other than select a component of the natural transformation. This is done automatically by Haskell's type inference engine.

Whiskering on the left is done by lifting the component of the natural transformation. Again, type inference picks the correct implementation of `fmap` (here, it's equivalent to precomposition) to perform that.

Compare this with the implementation of `join`:

```
join :: State s (State s a) -> State s a
join mma = State (fmap (uncurry runState) (runState mma))
```

Notice the dual use of `runState`:

```
runState :: State s a -> s -> (a, s)
runState (State h) s = h s
```

When it's uncurried, its type signature becomes:

```
uncurry runState :: (State s a , s) -> (a, s)
```

which is equivalent to that of `epsilon`. When partially applied, it just strips the data constructor exposing the underlying function type:

```
runState st ::  s -> (a, s)
```

M-sets and the writer monad

The writer monad:

```
newtype Writer m a = Writer (a, m)
```

is parameterized by a monoid `m`. This monoid is used for accumulating log entries. The adjunction we are going to use involves a category of M-sets for that monoid.

An M-set is a set S on which we define the action of a monoid M. Such an action is a mapping:

$$a \colon M \times S \to S$$

We often use the curried version of the action, with the monoid element in the subscript position. Thus a_m becomes a function $S \to S$.

This mapping has to satisfy some constraints. The action of the monoidal unit 1 must not change the set, so it has to be the identity function:

$$a_1 = id_S$$

and two consecutive actions must combine to an action of their monoidal product:

$$a_{m_1} \circ a_{m_2} = a_{m_1 \cdot m_2}$$

This choice of the order of multiplication defines what it called the *left action*. (The right action has the two monoidal elements swapped on the right-hand side.)

M-sets form a category **MSet**. The objects are pairs $(S, a \colon M \times S \to S)$ and the arrows are *equivariant maps*, that is functions between sets that preserve actions.

A function $f \colon S \to R$ is an *equivariant* mapping from (S, a) to (R, b) if the following diagram commutes, for every $m \in M$:

$$\begin{array}{ccc} S & \xrightarrow{f} & R \\ \downarrow{\scriptstyle a_m} & & \downarrow{\scriptstyle b_m} \\ S & \xrightarrow{f} & R \end{array}$$

In other words, it doesn't matter if we first do the action a_m, and then map the set; or first map the set, and then do the corresponding action b_m.

There is a forgetful functor U from **MSet** to **Set**, which assigns the set S to to the pair (S, a), thus forgetting the action.

Corresponding to it, there is a free functor F. Its action on a set S produces an M-set that is a cartesian product of S and M (where M is treated as a set of elements). An element of this M-set is a pair $(x \in S, m \in M)$ and the free action is defined by:

$$\phi_n \colon (x, m) \mapsto (x, n \cdot m)$$

To show that F is left adjoint to U we have to construct a natural isomorphism, for any set S and any M-set Q:

$$\mathbf{MSet}(FS, Q) \cong \mathbf{Set}(S, UQ)$$

The trick here is to notice that an equivariant mapping on the left is fully determined by its action on the elements of the form $(x, 1)$.

The unit of this adjunction $\eta_S \colon S \to U(FS)$ maps an element x to a pair $(x, 1)$. Compare this with the definition of `return` for the writer monad:

```
return a = Writer (a, mempty)
```

The counit is given by an equivariant map:

$$\varepsilon_Q \colon F(UQ) \to Q$$

The left hand side is an M-set constructed by taking the underlying set of Q and crossing it with M. The original action of Q is forgotten and replaced by the free action. The obvious choice for counit is:

$$\varepsilon_Q \colon (x, m) \mapsto a_m x$$

where a is the action defined in Q.

Monad multiplication μ is given by the whiskering of the counit.

$$\mu = U \circ \varepsilon \circ F$$

It means replacing Q in the definition of ε_Q with a free M-set whose action is the free action. (Whiskering with U doesn't change anything.)

$$\mu_S \colon ((x, m), n) \mapsto \phi_n(x, m) = (x, n \cdot m)$$

Compare this with the definition of `join` for the writer monad:

```
join :: Monoid m => Writer m (Writer m a) -> Writer m a
join (Writer ( Writer (x, m), m')) = Writer (x, mappend m' m)
```

Pointed objects and the `Maybe` monad

Pointed objects are objects with a designated element. Since picking an element is done using the arrow from the terminal object, the category of pointed objects is defined using pairs $(A, p \colon 1 \to A)$, where A is an object in $\mathcal{C}$.

The arrows between these pairs are the arrows in $\mathcal{C}$ that preserve the points. Thus an arrow from $(A, p \colon 1 \to A)$ to $(B, q \colon 1 \to B)$ is an arrow $f \colon A \to B$ such that $q = f \circ p$. This category is also called a *coslice category* and is written as $1/\mathcal{C}$.

There is an obvious forgetful functor $U \colon 1/\mathcal{C} \to \mathcal{C}$ that forgets the point. Its left adjoint is a free functor F that maps an object A to a pair $(1 + A, \text{Left})$. In other words, F freely ads a point to an object.

Exercise 15.3.1. *Show that $U \circ F$ is the `Maybe` monad.*

The `Either` monad is similarly constructed by replacing 1 with a fixed object E.

The continuation monad

The continuation monad is defined in terms of a pair of contravariant functors in the category of sets. We don't have to modify the definition of an adjunction to work with contravariant functors. It's enough to select the opposite category for one of the endpoints.

We'll define the left functor as:

$$L_R \colon \mathbf{Set}^{op} \to \mathbf{Set}$$

It maps a set X to the hom-set in **Set**:

$$L_R X = \mathbf{Set}(X, R)$$

This functor is parameterized by another set R. The right functor is defined by essentially the same formula:

$$L_R \colon \mathbf{Set} \to \mathbf{Set}^{op}$$

$$L_R X = \mathbf{Set}^{\mathbf{op}}(R, X) = \mathbf{Set}(X, R)$$

The composition $R \circ L$ can be written in Haskell as `((x -> r) -> r)`, which is the same as the (covariant) endofunctor that defines the continuation monad.

15.4 Monad Transformers

Suppose that you want to combine multiple effects, say, state with the possibility of failure. One option is to define your own monad from scratch. You define a functor:

```
newtype MaybeState s a = MS (s -> Maybe (a, s))
  deriving Functor
```

and the function to extract the result (or admit failure):

```
runMaybeState :: MaybeState s a -> s -> Maybe (a, s)
runMaybeState (MS h) s = h s
```

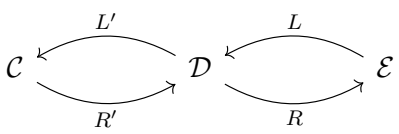
You define the monad instance for it:

```
instance Monad (MaybeState s) where
  return a = MS (\s -> Just (a, s))
  ms >>= k = MS (\s -> case runMaybeState ms s of
                         Nothing -> Nothing
                         Just (a, s') -> runMaybeState (k a) s')
```

and, if you are diligent enough, check that it satisfies the monad laws.

There is no general recipe for combining monads. In that sense, monads are not composable. However, we know that adjunctions are composable. We've seen how to get monads from adjunctions and, as we'll soon see, every monad can be obtained this way. So, if we can match the right adjunctions, the monads they generate will automatically compose.

Consider two composable adjunctions:



There are three monads in this picture. There is the inner monad $R' \circ L'$ and the outer monad $R \circ L$ as well as the composite $R \circ R' \circ L' \circ L$.

If we call the inner monad $T = R' \circ L'$, then $R \circ T \circ L$ is the composite monad called the *monad transformer*, because it transforms the monad T into a new monad.

In our example, `Maybe` is the inner monad:

$$TA = 1 + A$$

It is transformed using the outer adjunction $L_S \dashv R_S$, where:

$$L_S A = A \times S$$

$$R_S C = C^S$$

The result is:

$$(1 + A \times S)^S$$

or, in Haskell:

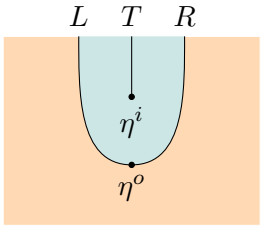
```
s -> Maybe (a, s)
```

In general, the inner monad T is defined by its unit η^i and multiplication μ^i. The outer adjunction is defined by its unit η^o and counit ε^o.

The unit of the composite monad is the natural transformation:

$$\eta \colon Id \to R \circ T \circ L$$

given by the string diagram:



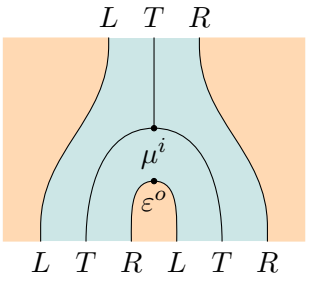
It is the vertical composition of the whiskered inner unit $R \circ \eta^i \circ L$ and the outer unit η^o. In components:

$$\eta_A = R(\eta^i_{LA}) \circ \eta^o_A$$

The multiplication of the composite monad is a natural transformation:

$$\mu \colon R \circ T \circ L \circ R \circ T \circ L \to R \circ T \circ L$$

given by the string diagram:



It's the vertical composition of the multiply whiskered outer counit:

$$R \circ T \circ L \circ \varepsilon^o \circ R \circ T \circ L$$

followed by the whiskered inner multiplication $R \circ \mu^i \circ L$. In components:

$$\mu_C = R(\mu^i_{LC}) \circ (R \circ T)(\varepsilon^o_{(T \circ L)C})$$

Let's unpack these equations for the case of the state monad transformer. Here, the left functor is the product functor `(a, s)` and the right functor is the exponential, a.k.a., the reader functor `s->a`.

and the counit ε^o_Y is function application, for the function `f :: x -> y`:

```
epsilon (f, x) = f x
```

We'll keep the inner monad (T, η^i, μ^i) arbitrary. In Haskell, we'll call these `m`, `return`, and `join`.

The monad that we get by applying the monad transformer for the state monad to the monad T, is the composition $R \circ T \circ L$ or, in Haskell:

```
newtype StateT s m a = StateT (s -> m (a, s))
```

```
runStateT :: StateT s m a -> s -> m (a, s)
runStateT (StateT h) s = h s
```

The unit of the monad transformer is the composition:

$$\eta_A = R(\eta^i_{LA}) \circ \eta^o_A$$

There are a lot of moving parts in this formula, so let's analyze it step-by-step. We start from the right: we have the A-component of the unit of the adjunction. It's an arrow from A to $R(LA)$. It's tranlated to a Haskell function. If we apply this function to a variable `x :: a`, we can pass the result to the next function in chain, which is $R(\eta^i_{LA})$.

Recall that the unit of the currying adjunction η^o_A is defined by:

```
eta x = \s -> (x, s)
```

so the result is again a function whose type is `s -> (a, s)`. Keep this in mind as we procede.

Now, what is $R(\eta^i_{LA})$? It's the component of `return` of the inner monad taken at LA. Here, LA is the type `(a, s)`. So we are instantiating `return` as a function `(a, s) -> m (a, s)`. (The type inferencer will do this automatically for us.)

Next, we are lifting this component of `return` using R. Here, R is the exponential $(-)^S$, so it lifts a function by post-composition. It will post-compose `return` to whatever function is passed to it. That's exactly the function that we produced using `eta`. Notice that the types match: we are post-composing `(a, s) -> m (a, s)` after `s -> (a, s)`.

We can write the result of this composition as:

```
return x = StateT (\s -> return (x, s))
```

with the data constructor `StateT` to make the type checker happy. This is the `return` of the composite monad in terms of the `return` of the inner monad.

The same reasoning can be applied to the formula for the component of the composite μ at some C:

$$\mu_C = R(\mu^i_{LC}) \circ (R \circ T)(\varepsilon^o_{(T \circ L)C})$$

The inner μ^i is the `join` of the monad `m`. Applying R turns it into post-composition.

The outer ε^o is function application taken at `m (a, s)` which, inserting the appropriate data constructors, can be written as `uncurry runStateT`:

```
uncurry runStateT :: (StateT s m a, s) -> m (a, s)
```

The application of $(R \circ T)$ lifts this component of ε using the composition of functors R and T. The former is implemented as post-composition, and the latter is the `fmap` of the monad m.

Putting all this together, we get a point-free formula for `join` of the state monad transformer:

```
join :: StateT s m (StateT s m a) -> StateT s m a
join mma = StateT (join . fmap (uncurry runStateT) . runStateT mma)
```

Here, the partially applied `runStateT mma` strips off the data constructor from the argument `mma`:

```
runStateT mma :: s -> m (a, x)
```

Our earlier example of `MaybeState` can now be rewritten using a monad transformer:

```
type MaybeState s a = StateT s Maybe a
```

The original `State` monad can be recovered by applying the `StateT` monad transformer to the identity functor, which has a `Monad` instance defined in the library (notice that the last type variable `a` is skipped in this definition):

```
type State s = StateT s Identity
```

Other monad transformers follow the same pattern. They are defined in the Monad Transformer Library, `MTL`.

15.5 Monad Algebras

Every adjunction generates a monad, and so far we've been able to define adjunctions for all the monads of interest for us. But is every monad generated by an adjunction? The answer is yes, and there are usually many adjunctions—in fact a whole category of adjunctions—for every monad.

Finding an adjunction for a monad is analogous to factorization. We want to express a functor as a composition of two other functors. The problem is complicated by the fact that this factorization also requires finding the appropriate intermediate category. We'll find such a category by studying algebras for a monad.

A monad is defined by an endofunctor, so it's possible to define algebras for this endofunctor. Mathematicians often think of monads as tools for generating expressions and algebras as tools for evaluating those expressions. However, expressions generated by monads impose some compatibility conditions on those algebras.

Consider the earlier example of the expression monad `Ex`. An algebra for this monad is a choice of the carrier type, say `Char` and an arrow

```
alg :: Ex Char -> Char
```

Since `Ex` is a monad, it defines a unit, or `return`, which is a polymorphic function that can be used to generate simple expressions from values. The unit of `Ex` is:

```
return x = Var x
```

We can instantiate the unit for an arbitrary type, in particular for the carrier type of our algebra. It makes sense that evaluating `Var c`, where `c` is an element of `Char` should give us back the same `c`. In other words, we'd like:

```
alg . return = id
```

This condition immediately eliminates a lot of algebras, such as:

```
alg (Var c) = 'a' -- not compatible with the monad Ex
```

The second condition we'd like to impose is that the algebra respects substitution. A monad lets us flatten nested expressions using `join`. An algebra lets us evaluate such expressions.

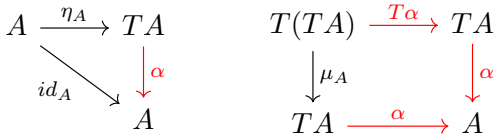
There are two ways of doing that: we can apply the algebra to a flattened expression, or we can apply it to the inner expression first (using `fmap`), and then evaluate the resulting expression.

```
alg (join mma) = alg (fmap alg mma)
```

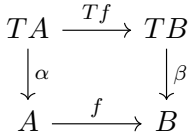
where `mma` is of the type `Ex (Ex Char)`.

In category theory these two conditions define a monad algebra.

$(A, \alpha\colon TA \to A)$ is a *monad algebra* for the monad (T, μ, η) if the following diagrams commute:



Since monad algebras are just special kinds of algebras, they form a sub-category of algebras. Recall that algebra morphisms are arrows that satisfy the following condition:



These laws are sometimes called the unit law and the multiplication law for monad algebras.

In light of this definition, we can re-interpret the second monad-algebra diagram as asserting that the structure map of a monad algebra is also an algebra morphism from (TA, μ_A) to (A, α). This will come in handy in what follows.

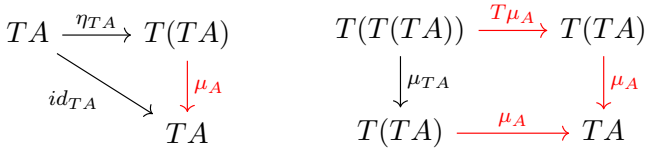
Eilenberg-Moore category

The category of monad algebras for a given monad T on $\mathcal{C}$ is called the Eilenberg-Moore category and is denoted by $\mathcal{C}^T$.

There is an obvious forgetful functor U^T from $\mathcal{C}^T$ to $\mathcal{C}$. It maps an algebra (A, α) to its carrier A, and treats algebra morphisms as regular morphisms between carriers.

There is a free functor F^T that is left adjoint to U^T. It's defined to map an object A of $\mathcal{C}$ to a monad algebra with the carrier TA. The structure map of this algebra is the component of monad multiplication $\mu_A\colon T(TA) \to TA$.

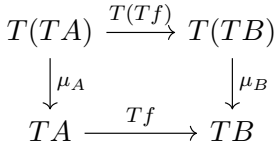
It's easy to check that (TA, μ_A) is indeed a monad algebra—the commuting conditions follow from monad laws. Indeed, substituting the algebra (TA, μ_A) into the monad-algebra diagrams, we get (with the algebra part drawn in red):



The first diagram is just the left monadic unit law in components. The η_{TA} arrow corresponds to the whiskering of $\eta \circ T$. The second diagram is the associativity of μ with the two whiskerings $\mu \circ T$ and $T \circ \mu$ expressed in components.

As is true for all adjunctions, the composition $U^T \circ F^T$ is a monad. However this particular monad is identical to the original monad T.

Indeed, on objects, it first maps A to a free monad algebra (TA, μ) and then forgets about the structure map. The net result is the mapping of A to TA, which is exactly what the original monad did. On arrows, it lifts an arrow $f\colon A \to B$ using T. The fact that the arrow Tf is an algebra morphism from (TA, μ_A) to (TB, μ_B) follows from naturality of μ:



To prove that we have an adjunction, we can either construct the natural isomorphism between hom-sets, or define two natural transformations to serve as the unit and the counit of the adjunction.

We define the unit of the adjunction as the monadic unit η of T. The counit is a natural transformation whose component at (A, α) is an algebra morphism from the free algebra generated by A, that is (TA, μ_A), back to (A, α). As we've seen earlier, α itself is such a morphism. We can therefore pick $\varepsilon_{(A,\alpha)} = \alpha$.

Triangular identities for these definitions of η and ε follow from unit laws for the monad and the monad algebra.

The whiskering of $U^T \circ \varepsilon F^T$ in components means instantiating ε at (TA, μ_A), which produces μ_A (the action of U^T on arrows is trivial).

We have thus shown that for any monad T we can define the Eilenberg-Moore category and a pair of adjoint functors that give rise to this monad.

Kleisli category

Inside every Eilenberg-Moore category there is a smaller Kleisli category struggling to get out. This smaller category is the image of the free functor we have constructed in the previous section.

Despite appearances, the image of a functor does not necessarily define a subcategory. Granted, it maps identities to identities and composition to composition. The problem may arise if two arrows that were not composable in the source category become composable in the target category. This may happen if the target of the first arrow is mapped to the same object as the source of the second arrow. However, the

free functor F^T maps distinct objects into distinct free algebras, so its image is indeed a subcategory of $\mathcal{C}^T$.

We have encountered the Kleisli category before. There are many ways of constructing the same category, and the simplest way to describe the Kleisli category is in terms of Kleisli arrows.

A Kleisli category for the monad (T, η, μ) is denoted by $\mathcal{C}_T$. Its objects are the same as the objects of $\mathcal{C}$, but an arrow in $\mathcal{C}_T$ from A to B is represented by an arrow in $\mathcal{C}$ that goes from A to TB. You may recognize it as the Kleisli arrow `a -> m b` we've defined before. Because T is a monad, these Kleisli arrows can be composed using the "fish" operator `<=<`.

To establish the adjunction, let's define the left functor $L_T \colon \mathcal{C} \to \mathcal{C}_T$ as identity on objects. We still have to define what it does to arrows. It must map a regular arrow $f \colon A \to B$ to a Kleisli arrow from A to B. This Kleisli arrow is represented by an arrow $A \to TB$ in $\mathcal{C}$. We pick the composite $\eta_B \circ f$:

$$L_T f \colon A \xrightarrow{f} B \xrightarrow{\eta_B} TB$$

The right functor $R_T \colon \mathcal{C}_T \to \mathcal{C}$ is defined on objects as a mapping that takes an A in the Kleisli category to an object TA in $\mathcal{C}$. Given a Kleisli arrow from A to B, which is represented by an arrow $g \colon A \to TB$, R_T will map it to an arrow $R_T A \to R_T B$, that is an arrow $TA \to TB$ in $\mathcal{C}$. We take this arrow to be $\mu_B \circ Tg$:

$$TA \xrightarrow{Tg} T(TB) \xrightarrow{\mu_B} TB$$

To establish the adjunction, we show the isomorphism of hom-sets:

$$\mathcal{C}_T(L_T A, B) \cong \mathcal{C}(A, R_T B)$$

An element of the left hand-side is a Kleisli arrow, which is represented by $f \colon A \to TB$. We can find the same arrow on the right hand side, since $R_T B$ is TB. So the isomorphism is between Kleisli arrows in $\mathcal{C}^T$ and the arrows in $\mathcal{C}$ that represent them.

The composite $R_T \circ L_T$ is equal to T and, indeed, it can be shown that this adjunction generates the original monad.

In general, there may be many adjunctions that generate the same monad. Adjunctions themselves form a 2-category, so it's possible to compare adjunctions using adjunction morphisms (1-cells in the 2-category). It turns out that the Kleisli adjunction is the initial object among all adjunctions that generate a given monad. Dually, the Eilenberg-Moore adjunction is terminal.

Chapter 16

Comonads

If it were easily pronounceable, we should probably call side effects "ntext," because the dual to side effects is "context."

Just like we were using Kleisli arrows to deal with side effects, we use co-Kleisli arrows to deal with contexts.

Let's start with the familiar example of an environment as a context. We have previously constructed a reader monad from it, by currying the arrow:

```
(a, e) -> b
```

This time, however, we'll treat it as a co-Kleisli arrow, which is an arrow from a "contextualized" argument.

As was the case with monads, we are interested in being able to compose such arrows. This is relatively easy for the environment-carrying arrows:

```
composeWithEnv :: ((b, e) -> c) -> ((a, e) -> b) -> ((a, e) -> c)
composeWithEnv g f = \(a, e) -> g (f (a, e), e)
```

It's also straightforward to implement an arrow that serves as an identity with respect to this composition:

```
idWithEnv :: (a, e) -> a
idWithEnv (a, e) = a
```

This shows that there is a category in which co-Kleisli arrows serve as morphisms.

16.1 Comonads in Programming

A functor `w` (consider it a stylized upside-down `m`) is a comonad if it supports composition of co-Kleisli arrows:

```
class Functor w => Comonad w where
   (=<=) :: (w b -> c) -> (w a -> b) -> (w a -> c)
   extract :: w a -> a
```

Here the composition is written in the form of an infix operator; and the unit of composition is called `extract`, since it extracts a value from the context.

Let's try it with our example. It's convenient to pass the environment as the first component of the pair. The comonad is then given by the functor that's a partial application of the pair constructor `((,) e)`.

```
instance Comonad ((,) e) where
  g =<= f = \ea -> g (fst ea, f ea)
  extract = snd
```

As with monads, co-Kleisli composition may be used in point-free style of programming. But we can also use the dual to `join` called `duplicate`:

```
  duplicate :: w a -> w (w a)
```

or the dual to bind called `extend`:

```
  extend :: (w a -> b) -> w a -> w b
```

Here's how we can implement co-Kleisli composition in terms of `duplicate` and `fmap`:

```
   g =<= f = g . fmap f . duplicate
```

Exercise 16.1.1. *Implement `duplicate` in terms of `extend` and vice versa.*

The `Stream` comonad

Interesting examples of comonads deal with larger, sometimes infinite, contexts. Here's an infinite stream:

```
data Stream a = Cons a (Stream a)
    deriving Functor
```

If we consider such a stream as a value of the type `a` in the context of the infinite tail, we can provide a `Comonad` instance for it:

```
instance Comonad Stream where
  extract (Cons a as) = a
  duplicate (Cons a as) = Cons (Cons a as) (duplicate as)
```

Here, `extract` returns the head of the stream and `duplicate` turns a stream into a stream of streams. Each consecutive stream is the tail of the previous one.

The intuition is that `duplicate` sets the stage for iteration, but it does it in a very general way. The head of each of the streams defines the "current position" in the stream.

It would be easy to perform a computation that goes over the head elements of these streams. But that's not where the power of a comonad lies. It lets us perform computations that require an arbitrary "look-ahead." Such a computation requires access not only to heads of consecutive streams, but to their tails as well.

This is what `extend` does: it applies a co-Kleisli arrow to all the streams generated by `duplicate`:

```
  extend f (Cons a as) = Cons (f (Cons a as)) (extend f as)
```

Here's an example of a co-Kleisli arrow that averages the first five elements of a stream:

```
avg :: Stream Double -> Double
avg  = (/5). sum . stmTake 5
```

It uses a helper function that extracts the first `n` items:

```
stmTake :: Int -> Stream a -> [a]
stmTake 0 _ = []
stmTake n (Cons a as) = a : stmTake (n - 1) as
```

We can run `avg` over the whole stream using `extend` to smooth local fluctuation. Electrical engineers might recognize this as a simple low-pass filter with `extend` implementing a convolution. It produces a running average of the original stream.

```
smooth :: Stream Double -> Stream Double
smooth = extend avg
```

Comonads are useful for structuring computations in spatially or temporally extended data structures. Such computations are local enough to define the "current location," but require gathering information from neighboring locations. Signal processing or image processing are good examples. So are simulations in which differential equations have to be iteratively solved inside volumes: climate simulations, cosmological models, or nuclear reactions come to mind. Conway's Game of Life is also a good testing ground for comonadic methods.

Sometimes it's convenient to perform calculation on continuous streams of data, postponing the sampling until the very last step. Here's an example of a signal that is a function of time (represented by `Double`)

```
data Signal a = Sig (Double -> a) Double
```

The second component is the current time.

This is the `Comonad` instance:

```
instance Comonad Signal where
  extract (Sig f x) = f x
  duplicate (Sig f x) = Sig (\y -> Sig f (x - y)) x
  extend g (Sig f x) = Sig (\y -> g (Sig f (x - y))) x
```

Here, `extend` convolves the filter

```
g :: Signal a -> a
```

over the whole stream.

Exercise 16.1.2. *Implement the `Comonad` instance for a bidirectional stream:*

```
data BiStream a = BStr [a] [a]
```

Assume that both list are infinite. Hint: Consider the first list as the past (in reverse order), the head of the second list as as the present, and its tail as the future.

Exercise 16.1.3. *Implement a low-pass filter for `BiStream` that averages over three values: the current one, one from the immediate past, and one from the immediate future. For electrical engineers: implement a Gaussian filter.*

16.2 Comonads Categorically

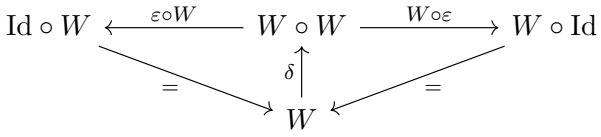
We can get the definition of a comonad by reversing the arrows in the definition of a monad. Our `duplicate` corresponds to the reversed `join`, and `extract` is the reversed `return`.

A comonad is thus an endofunctor W equipped with two natural transformations:

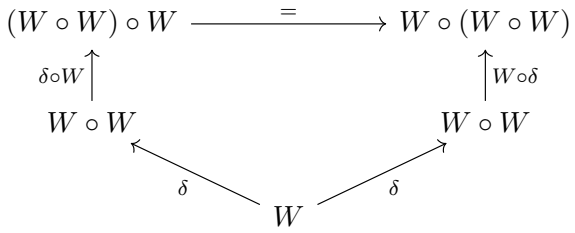
$$\delta \colon W \to W \circ W$$
$$\varepsilon \colon W \to Id$$

These transformations (corresponding to `duplicate` and `extract`, respectively) must satisfy the same identities as the monad, except with the arrows reversed.

These are the counit laws:



and this is the associativity law:



Comonoids

We've seen how monadic laws followed from monoid laws. We can expect that comonad laws should follow from a dual version of a monoid.

Indeed, a *comonoid* is an object in a monoidal category $(\mathcal{C}, \otimes, I)$ equipped with two morphisms called co-multiplication and a co-unit:

$$\delta \colon W \to W \otimes W$$
$$\varepsilon \colon W \to I$$

We can replace the tensor product with endofunctor composition and the unit object with the identity functor to get the definition of a comonad as a comonoid in the category of endofunctors.

In Haskell we can define a `Comonoid` typeclass for the cartesian product:

```
class Comonoid w where
  split   :: w -> (w, w)
  destroy :: w -> ()
```

Comonoids are less talked about than their siblings, monoids, mainly because they are taken for granted. In a cartesian category, every object can be made into a comonoid: just by using the diagonal mapping $\Delta_a \colon a \to a \times a$ for co-multiplication, and the unique arrow to the terminal object for counit.

In programming this is something we do without thinking. Co-multiplication means being able to duplicate a value, and counit means being able to abandon a value.

In Haskell, we can easily implement the `Comonoid` instance for any type:

```
instance Comonoid w where
  split w   = (w, w)
  destroy w = ()
```

In fact, we don't think twice of using the argument of a function twice, or not using it at all. But, if we wanted to be explicit, functions like:

```
f x = x + x
g y = 42
```

could be written as:

```
f x = let (x1, x2) = split x
      in x1 + x1
g y = let () = destroy y
      in 42
```

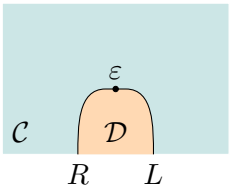
There are some situations, though, when duplicating or discarding a variable is undesirable. This is the case when the argument is an external resource, like a file handle, network port, or a chunk of memory allocated on the heap. Such resources are supposed to have well-defined lifetimes between being allocated and deallocated. Tracking lifetimes of objects that can be easily duplicated or discarded is very difficult and a notorious source of programming errors.

A programming model based on a cartesian category will always have this problem. The solution is to instead use a monoidal (closed) category that doesn't support duplication or destruction of objects. Such a category is a natural model for *linear types*. Elements of linear types are used in Rust and, at the time of this writing, are being tried in Haskell. In C++ there are constructs that mimic linearity, like `unique_ptr` and move semantics.

16.3 Comonads from Adjunctions

We've seen that an adjunction $L \dashv R$ between two functors $L\colon \mathcal{D} \to \mathcal{C}$ and $R\colon \mathcal{C} \to \mathcal{D}$ gives rise to a monad $R \circ L\colon \mathcal{D} \to \mathcal{D}$. The other composition, $L \circ R$, which is an endofunctor in $\mathcal{C}$, turns out to be a comonad.

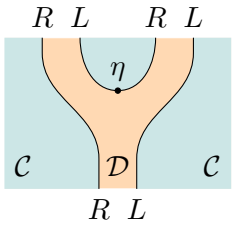
The counit of the adjunction serves as the counit of the comonad. This can be illustrated by the following string diagram:



The comultiplication is given by the whiskering of η:

$$\delta = L \circ \eta \circ R$$

as illustrated by this string diagram:



As before, comonad laws can be derived from triangle identities.

Costate comonad

We've seen that the state monad can be generated by the currying adjunction between the product and the exponential. The left functor was defined as a product with some fixed object S:

$$L_S A = A \times S$$

and the right functor was the exponentiation, parameterized by the same object S:

$$R_S C = C^S$$

The composition $L_S \circ R_S$ generates a comonad called the *costate comonad* or the *store comonad*.

Translated to Haskell, the right functor assigns a function type `s -> c` to `c`, and the left functor pairs `c` with `s`. The result of the composition is the endofunctor:

```
data Store s c = St (s -> c) s
```

or, using GADT notation:

```
data Store s c where
    St :: (s -> c) -> s -> Store s c
```

The functor instance post-composes the function to the first component of `Store`:

```
instance Functor (Store s) where
  fmap g (St f s) = St (g . f) s
```

The counit of this adjunction, which becomes the comonadic `extract`, is function application:

```
extract :: Store s c -> c
extract (St f s) = f s
```

The unit of this adjunction is a natural transformation $\eta \colon Id \to R_S \circ L_S$. We've used it as the `return` of the state monad. This is its component at `c`:

```
eta :: c -> (s -> (c, s))
eta c = \s -> (c, s)
```

To get `duplicate` we need to whisker η it between the two functors:

$$\delta = L_S \circ \eta \circ R_S$$

Whiskering on the right means taking the component of η at the object $R_S C$, and whiskering on the left means lifting this component using L_S. Since Haskell translation of whiskering is a tricky process, let's analyze it step-by-step.

For simplicity, let's fix the type `s` to, say, `Int`. We encapsulate the left functor into a `newtype`:

```
newtype Pair c = P (c, Int)
  deriving Functor
```

and keep the right functor a type synonym:

```
type Fun c = Int -> c
```

The unit of the adjunction can be written as a natural transformation using explicit `forall`:

```
eta :: forall c. c -> Fun (Pair c)
eta c = \s -> P (c, s)
```

We can now implement comultiplication as the whiskering of `eta`. The whiskering on the right is encoded in the type signature, by using the component of `eta` at `Fun c`. The whiskering on the left is done by lifting `eta` using the `fmap` defined for the `Pair` functor. We use the language pragma `TypeApplications` to make it explicit:

```
delta :: forall c. Pair (Fun c) -> Pair (Fun (Pair (Fun c)))
delta = fmap @Pair eta
```

This can be rewritten more explicitly as:

```
delta (P (f, s)) = P (\s' -> P (f, s'), s)
```

The `Comonad` instance can thus be written as:

```
instance Comonad (Store s) where
  extract (St f s) = f s
  duplicate (St f s) = St (St f) s
```

The store comonad is a useful programming concept. To understand that, let's consider again the case where `s` is `Int`.

We interpret the first component of `Store Int c`, the function `f :: Int -> c`, to be an accessor to an imaginary infinite stream of values, one for each integer.

The second component can be interpreted as the current index. Indeed, `extract` uses this index to retrieve the current value.

With this interpretation, `duplicate` produces an infinite stream of streams, each shifted by a different offset, and `extend` performs a convolution on this stream. Of course, laziness saves the day: only the values we explicitly demand will be evaluated.

Notice also that our earlier example of the `Signal` comonad is reproduced by `Store Double`.

Exercise 16.3.1. *A cellular automaton can be implemented using the store comonad. This is the co-Kleisli arrow describing rule 110:*

```
step :: Store Int Cell -> Cell
step (St f n) =
    case (f (n-1), f n, f (n+1)) of
    (L, L, L) -> D
    (L, D, D) -> D
    (D, D, D) -> D
    _ -> L
```

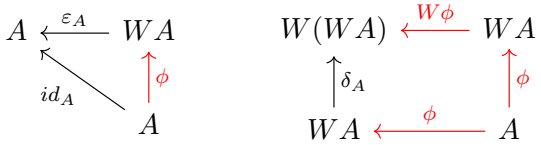
A cell can be either live or dead:

```
data Cell = L | D
    deriving Show
```

Run a few generation of this automaton. Hint: Use the function `iterate` *from the Prelude.*

Comonad coalgebras

Dually to monad algebras we have comonad coalgebras. Given a comonad (W, ε, δ), we can construct a coalgebra, which consists of a carrier object A and an arrow $\phi\colon A \to WA$. For this coalgebra to compose nicely with the comonad, we'll require that we can extract the value that was injected using ϕ, and that the lifting of ϕ is equivalent to duplication when acting on the result of ϕ:



Just like with monad algebras, comonad coalgebras form a category. Given a comonad (W, ε, δ) in $\mathcal{C}$, its comonad coalgebras form a category called the Eilenberg-Moore category (sometimes prefixed with co-) $\mathcal{C}^W$.

There is a co-Kleisli subcategory of $\mathcal{C}^W$ denoted by $\mathcal{C}_W$

Given a comonad W, we can construct an adjunction using either $\mathcal{C}^W$ or $\mathcal{C}_W$ that gives rise to W. The construction is fully analogous to the one for monads.

Lenses

The coalgebra for the `Store` comonad is of particular interest. We'll do some renaming first: we'll call the carrier `s` and the state `a`. The coalgebra is given by a function:

```
phi :: s -> Store a s
```

which is equivalent to a pair of functions:

```
set :: s -> a -> s
get :: s -> a
```

Such a pair is called a lens: `s` is called the source, and `a` is the focus.

With this interpretation `get` lets us extract the focus, and `set` replaces the focus with a new value to produce a new `s`.

Lenses were first introduced to describe the retrieval and modification of data in database records. Then they found application is working with data structures. A lens objectifies the idea of having read/write access to a part of a larger object. For instance, a lens can focus on one of the components of a pair or a particular component of a record. We'll discuss lenses and optics in the next chapter.

Let's apply the laws of the comonad coalgebra to a lens. For simplicity, let's omit data constructors from the equations. We get:

```
phi s = (set s, get s)
epsilon (f, a) = f a
delta (f, a) = (\x -> (f, x), a)
```

The first law tells us that applying the result of `set` to the result of `get` results in identity:

```
set s (get  s) = s
```

This is called the set/get law of the lens. Nothing changes when you replace the focus with the same focus.

The second law requires the application of `fmap phi` to the result of `phi`:

```
fmap phi (set s, get s) = (phi . set s, get s)
```

This should be equal to the application of `delta`:

```
delta (set s, get s) = (\x -> (set s, x), get s)
```

Comparing the two, we get:

```
phi . set s = \x -> (set s, x)
```

We can apply it to some `a`:

```
phi (set s a) = (set s, a)
```

Using the definition of `phi` gives us:

```
(set (set s a), get (set s a)) = (set s, a)
```

We get two equalitiies. The first components are functions, so we apply them to some `a'` and get the set/set lens law:

```
set (set s a) a' = set s a'
```

Setting the focus to `a` and then overwriting it with `a'` is the same as setting the focus directly to `a'`.

The second components give us the get/set law:

```
get (set s a) = a
```

After we set the focus to `a`, the result of `get` is `a`.

Lenses that satisfy these laws are called *lawful lenses*. They are comonad coalgebras for the store comonad.

Chapter 17

Ends and Coends

17.1 Profunctors

In the rarified air of category theory we encounter patterns that are so far removed from their origins that we have problems visualizing them. It doesn't help that the more abstract a pattern gets the more dissimilar the concrete examples of it are.

An arrow from A to B is relatively easy to visualize. We have a very familiar model for it: a function that consumes elements of A and produces elements of B. A hom-set is a collection of such arrows.

A functor is an arrow between categories. It consumes objects and arrows from one category and produces objects and arrows from another. We can think of it a recipe for building such objects and arrows from materials provided by the source category. In particular, we often think of an endofunctor as a container of building materials.

A profunctor maps a pair of objects $\langle A, B\rangle$ to a set $P\langle A, B\rangle$ and a pair of arrows:

$$\langle f\colon S \to A, g\colon B \to T\rangle$$

to a function:

$$P\langle f, g\rangle\colon P\langle A, B\rangle \to P\langle S, T\rangle$$

A profunctor is an abstraction that combines elements of many other abstractions. Since it's a functor $\mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$, we can think of it as constructing a set from a pair of objects and a function from a pair of arrows (one going in the opposite direction). This doesn't help our imagination though.

Fortunately, we have a good model for a profunctor: the hom-functor. The set of arrows between two objects behaves like a profunctor when you vary the objects. It also makes sense that there is a difference between varying the source and the target of the hom-set.

We can, therefore, think of an arbitrary profunctor as generalizing a hom-functor. A profunctor thus provides additional bridges between objects.

There is, however one big difference between an element of the hom-set $\mathcal{C}(A, B)$ and an element of the set $P\langle A, B\rangle$. Elements of hom-sets are arrows, and arrows can be composed. It's not immediately obvious how to compose profunctors.

However, the lifting of arrows by a profunctor can be seen as generalizing composition—just not between profuctors, but between hom-sets and profunctors. For instance, we

can precompose $P\langle A, B\rangle$ with an arrow $f\colon S \to A$ to obtain $P\langle S, B\rangle$:

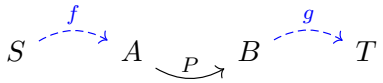
$$P\langle f, id_B\rangle\colon P\langle A, B\rangle \to P\langle S, B\rangle$$

Similarly, we can postcompose it with $g\colon B \to T$:

$$P\langle id_A, g\rangle\colon P\langle A, B\rangle \to P\langle A, T\rangle$$

This kind of heterogenous composition takes a composable pair consisting of an arrow and an element of a profunctor and produces an element of a profunctor.

In general, a profunctor can be extended this way on both sides by lifting a pair of arrows:



Collages

There is no reason to restrict a profunctor to a single category. We can easily define a profunctor between two categories as a functor $P\colon \mathcal{C}^{op} \times \mathcal{D} \to \mathbf{Set}$. Such a profunctor can be used to glue two categories together by generating the missing hom-sets from the objects in $\mathcal{C}$ to the objects in $\mathcal{D}$.

A collage (or a cograph) of two categories $\mathcal{C}$ and $\mathcal{D}$ is a category whose objects are objects from both categories (a disjoint union). A hom-set between two objects X and Y is either a hom-set in $\mathcal{C}$, if both objects are in $\mathcal{C}$; a hom-set in $\mathcal{D}$, if both are in $\mathcal{D}$; or the set $P\langle X, Y\rangle$ if X is in $\mathcal{C}$ and Y is in $\mathcal{D}$. Otherwise the hom-set is empty.

Composition of morphisms is the usual composition, except if one of the morphisms is an element of $P\langle X, Y\rangle$. In that case we use the profunctor to lift the morphism we're trying to compose.

It's easy to see that a collage is indeed a category. The new morphisms that go between the two sides of the collage are sometimes called heteromorphisms. They can only go from $\mathcal{C}$ to $\mathcal{D}$, never the other way around.

Seen this way, a profunctor $\mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$ should really be called an endo-profunctor. It defines a collage of $\mathcal{C}$ with itself.

Exercise 17.1.1. *Show that there is a functor from a collage of two categories to a stick-figure "walking arrow" category that has two objects and one arrow between them (and two identity arrows).*

Exercise 17.1.2. *Show that, if there is a functor from $\mathcal{C}$ to the walking arrow category then $\mathcal{C}$ can be split into a collage of two categories.*

Profunctors as relations

Under a microscope, a profunctor looks like a hom-functor, and the elements of the set $P\langle A, B\rangle$ look like individual arrows. But when we zoom out, we can view a profunctor as a relation between objects. These are not the usual relations; they are *proof-relevant* relations.

To understand this concept better, let's consider a regular functor $F\colon \mathcal{C} \to \mathbf{Set}$ (in other words, a co-presheaf). One way to interpret it is to say that it definines a subset

of objects of $\mathcal{C}$, namely those objects that are mapped to non-empty sets. Every element of FA is then treated as a proof that A is a member of this subset. If, on the other hand, FA is an empty set, then A is not a member of the subset.

We can apply the same interpretation to profunctors. If the set $P\langle A, B\rangle$ is empty, we say that B is not related to A. If it's not empty, we say that each element of the set represents a proof that B is related to A. We can then treat a profunctor as a proof-relevant relation.

Notice that we don't assume anything about this relation. It doesn't have to be reflexive, as it's possible for $P\langle A, A\rangle$ to be empty (in fact, $P\langle A, A\rangle$ makes sense only for endo-profunctors). It doesn't have to be symmetric either.

Since the hom-functor is an example of an (endo-) profunctor, this interpretation lets us view the hom-functor in a new light, as a built-in proof-relevant relation between objects in a category. If there's an arrow between two objects, they are related. Notice that this relation is reflexive, since $\mathcal{C}(A, A)$ is never empty: at the very least, it contains the identity morphism.

Moreover, as we've seen before, hom-functors interact with profunctors. If A is related to B through P, and the hom-sets $\mathcal{C}(S, A)$ and $\mathcal{D}(B, T)$ are non-empty, then automatically S is related to T through P. Profunctors are therefore proof-relevant relations that are compatible with the structure of the categories in which they operate.

We know how to compose a profunctor with hom-functors, but how would we compose two profunctors? We can get a clue from the composition of relations.

Suppose that you want to charge your cellphone, but you don't have a charger. In order to connect you to a charger it's enough that you have a friend who owns a charger. Any friend will do. You compose the relation of having a friend with the relation of a person having a charger to get a relation of being able to charge your phone. The proof that you can charge your phone is a pair of proofs, one of friendship and one of the possession of a charger.

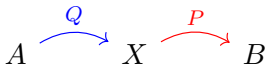
In general, we say that two objects are related by the composite relation if there exists an object in the middle that is related to both of them.

Profunctor composition in Haskell

Composition of relations can be translated to profunctor composition in Haskell. Let's first recall the definition of a profunctor:

```
class Profunctor p where
  dimap :: (s -> a) -> (b -> t) -> (p a b -> p s t)
```

The key to understanding profunctor composition is that it requires the *existence* of the object in the middle. For object B to be related to object A through the composite $P \diamond Q$ there has to exist an object X that bridges the gap:



This can be encoded in Haskell using an existential type. Given two profunctors `p` and `q`, their composition is a new profunctor `Procompose p q`:

```
data Procompose p q a b where
  Procompose ::  q a x -> p x b -> Procompose p q a b
```

We are using a `GADT` to express the existential nature of the object `x`. The two arguments to the constructor can be seen as a pair of proofs: one proves that `x` is related to `a`, and the other that `b` is related to `x`. This pair then constitutes the proof that `b` is related to `a`.

An existential type can be seen as a generalization of a sum type. We are summing over all possible types `x`. Just like a finite sum can be constructed by injecting one of the alternatives (think of the two constructors of `Either`), the existential type can be constructed by picking one particular type for `x` and injecting it into the definition of `Procompose`.

Just as mapping out from a sum type requires a pair of function, one per each alternative; a mapping out from an existential type requires a family of functions, one per every type. Such a family, in Haskell, is given by a polymorphic function:

```
mapOut :: Procompose p q a b -> (forall x. q a x -> p x b -> c) -> c
mapOut (Procompose qax pxb) f = (f qax pxb)
```

The composition of profunctors is again a profunctor, as can be seen from this instance:

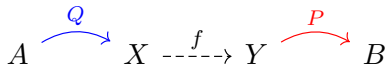
```
instance (Profunctor p, Profunctor q) => Profunctor (Procompose p q)
  where
    dimap l r (Procompose qax pxb) =
                Procompose (dimap l id qax) (dimap id r pxb)
```

This just says that you can extend the composite profunctor by extending the first one on the left and the second one on the right.

The fact that this definition of profunctor composition happens to work in Haskell is due to parametricity. The language constraints the types of profunctors in a way that makes it work. In general, though, taking a simple sum over intermediate objects would result in over-counting, so in category theory we have to compensate for that.

17.2 Coends

The over-counting in the naive definition of profunctor composition happens when two candidates for the object in the middle are connected by a morphism:



We can either extend Q on the right, by lifting $Q\langle id, f\rangle$, and use Y as the middle object; or we can extend P on the left, by lifting $P\langle f, id\rangle$, and use X as the intermediary.

In order to avoid the double-counting, we have to tweak our definition of a sum type when applied to profunctors. The resulting construction is called a coend.

First, let's re-formulate the problem. We are trying to sum over all objects X in the cartesian product:

$$P\langle A, X\rangle \times Q\langle X, B\rangle$$

The double-counting happens because we can open up the gap between the two profunctors, as long as there is a morphism that we can fit between them. So we are really looking at a more general product:

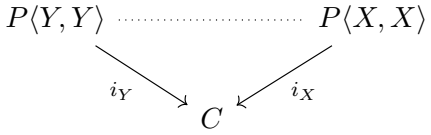
$$P\langle A, X\rangle \times Q\langle Y, B\rangle$$

The important observation is that, if we fix the endpoints A and B, this product is a profunctor in $\langle Y, X\rangle$. This is easily seen after a little rearrangement (up to isomorphism):

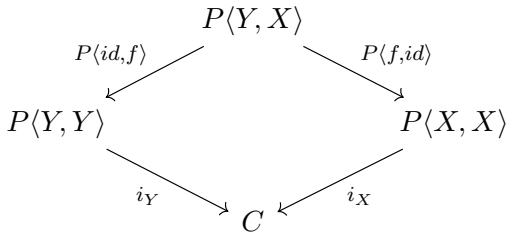
$$Q\langle Y, B\rangle \times P\langle A, X\rangle$$

We are interested in the sum of the diagonal parts of this profunctor, that is when X is equal to Y.

So let's see how we would go about defining the sum of all diagonal entries of a general profunctor P. The sum is defined by injections; in this case, one per every object in the category. Here just two of them are shown:



If we were defining a sum, we'd make it a universal set equipped with such injections. But because we are dealing with profunctors, we want to identify the injections that are related by "extending" a common ancestor. We want the following diagram to commute, whenever there is a connecting morphism $f\colon X \to Y$:



This diagram is called a co-wedge, and its commuting condition is called the co-wedge condition. The universal co-wedge is called a coend.

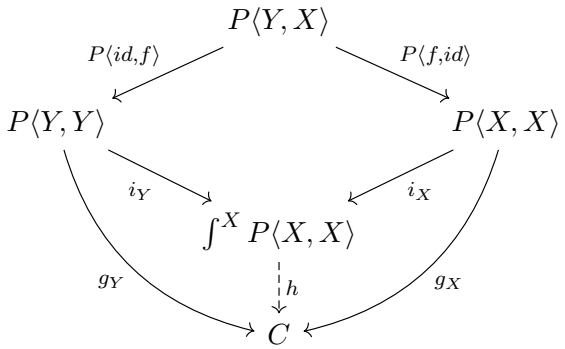
Since a coend generalizes a sum to a potentially infinite domain, we write it using the integral sign, with the "integration variable" in the suffix position:

$$\int^{X\colon \mathcal{C}} P\langle X, X\rangle$$

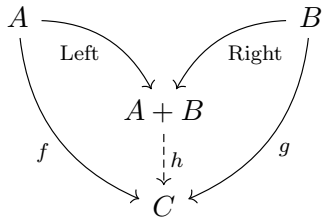
Universality means that, whenever there is a set C with a family of functions $g_X\colon P\langle X, X\rangle \to C$, there is a unique mapping out:

$$h\colon \int^{X\colon \mathcal{C}} P\langle X, X\rangle \to C$$

that factorizes every g_X through the injection i_X:



Compare this with the definition of a sum of two objects:



Just like the sum was defined as a universal cospan, a coend is defined as a universal co-wedge.

If you were to construct a coend, you would start with a sum (discriminated union) of all the sets $P\langle X, X\rangle$. Then you would identify the elements of this sum that satisfy the co-wedge condition. You'd identify the element $x \in P\langle X, X\rangle$ with the element $y \in P\langle Y, Y\rangle$ whenever there is an element $z \in P\langle Y, X\rangle$ and a morphism $f\colon X \to Y$, such that:

$$P\langle id, f\rangle(z) = y$$

and

$$P\langle f, id\rangle(z) = x$$

Notice that, in a discrete category (which is just a set of objects with no arrows between them) the co-wedge condition is trivial (there are no fs other than identities), so a coend is just a straightforward sum (discriminated union) of the diagonal sets $P\langle X, X\rangle$.

Profunctor composition using coends

Equipped with the definition of a coend we can now formally define the composition of two profunctors:

$$(P \diamond Q)\langle A, B\rangle = \int^{X\colon \mathcal{C}} Q\langle A, X\rangle \times P\langle X, B\rangle$$

Compare this with:

```
data Procompose p q a b where
  Procompose ::  q a x -> p x b -> Procompose p q a b
```

The reason why, in Haskell, we don't have to worry about the co-wedge condition is analogous to the reason why all parametrically polymorphic functions automatically satisfy the naturality condition. A coend is defined using a family of injections; in Haskell they are all given as one polymorphic function:

```
data Coend p where
  Coend ::  p x x -> Coend p
```

Coends introduce a new level of abstraction for dealing with profunctors. Calculations using coends usually take advantage of their mapping-out property. To define a mapping out of a coend to some set C:

$$\int^{X} P\langle X, X\rangle \to C$$

it's enough to define a family of functions from the diagonal entries of the profunctor to C:

$$g_X : P\langle X, X\rangle \to C$$

You can get a lot of mileage from this trick, especially when combined with the Yoneda lemma. We'll see examples of this in what follows.

Exercise 17.2.1. *Define a* `Profunctor` *instance for the pair of profunctors:*

```
newtype ProPair q p a b x y = ProPair (q a y, p x b)
```

Hint: Keep the first four parameters fixed:

```
instance (Profunctor p, Profunctor q) => Profunctor (ProPair q p a b)
```

Exercise 17.2.2. *Profunctor composition can be expressed using a coend:*

```
newtype CoEndCompose p q a b = CoEndCompose (Coend (ProPair q p a b))
```

Define a `Profunctor` *instance for* `CoEndCompose`*.*

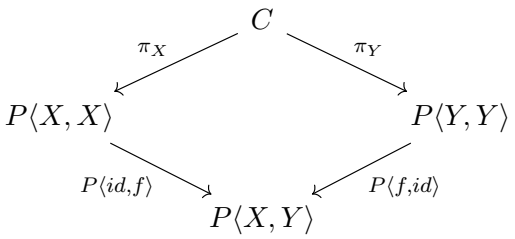
17.3 Ends

Just like a coend generalizes a sum of the diagonal elements of a profunctor; its dual, an end, generalizes the product. A product is defined by its projections, and so is an end.

The generalization of a span that we used in the definition of a product would be a set C with a family of projections, one per every object X in the category:

$$\pi_X : C \to P\langle X, X\rangle$$

The dual to a co-wedge is called a wedge:



You can think of constructing a set C by first starting with a bigger set that is equipped with all the projections, and then restricting it to the elements that can be connected by way of lifting a morphism.

The end is a universal wedge. We use the integral sign for it too, but with the "integration variable" in the subscript position.

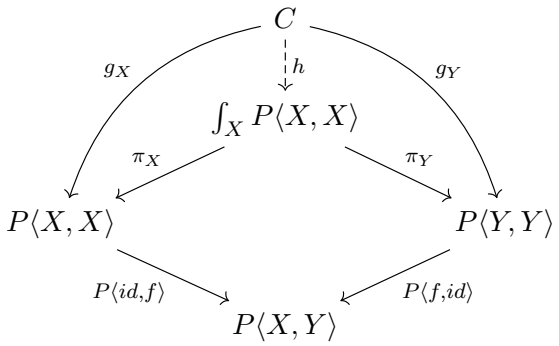
$$\int_X P\langle X, X\rangle$$

You might be wondering why integrals based on multiplication are rarely used in calculus. That's because we can use a logarithm to replace multiplication with addition. We don't have this luxury in category theory.

An end is a set equipped with a family of functions (projections):

$$\pi_A \colon \left(\int_X P\langle X, X\rangle \right) \to P\langle A, A\rangle$$

satisfying the wedge condition. It is universal among such sets; that is, for any other set C equipped with a family of functions g_X, satisfying the wedge condition, there is a unique function h that factorizes the family g_X through the family π_X.



If you were to construct this set, you'd start with a product of all $P\langle X, X\rangle$ for all the objects in the category.

Imagine, for a moment, using the singleton set 1 in place of C. The family g_X would select one element from each $P\langle X, X\rangle$. This would give you a giant tuple. You'd weed out most of these tuples, leaving only the ones that satisfy the wedge condition.

Again, in Haskell, due to parametricity, the wedge condition is automatically satisfied, and the definition of an end for a profunctor p simplifies to:

```
type End p = forall x. p x x
```

The Haskell implementation of an `End` doesn't showcase the fact that it is dual to a `Coend`. This is because Haskell doesn't have a built-in syntax for existential types. If it did, the `Coend` would be implemented as:

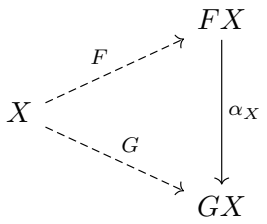
```
type Coend p = exists x. p x x
```

The existential/universal duality between a `Coend` and an `End` means that it's easy to construct a `Coend`—all you need is to pick one type `x` for which you have a value of the type `p x x`. On the other end, to construct an `End` you have to provide a whole family of values `p x x`, one for every type `x`. In other words, you need a polymorphic formula that is parameterized by `x`. A definition of a polymorphic function is a canonical example of such a formula.

Natural transformations as an end

The most interesting application of an end is in concisely defining natural transformations. Consider two functors, F and G, going between two categories $\mathcal{B}$ and $\mathcal{C}$. A natural transformation between them is a family of arrows α_X in $\mathcal{C}$. You can think of

it as picking one element α_X from each hom-set $\mathcal{C}(FX, GX)$.



The mapping $\langle A, B\rangle \to \mathcal{C}(FA, GB)$ is a profunctor. Its action on a pair of arrows $\langle f, g\rangle$ is a combination of pre- and post-composition of lifted arrows $(Gg) \circ - \circ (Ff)$.

Indeed, an element of the set $\mathcal{C}(FA, GB)$ is an arrow $h\colon FA \to GB$. We are trying to lift a pair of arrows $f\colon S \to A$ and $g\colon B \to T$. We can do it with a pair of arrows in $\mathcal{C}$: the first one is $Ff\colon FS \to FA$, and the second one is $Gg\colon GB \to GT$. The composition $Gg \circ h \circ Ff$ gives us the desired result $FS \to GT$, which is an element of $\mathcal{C}(FS, GT)$.

$$FS \xrightarrow{Ff} FA \xrightarrow{h} GB \xrightarrow{Gg} GT$$

The diagonal parts of this profunctor are good candidates for the components of a natural transformation. In fact, the end:

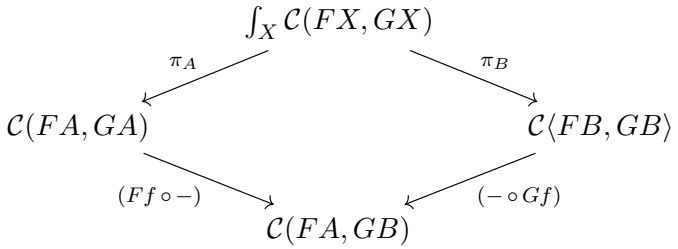
$$\int_{X\colon \mathcal{B}} \mathcal{C}(FX, GX)$$

defines a set of natural transformations from F to G.

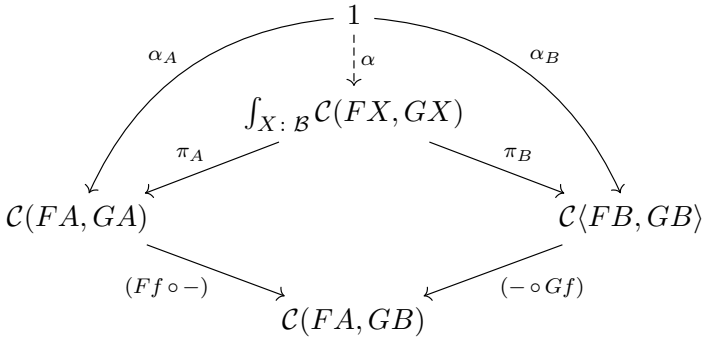
In Haskell, this is consistent with our earlier definition:

```
type Natural f g = forall x. f x -> g x
```

In category theory, though, we have to check the wedge condition. Plugging in our profunctor, we get:



We can focus on a single element of the set $\int_X \mathcal{C}(FX, GX)$ by instantiating the universal condition for the singleton set:



It picks the component α_A from the hom-set $\mathcal{C}(FA, GA)$ and the component α_B from $\mathcal{C}(FB, GB)$. The wedge condition then boils down to:

$$Ff \circ \alpha_A = \alpha_B \circ Gf$$

for any $f\colon A \to B$. This is exactly the naturality condition. So an element α of this end is indeed a natural transformation.

The set of natural transformations, or the hom-set in the functor category, is thus given by the end:

$$[\mathcal{C}, \mathcal{D}](F, G) \cong \int_{X\colon \mathcal{B}} \mathcal{C}(FX, GX)$$

As we discussed earlier, to construct an `End` we have to give it a whole family of values parameterized by types. Here, these values are polymorphic functions.

17.4 Continuity of the Hom-Functor

In category theory, a functor is called continuous if it preserves limits (and co-continuous, if it preserves colimits). It means that, if you have a diagram in the source category, then it doesn't matter if you first use the functor to map the diagram, and then take the limit in the target category; or take the limit in the source category, and then map this limit using the functor.

The hom-functor is an example of a functor that is continuous in its second argument. Since a product is the simplest example of a limit, we should have:

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

The left hand side applies the hom-functor to the product. The right hand side maps the diagram, a pair of objects, and takes the product in the target category, which for the hom-functor is **Set**. The two sides are isomorphic by the universal property of the product: the mapping into the product is defined by a pair of mappings into the two objects.

Continuity of the hom-functor in the first argument is reversed: it maps colimits to limits. Again, the simplest example of a colimit is the sum, so we have:

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

This follows from the universality of the sum: a mapping out of the sum is defined by a pair of mapping out of the two objects.

It can be shown that an end can be expressed as a limit, and a coend as a colimit. Therefore, by continuity of the hom-functor, we can always pull out the integral sign from inside a hom-set. By analogy with the product, we have the mapping-in formula for an end:

$$\mathbf{Set}\left(X, \int_A P\langle A, A\rangle\right) \cong \int_A \mathbf{Set}(X, P\langle A, A\rangle)$$

By analogy with the sum, we have a mapping-out formula for the coend:

$$\mathbf{Set}\left(\int^A P\langle A, A\rangle, X\right) \cong \int_A \mathbf{Set}(P\langle A, A\rangle, X)$$

Notice that, in both cases, the right-hand side is an end.

17.5 Ninja Yoneda Lemma

Having expressed natural transformations as an end, we can now rewrite the Yoneda lemma. This is the original formulation:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(A, -), F) \cong FA$$

F is a (covariant) functor from $\mathcal{C}$ to **Set** (a co-presheaf) and so is the hom-functor $\mathcal{C}(A, -)$. Expressing the set of natural transformations as an end we get:

$$\int_{X\,:\,\mathcal{C}} \mathbf{Set}(\mathcal{C}(A, X), FX) \cong FA$$

Similarly, we have the Yoneda lemma for a contravariant functor (a presheaf) G:

$$\int_{X\,:\,\mathcal{C}} \mathbf{Set}(\mathcal{C}(X, A), GX) \cong GA$$

These versions of the Yoneda lemma, expressed in terms of ends, are often half-jokingly called ninja-Yoneda lemmas. The fact that the "integration variable" is explicit makes them somewhat easier to parse.

There is also a dual set of ninja co-Yoneda lemmas that use coends instead. For a covariant functor, we have:

$$\int^{X\,:\,\mathcal{C}} \mathcal{C}(X, A) \times FX \cong FA$$

and for the contravariant one we have:

$$\int^{X\,:\,\mathcal{C}} \mathcal{C}(A, X) \times GX \cong GA$$

Physicists might notice the similarity of these formulas to integrals involving the Dirac delta function (actually, a distribution). This is why profunctors are sometimes called distributors, following the adage that "distributors are to functors as distributions are to functions."

Yet another name for profunctors is "bimodules."

The proof of the co-Yoneda lemma is quite instructive, as it uses a few common tricks. Most importantly, we rely on the corollary of the Yoneda lemma, which says that, if the mappings out to an arbitrary object are isomorphic, then the objects themselves are isomorphic. We'll start, therefore, with such a mapping-out to an arbitrary set S:

$$\mathbf{Set}\left(\int^{X\,:\,\mathcal{C}} \mathcal{C}(X, A) \times FX, S\right)$$

Using the co-continuity of the hom-functor, we can pull out the integral sign, replacing the coend with the end:

$$\int_{X\,:\,\mathcal{C}} \mathbf{Set}\left(\mathcal{C}(X, A) \times FX, S\right)$$

Since the category of sets is cartesian closed, we can curry the product:

$$\int_{X\,:\,\mathcal{C}} \mathbf{Set}\left(\mathcal{C}(X, A), S^{FX}\right)$$

We can now use the Yoneda lemma to "integrate over X." The result is S^{FA}. Finally, in **Set**, the exponential object is isomorphic to the hom-set:

$$S^{FA} \cong \mathbf{Set}(FA, S)$$

Since S was arbitrary, we conclude that:

$$\int^{X : \mathcal{C}} \mathcal{C}(X, A) \times FX \cong FA$$

Exercise 17.5.1. *Prove the contravariant version of the co-Yoneda lemma.*

Yoneda lemma in Haskell

We've seen the Yoneda lemma implemented in Haskell. We can now rewrite it in terms of an end. We start by defining a profunctor that will go under the end. Its type constructor takes a functor `f` and a type `a` and generates a profunctor that's contravariant in `x` and covariant in `y`:

```
data Yo f a x y = Yo ((a -> x) -> f y)
```

The Yoneda lemma establishes the isomorphism between the end over this profunctor and the type obtained by acting with the functor `f` on `a`. This isomorphism is witnessed by a pair of functions:

```
yoneda :: Functor f => End (Yo f a) -> f a
yoneda (Yo g) = g id

yoneda_1 :: Functor f => f a -> End (Yo f a)
yoneda_1 fa = Yo (\h -> fmap h fa)
```

Similarly, the co-Yoneda lemma uses a coend over the following profunctor:

```
data CoY f a x y = CoY (x -> a) (f y)
```

The isomorphism is witnessed by a pair of functions. The first one says that if you have a function `x -> a` and a functorful of `x` then you can make a functorful of `a` using the `fmap`:

```
coyoneda :: Functor f => Coend (CoY f a) -> f a
coyoneda (Coend (CoY g fa)) = fmap g fa
```

You can do it without knowing anything about the existential type `x`.

The second says that if you have a functorful of `a` then you can create a coend by injecting it (together with the identity function) into the existential type:

```
coyoneda_1 :: Functor f => f a -> Coend (CoY f a)
coyoneda_1 fa = Coend (CoY id fa)
```

17.6 The Bicategory of Profunctors

Since we know how to compose profunctors using coends, the question arises: is there a category in which they serve as morphisms? The answer is yes, as long as we relax the rules a bit. The problem is that the categorical laws for profunctor composition are satisfied up to isomorphism.

For instance, we can try to show associativity of profunctor composition. We start with:

$$((P \diamond Q) \diamond R)\langle S, T\rangle = \int^B \left(\int^A P\langle S, A\rangle \times Q\langle A, B\rangle \right) \times R\langle B, T\rangle$$

and, after a few transformations, arrive at:

$$(P \diamond (Q \diamond R))\langle S, T\rangle = \int^A P\langle S, A\rangle \times \left(\int^B Q\langle A, B\rangle \times R\langle B, T\rangle \right)$$

We use the associativity of the product and the fact that we can switch the order of coends (it's called the Fubini theorem). Both are true only up to isomorphism. We don't get associativity "on the nose."

The identity profunctor turns out to be the hom-functor, which can be written symbolically as $\mathcal{C}(-, =)$, with placeholders for both arguments. For instance:

$$\left(\mathcal{C}(-, =) \diamond P\right)\langle S, T\rangle \cong \int^A \mathcal{C}(S, A) \times P\langle A, T\rangle = P\langle S, T\rangle$$

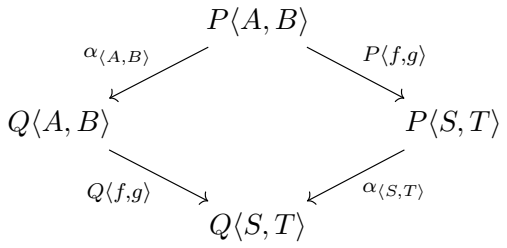
This is the consequence of the (contravariant) ninja co-Yoneda lemma, which is also an isomorphism—not an equality.

A category in which categorical laws are satisfied up to isomorphism is called a bicategory. Notice that such a category must be equipped with 2-cells—morphisms between morphisms, which we've seen in the definition of a 2-category. We need those in order to be able to define isomorphisms between 1-cells.

A bicategory **Prof** has (small) categories as objects, profunctors as 1-cells, and natural transformations as 2-cells.

Since profunctors are functors $\mathcal{C}^{op} \times \mathcal{D} \to \mathbf{Set}$, the standard definition of natural transformations between them applies. It's a family of functions parameterized by objects of $\mathcal{C}^{op} \times \mathcal{D}$, which are themseves pairs of objects.

The naturality condition for a transformation $\alpha_{\langle A,B\rangle}$ between two profunctors P and Q takes the form:



for every pair of arrows:

$$\langle f \colon S \to A, g \colon B \to T\rangle$$

17.7 Existential Lens

The first rule of category-theory club is that you don't talk about the internals of objects.

The second rule of category-theory club is that, if you have to talk about the internals of objects, use arrows only.

Existential lens in Haskell

What does it mean for an object to be a composite—to have parts? At the very minimum, you should be able to retrieve a part of such an object. Even better if you can replace that part with a new one. This pretty much defines a lens:

```
get :: s -> a
set :: s -> a -> s
```

Here, `get` extracts the part `a` from the whole `s`, and `set` replaces that part with a new `a`. Lens laws help to reinforce this picture. And it's all done in terms of arrows.

Another way of describing a composite object is to say that it can be split into a focus and a residue. The trick is that, although we want to know what type the focus is, we don't care about the type of the residue. All we need to know about the residue is that it can be combined with the focus to recreate the whole object.

In Haskell, we would express this idea using an existential type:

```
data Lens s a where
    Lens :: (s -> (c, a), (c, a) -> s) -> Lens s a
```

This tells us that there exists some unspecified type `c` such that `s` can be split into, and reconstructed from, a product `(c, a)`.

The `get`/`set` version of the lens can be derived from this existential form.

```
toGet :: Lens s a -> (s -> a)
toGet (Lens (l, r)) = snd . l

toSet :: Lens s a -> (s -> a -> s)
toSet (Lens (l, r)) s a = r (fst (l s), a)
```

Notice that we don't need to know anything about the type of the residue. We take advantage of the fact that the existential lens contains both the producer and the consumer of `c`.

It's impossible to extract a "naked" residue, as witnessed by the fact that the following code doesn't compile:

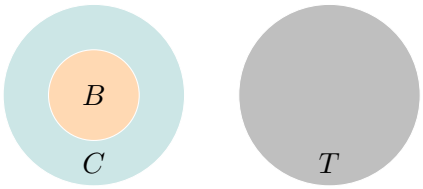
```
getResidue :: Lens s a -> c
getResidue (Lens (l, r)) = fst . l
```

Existential lens in category theory

We can easily translate the new definition of the lens to category theory by expressing the existential type as a coend:

$$\int^{C} \mathcal{C}(S, C \times A) \times \mathcal{C}(C \times A, S)$$

In fact, we can generalize it to a type-changing lens, in which the focus A can be replaced with a new focus of a different type B. Replacing A with B will produce a new composite object T:



The lens is now parameterized by two pairs of objects: $\langle S, T\rangle$ for the outer ones, and $\langle A, B\rangle$ for the inner ones. The residue C remains the same:

$$\mathcal{L}\langle S,T\rangle\langle A,B\rangle = \int^{C} \mathcal{C}(S, C \times A) \times \mathcal{C}(C \times B, T)$$

The product under the coend is the diagonal part of the profunctor that is covariant in Y and contravariant in X:

$$\mathcal{C}(S, Y \times A) \times \mathcal{C}(X \times B, T)$$

Exercise 17.7.1. *Show that:*

$$\mathcal{C}(S, Y \times A) \times \mathcal{C}(X \times B, T)$$

is a profunctor in $\langle X, Y\rangle$.

Type-changing lens in Haskell

In Haskell, we can define a type-changing lens as an existential type:

```
data Lens s t a b where
 Lens :: (s -> (c, a)) -> ((c, b) -> t) -> Lens s t a b
```

As before, we can use it to get and set the focus:

```
toGet :: Lens s t a b -> (s -> a)
toGet (Lens l r) = snd . l

toSet :: Lens s t a b -> (s -> b -> t)
toSet (Lens l r) s a = r (fst (l s), a)
```

The simplest example of a lens acts on a product. It can extract or replace one component of the product, treating the other as the residue. In Haskell, we'd implement it as:

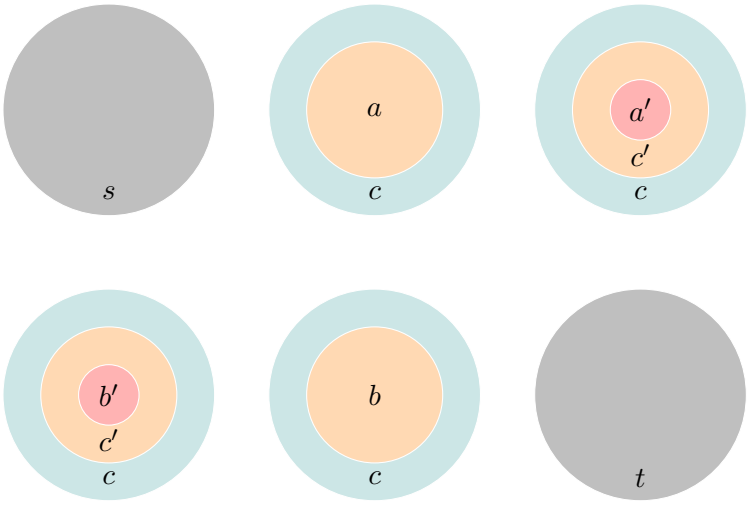
```
prodLens :: Lens (c, a) (c, b) a b
prodLens = Lens id id
```

Here, the type of the whole is a product `(c, a)`. When we replace `a` with `b` we end up with the target type `(c, b)`.

Lens composition

The main advantage of using lenses is that they compose. A composition of two lenses lets us zoom in on a subcomponent of a component.

Suppose that we start with a lens that lets us access and modify the focus described by the pair `a` and `b`. This focus is part of a whole described by the pair `s` and `t`. We also have a lens that can access the focus of `a'` and `b'` inside the whole of `a` and `b`. We can now construct a new lens that can access `a'` and `b'` inside of `s` and `t`. The trick is to realize that we can take as the new residue a product of the two residues:




```
compLens :: Lens a b a' b' -> Lens s t a b -> Lens s t a' b'
compLens (Lens l2 r2) (Lens l1 r1) = Lens l3 r3
  where l3 = assoc' . bimap id l2  . l1
        r3 = r1 . bimap id r2 . assoc
```

The left mapping in the new lens is given by the following composite:

$$s \xrightarrow{l_1} (c, a) \xrightarrow{(id, l_2)} (c, (c', a)) \xrightarrow{assoc'} ((c, c'), a)$$

and the right mapping is given by:

$$((c, c'), b') \xrightarrow{assoc} (c, (c', b')) \xrightarrow{(id, r_2)} (c, b) \xrightarrow{r_1} t$$

We have used the associativity and functoriality of the product:

```
assoc :: ((c, c'), b') -> (c, (c', b'))
assoc ((c, c'), b') = (c, (c', b'))

assoc' :: (c, (c', a')) -> ((c, c'), a')
assoc' (c, (c', a')) = ((c, c'), a')

instance Bifunctor (,) where
  bimap f g (a, b) = (f a, g b)
```

As an example, let's compose two product lenses:

```
l3 :: Lens (c, (c', a')) (c, (c', b')) a' b'
l3 = compLens prodLens prodLens
```

and apply it to a nested product:

```
x :: (String, (Bool, Int))
x = ("Outer", (True, 42))
```

Our composite lens lets us not only retrieve the innermost component:

```
toGet l3 x
> 42
```

but also replace it with a value of a different type (here, `Char`):

```
toSet l3 x 'z'
> ("Outer",(True,'z'))
```

Category of lenses

Since lenses can be composed, you might be wondering if there is a category in which lenses serve as hom-sets.

Indeed, there is a category **Lens** whose objects are pairs of objects in $\mathcal{C}$, and arrows from $\langle S, T\rangle$ to $\langle A, B\rangle$ are elements of $Lens\langle S, T\rangle\langle A, B\rangle$.

The formula for the composition of existential lenses is too complicated to be useful in practice. In the next chapter we'll see an alternative representation of lenses using Tambara modules, in which composition of lenses is just a composition of functions.

17.8 Lenses and Fibrations

There is an alternative view of lenses using the language of fiber bundles. A projection p that defines a fibration can be seen as "decomposing" the bundle E into fibers.

In this view, p plays the role of `get`:

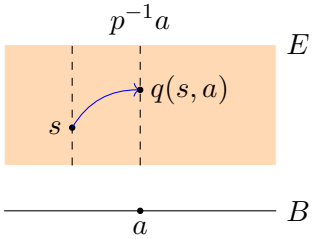
$$p\colon E \to B$$

The base B represents the type of the focus.

The other part of the lens, `set`, is a mapping:

$$q\colon E \times B \to E$$

It can be interpreted as "transporting" an element of the bundle to a new fiber.



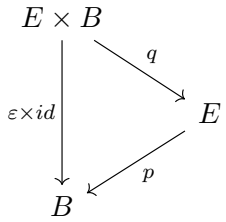
We know that $q(s, a)$ transports s to a fiber over a because of the get/set lens law:

```
get (set s a) = a
```

We can rewrite this law in terms of p and q:

$$p \circ q = \pi_1$$

Equivalently, we can represent it as a commuting diagram:



Here, instead of using the projection π_2, I used a comonoidal counit ε:

$$\varepsilon \colon E \to 1$$

and the unit law for the product. Using a comonoid makes it easier to generalize this construction to a tensor product in a monoidal category.

Here's the set/get law:

```
set s (get  s) = s
```

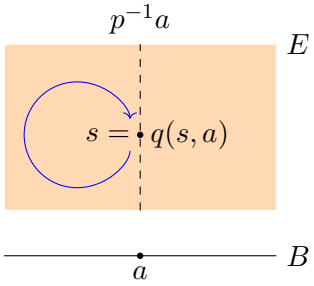
We can write it in terms of a comonoidal comultiplication:

$$\delta \colon E \to E \times E$$

The set/get law requires the following composite to be an identity:

$$E \xrightarrow{\delta} E \times E \xrightarrow{id \times p} E \times B \xrightarrow{q} E$$

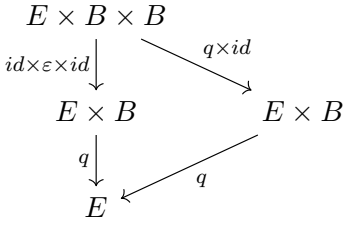
Here's the illustration of this law in a bundle:



Finally, here's the set/set law:

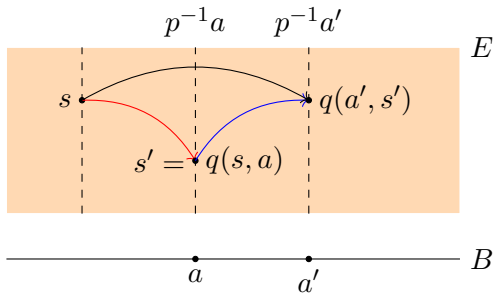
```
set (set s a) a' = set s a'
```

and the corresponding commuting diagram:



Again, I used the counit rather than a projection from a product.

This is what the set/set law looks like in a bundle:



You may notice that the last two laws look like unit and composition for the bundle transport q.

A type-changing lens generalizes transport to act between bundles:

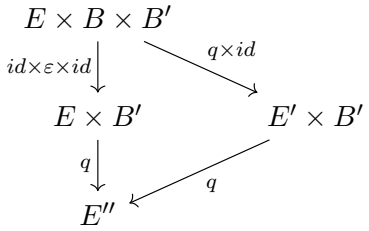
$$q \colon E \times B' \to E'$$

where E' is a bundle over B'.

The unit law applies to transport within one bundle, but the composition law can be extended to jump between three different bundles.

$$q(q(s, b), b') = q(s, b')$$

or, diagrammatically:



The existential representation of the lens can also be described using the language of bundles. The existential residue becomes a generic fiber F. The two components of the existential lens:

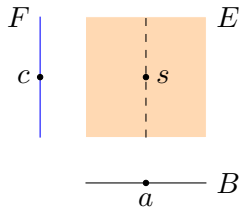
$$\int^{C} \mathcal{C}(S, C \times A) \times \mathcal{C}(C \times A, S)$$

are the mappings:

$$E \to F \times B$$
$$F \times B \to E$$

They decompose the bundle into the product of the generic fiber and the base.

A type-changing lens reuses the same generic fiber while switching the bundles.



We'll see later that this picture can be further generalized by replacing the generic fiber with a monoidal category that can act on the fibers of E in a way reminiscent of gauge transformations in physics.

17.9 Important Formulas

This is a handy (co-)end calculus cheat-sheet.

- Continuity of the hom-functor:

$$\mathbf{Set}\left(X, \int_A P\langle A, A\rangle\right) \cong \int_A \mathbf{Set}(X, P\langle A, A\rangle)$$

- Co-continuity of the hom-functor:

$$\mathbf{Set}\left(\int^A P\langle A, A\rangle, X\right) \cong \int_A \mathbf{Set}(P\langle A, A\rangle, X)$$

- Ninja Yoneda:

$$\int_X \mathbf{Set}(\mathcal{C}(A, X), FX) \cong FA$$

- Ninja co-Yoneda:

$$\int^X \mathcal{C}(X, A) \times FX \cong FA$$

- Ninja Yoneda for contravariant functors (pre-sheaves):

$$\int_X \mathbf{Set}(\mathcal{C}(X, A), GX) \cong GA$$

- Ninja co-Yoneda for contravariant functors:

$$\int^X \mathcal{C}(A, X) \times GX \cong GA$$

Chapter 18

Tambara Modules

Index