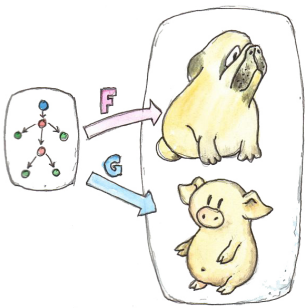
10

Natural Transformations

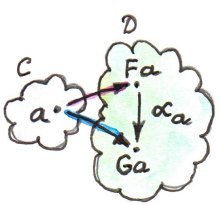
We talked about functors as mappings between categories that preserve their structure.

A functor "embeds" one category in another. It may collapse multiple things into one, but it never breaks connections. One way of thinking about it is that with a functor we are modeling one category inside another. The source category serves as a model, a blueprint, for some structure that's part of the target category.



There may be many ways of embedding one category in another. Sometimes they are equivalent, sometimes very different. One may collapse the whole source category into one object, another may map every object to a different object and every morphism to a different morphism. The same blueprint may be realized in many different ways. Natural transformations help us compare these realizations. They are mappings of functors — special mappings that preserve their functorial nature.

Consider two functors F and G between categories **C** and **D**. If you focus on just one object a in **C**, it is mapped to two objects: Fa and Ga. A mapping of functors should therefore map Fa to Ga.



Notice that Fa and Ga are objects in the same category **D**. Mappings between objects in the same category should not go against the grain of

the category. We don't want to make artificial connections between objects. So it's *natural* to use existing connections, namely morphisms. A natural transformation is a selection of morphisms: for every object a, it picks one morphism from Fa to Ga. If we call the natural transformation α, this morphism is called the *component* of α at a, or α_a.

$$\alpha_a :: Fa \to Ga$$

Keep in mind that a is an object in **C** while α_a is a morphism in **D**.

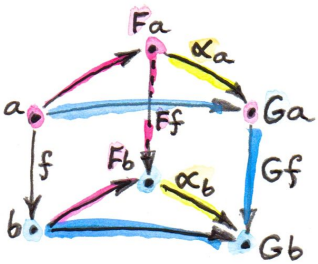
If, for some a, there is no morphism between Fa and Ga in **D**, there can be no natural transformation between F and G.

Of course that's only half of the story, because functors not only map objects, they map morphisms as well. So what does a natural transformation do with those mappings? It turns out that the mapping of morphisms is fixed — under any natural transformation between F and G, $F\ f$ must be transformed into $G\ f$. What's more, the mapping of morphisms by the two functors drastically restricts the choices we have in defining a natural transformation that's compatible with it. Consider a morphism f between two objects a and b in **C**. It's mapped to two morphisms, $F\ f$ and Gf in **D**:

$$Ff :: Fa \to Fb$$
$$Gf :: Ga \to Gb$$

The natural transformation α provides two additional morphisms that complete the diagram in D:

$$\alpha_a :: Fa \to Ga$$
$$\alpha_b :: Fb \to Gb$$

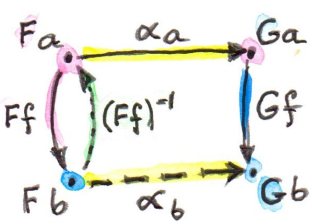


Now we have two ways of getting from Fa to Gb. To make sure that they are equal, we must impose the *naturality condition* that holds for any f:

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

The naturality condition is a pretty stringent requirement. For instance, if the morphism Ff is invertible, naturality determines α_b in terms of α_a. It *transports* α_a along f:

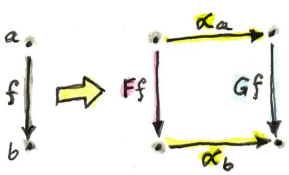
$$\alpha_b = (Gf) \circ \alpha_a \circ (Ff)^{-1}$$



If there is more than one invertible morphism between two objects, all these transports have to agree. In general, though, morphisms are not

invertible; but you can see that the existence of natural transformations between two functors is far from guaranteed. So the scarcity or the abundance of functors that are related by natural transformations may tell you a lot about the structure of categories between which they operate. We'll see some examples of that when we talk about limits and the Yoneda lemma.

Looking at a natural transformation component-wise, one may say that it maps objects to morphisms. Because of the naturality condition, one may also say that it maps morphisms to commuting squares — there is one commuting naturality square in **D** for every morphism in **C**.



This property of natural transformations comes in very handy in a lot of categorical constructions, which often include commuting diagrams. With a judicious choice of functors, a lot of these commutativity conditions may be transformed into naturality conditions. We'll see examples of that when we get to limits, colimits, and adjunctions.

Finally, natural transformations may be used to define isomorphisms of functors. Saying that two functors are naturally isomorphic is almost like saying they are the same. *Natural isomorphism* is defined as a natural transformation whose components are all isomorphisms (invertible morphisms).

10.1 Polymorphic Functions

We talked about the role of functors (or, more specifically, endofunctors) in programming. They correspond to type constructors that map types to types. They also map functions to functions, and this mapping is implemented by a higher order function **fmap** (or **transform**, **then**, and the like in C++).

To construct a natural transformation we start with an object, here a type, **a**. One functor, **F**, maps it to the type *Fa*. Another functor, **G**, maps it to *Ga*. The component of a natural transformation **alpha** at **a** is a function from *Fa* to *Ga*. In pseudo-Haskell:

```
alpha_a :: F a -> G a
```

A natural transformation is a polymorphic function that is defined for all types **a**:

```
alpha :: forall a . F a -> G a
```

The **forall a** is optional in Haskell (and in fact requires turning on the language extension **ExplicitForAll**). Normally, you would write it like this:

```
alpha :: F a -> G a
```

Keep in mind that it's really a family of functions parameterized by **a**. This is another example of the terseness of the Haskell syntax. A similar construct in C++ would be slightly more verbose:

```
template<class A> G<A> alpha(F<A>);
```

There is a more profound difference between Haskell's polymorphic functions and C++ generic functions, and it's reflected in the way these functions are implemented and type-checked. In Haskell, a polymorphic function must be defined uniformly for all types. One formula must work across all types. This is called *parametric polymorphism*.

C++, on the other hand, supports by default *ad hoc polymorphism*, which means that a template doesn't have to be well-defined for all types. Whether a template will work for a given type is decided at instantiation time, where a concrete type is substituted for the type parameter. Type checking is deferred, which unfortunately often leads to incomprehensible error messages.

In C++, there is also a mechanism for function overloading and template specialization, which allows different definitions of the same function for different types. In Haskell this functionality is provided by type classes and type families.

Haskell's parametric polymorphism has an unexpected consequence: any polymorphic function of the type:

```
alpha :: F a -> G a
```

where `F` and `G` are functors, automatically satisfies the naturality condition. Here it is in categorical notation (f is a function $f :: a \to b$):

$$G f \circ \alpha_a = \alpha_b \circ F f$$

In Haskell, the action of a functor `G` on a morphism `f` is implemented using `fmap`. I'll first write it in pseudo-Haskell, with explicit type annotations:

```
fmap_G f . alpha_a = alpha_b . fmap_F f
```

Because of type inference, these annotations are not necessary, and the following equation holds:

```
fmap f . alpha = alpha . fmap f
```

This is still not real Haskell — function equality is not expressible in code — but it's an identity that can be used by the programmer in equational reasoning; or by the compiler, to implement optimizations.

The reason why the naturality condition is automatic in Haskell has to do with "theorems for free." Parametric polymorphism, which is used to define natural transformations in Haskell, imposes very strong limitations on the implementation — one formula for all types. These limitations translate into equational theorems about such functions. In the case of functions that transform functors, free theorems are the naturality conditions.[1]

One way of thinking about functors in Haskell that I mentioned earlier is to consider them generalized containers. We can continue this analogy and consider natural transformations to be recipes for repackaging the contents of one container into another container. We are not touching the items themselves: we don't modify them, and we don't create new ones. We are just copying (some of) them, sometimes multiple times, into a new container.

The naturality condition becomes the statement that it doesn't matter whether we modify the items first, through the application of `fmap`, and repackage later; or repackage first, and then modify the items in

[1] You may read more about free theorems in my blog "Parametricity: Money for Nothing and Theorems for Free."

the new container, with its own implementation of `fmap`. These two actions, repackaging and `fmap`ping, are orthogonal. "One moves the eggs, the other boils them."

Let's see a few examples of natural transformations in Haskell. The first is between the list functor, and the `Maybe` functor. It returns the head of the list, but only if the list is non-empty:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

It's a function polymorphic in `a`. It works for any type `a`, with no limitations, so it is an example of parametric polymorphism. Therefore it is a natural transformation between the two functors. But just to convince ourselves, let's verify the naturality condition.

```
fmap f . safeHead = safeHead . fmap f
```

We have two cases to consider; an empty list:

```
fmap f (safeHead []) = fmap f Nothing = Nothing
```

```
safeHead (fmap f []) = safeHead [] = Nothing
```

and a non-empty list:

```
fmap f (safeHead (x:xs)) = fmap f (Just x) = Just (f x)
```

```
safeHead (fmap f (x:xs)) = safeHead (f x : fmap f xs) = Just (f x)
```

I used the implementation of **fmap** for lists:

```
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
```

and for **Maybe**:

```
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

An interesting case is when one of the functors is the trivial **Const** functor. A natural transformation from or to a **Const** functor looks just like a function that's either polymorphic in its return type or in its argument type.

For instance, **length** can be thought of as a natural transformation from the list functor to the **Const Int** functor:

```
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + unConst (length xs))
```

Here, **unConst** is used to peel off the **Const** constructor:

```
unConst :: Const c a -> c
unConst (Const x) = x
```

Of course, in practice **length** is defined as:

```
length :: [a] -> Int
```

which effectively hides the fact that it's a natural transformation.

Finding a parametrically polymorphic function *from* a `Const` functor is a little harder, since it would require the creation of a value from nothing. The best we can do is:

```
scam :: Const Int a -> Maybe a
scam (Const x) = Nothing
```

Another common functor that we've seen already, and which will play an important role in the Yoneda lemma, is the `Reader` functor. I will rewrite its definition as a `newtype`:

```
newtype Reader e a = Reader (e -> a)
```

It is parameterized by two types, but is (covariantly) functorial only in the second one:

```
instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))
```

For every type `e`, you can define a family of natural transformations from `Reader e` to any other functor `f`. We'll see later that the members of this family are always in one to one correspondence with the elements of `f e` (the Yoneda lemma).

For instance, consider the somewhat trivial unit type `()` with one element `()`. The functor `Reader ()` takes any type `a` and maps it into a function type `() -> a`. These are just all the functions that pick a single element from the set `a`. There are as many of these as there are elements in `a`. Now let's consider natural transformations from this functor to the `Maybe` functor:

```
alpha :: Reader () a -> Maybe a
```

There are only two of these, `dumb` and `obvious`:

```
dumb (Reader _) = Nothing
```

and

```
obvious (Reader g) = Just (g ())
```

(The only thing you can do with `g` is to apply it to the unit value `()`.)

And, indeed, as predicted by the Yoneda lemma, these correspond to the two elements of the `Maybe ()` type, which are `Nothing` and `Just ()`. We'll come back to the Yoneda lemma later — this was just a little teaser.

10.2 Beyond Naturality

A parametrically polymorphic function between two functors (including the edge case of the `Const` functor) is always a natural transformation. Since all standard algebraic data types are functors, any polymorphic function between such types is a natural transformation.

We also have function types at our disposal, and those are functorial in their return type. We can use them to build functors (like the `Reader` functor) and define natural transformations that are higher-order functions.

However, function types are not covariant in the argument type. They are *contravariant*. Of course contravariant functors are equivalent

to covariant functors from the opposite category. Polymorphic functions between two contravariant functors are still natural transformations in the categorical sense, except that they work on functors from the opposite category to Haskell types.

You might remember the example of a contravariant functor we've looked at before:

```
newtype Op r a = Op (a -> r)
```

This functor is contravariant in `a`:

```
instance Contravariant (Op r) where
    contramap f (Op g) = Op (g . f)
```

We can write a polymorphic function from, say, `Op Bool` to `Op String`:

```
predToStr (Op f) = Op (\x -> if f x then "T" else "F")
```

But since the two functors are not covariant, this is not a natural transformation in **Hask**. However, because they are both contravariant, they satisfy the "opposite" naturality condition:

```
contramap f . predToStr = predToStr . contramap f
```

Notice that the function `f` must go in the opposite direction than what you'd use with `fmap`, because of the signature of `contramap`:

```
contramap :: (b -> a) -> (Op Bool a -> Op Bool b)
```

Are there any type constructors that are not functors, whether covariant or contravariant? Here's one example:

```
a -> a
```

This is not a functor because the same type `a` is used both in the negative (contravariant) and positive (covariant) position. You can't implement `fmap` or `contramap` for this type. Therefore a function of the signature:

```
(a -> a) -> f a
```

where `f` is an arbitrary functor, cannot be a natural transformation. Interestingly, there is a generalization of natural transformations, called dinatural transformations, that deals with such cases. We'll get to them when we discuss ends.

10.3 Functor Category

Now that we have mappings between functors — natural transformations — it's only natural to ask the question whether functors form a category. And indeed they do! There is one category of functors for each pair of categories, **C** and **D**. Objects in this category are functors from **C** to **D**, and morphisms are natural transformations between those functors.

We have to define composition of two natural transformations, but that's quite easy. The components of natural transformations are morphisms, and we know how to compose morphisms.

Indeed, let's take a natural transformation α from functor F to G. Its component at object a is some morphism:

$$\alpha_a :: Fa \to Ga$$

We'd like to compose α with β, which is a natural transformation from functor G to H. The component of β at a is a morphism:

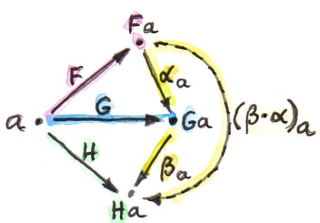
$$\beta_a :: Ga \to Ha$$

These morphisms are composable and their composition is another morphism:

$$\beta_a \circ \alpha_a :: Fa \to Ha$$

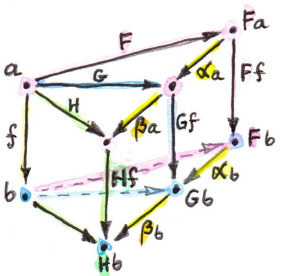
We will use this morphism as the component of the natural transformation $\beta \cdot \alpha$ — the composition of two natural transformations β after α:

$$(\beta \cdot \alpha)_a = \beta_a \circ \alpha_a$$



One (long) look at a diagram convinces us that the result of this composition is indeed a natural transformation from F to H:

$$Hf \circ (\beta \cdot \alpha)_a = (\beta \cdot \alpha)_b \circ Ff$$



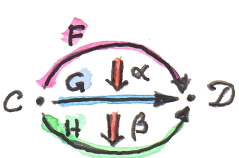
Composition of natural transformations is associative, because their components, which are regular morphisms, are associative with respect to their composition.

Finally, for each functor F there is an identity natural transformation 1_F whose components are the identity morphisms:

$$\mathbf{id}_{Fa} :: Fa \to Fa$$

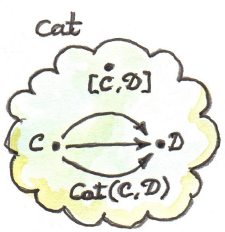
So, indeed, functors form a category.

A word about notation. Following Saunders Mac Lane I use the dot for the kind of natural transformation composition I have just described. The problem is that there are two ways of composing natural transformations. This one is called the vertical composition, because the functors are usually stacked up vertically in the diagrams that describe it. Vertical composition is important in defining the functor category. I'll explain horizontal composition shortly.



The functor category between categories **C** and **D** is written as $\mathbf{Fun}(\mathbf{C}, \mathbf{D})$, or $[\mathbf{C}, \mathbf{D}]$, or sometimes as $\mathbf{D}^{\mathbf{C}}$. This last notation suggests that a functor category itself might be considered a function object (an exponential) in some other category. Is this indeed the case?

Let's have a look at the hierarchy of abstractions that we've been building so far. We started with a category, which is a collection of objects and morphisms. Categories themselves (or, strictly speaking *small* categories, whose objects form sets) are themselves objects in a higher-level category **Cat**. Morphisms in that category are functors. A Hom-set in **Cat** is a set of functors. For instance $\mathbf{Cat}(\mathbf{C}, \mathbf{D})$ is a set of functors between two categories **C** and **D**.



A functor category $[\mathbf{C}, \mathbf{D}]$ is also a set of functors between two categories (plus natural transformations as morphisms). Its objects are the same as the members of $\mathbf{Cat}(\mathbf{C}, \mathbf{D})$. Moreover, a functor category, being a category, must itself be an object of **Cat** (it so happens that the functor category between two small categories is itself small). We have a relationship between a Hom-set in a category and an object in the same category. The situation is exactly like the exponential object that we've seen in the last section. Let's see how we can construct the latter in **Cat**.

As you may remember, in order to construct an exponential, we need to first define a product. In **Cat**, this turns out to be relatively easy,

because small categories are *sets* of objects, and we know how to define Cartesian products of sets. So an object in a product category $\mathbf{C} \times \mathbf{D}$ is just a pair of objects, (c, d), one from $\mathbf{C}$ and one from $\mathbf{D}$. Similarly, a morphism between two such pairs, (c, d) and (c', d'), is a pair of morphisms, (f, g), where $f :: c \to c'$ and $g :: d \to d'$. These pairs of morphisms compose component-wise, and there is always an identity pair that is just a pair of identity morphisms. To make the long story short, **Cat** is a full-blown Cartesian closed category in which there is an exponential object $\mathbf{D}^{\mathbf{C}}$ for any pair of categories. And by "object" in **Cat** I mean a category, so $\mathbf{D}^{\mathbf{C}}$ is a category, which we can identify with the functor category between $\mathbf{C}$ and $\mathbf{D}$.

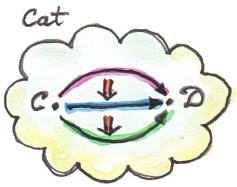
10.4 2-Categories

With that out of the way, let's have a closer look at **Cat**. By definition, any Hom-set in **Cat** is a set of functors. But, as we have seen, functors between two objects have a richer structure than just a set. They form a category, with natural transformations acting as morphisms. Since functors are considered morphisms in **Cat**, natural transformations are morphisms between morphisms.

This richer structure is an example of a **2**-category, a generalization of a category where, besides objects and morphisms (which might be called 1-morphisms in this context), there are also 2-morphisms, which are morphisms between morphisms.

In the case of **Cat** seen as a **2**-category we have:

- Objects: (Small) categories
- 1-morphisms: Functors between categories
- 2-morphisms: Natural transformations between functors.



Instead of a Hom-set between two categories **C** and **D**, we have a Hom-category — the functor category $\mathbf{D}^{\mathbf{C}}$. We have regular functor composition: a functor F from $\mathbf{D}^{\mathbf{C}}$ composes with a functor G from $\mathbf{E}^{\mathbf{D}}$ to give $G \circ F$ from $\mathbf{E}^{\mathbf{C}}$. But we also have composition inside each Hom-category — vertical composition of natural transformations, or 2-morphisms, between functors.

With two kinds of composition in a 2-category, the question arises: How do they interact with each other?

Let's pick two functors, or 1-morphisms, in **Cat**:

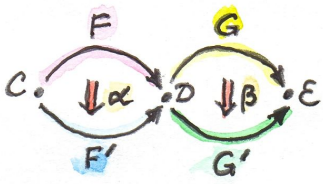
$$F :: \mathbf{C} \to \mathbf{D}$$
$$G :: \mathbf{D} \to \mathbf{E}$$

and their composition:

$$G \circ F :: \mathbf{C} \to \mathbf{E}$$

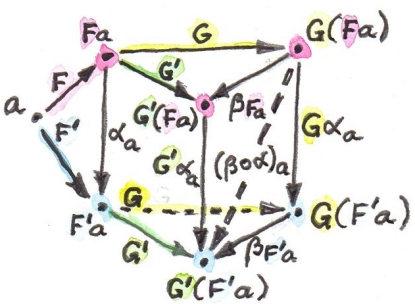
Suppose we have two natural transformations, α and β, that act, respectively, on functors F and G:

$$\alpha :: F \to F'$$
$$\beta :: G \to G'$$



Notice that we cannot apply vertical composition to this pair, because the target of α is different from the source of β. In fact they are members of two different functor categories: $\mathbf{D}^{\mathbf{C}}$ and $\mathbf{E}^{\mathbf{D}}$. We can, however, apply composition to the functors F' and G', because the target of F' is the source of G' — it's the category $\mathbf{D}$. What's the relation between the functors $G' \circ F'$ and $G \circ F$?

Having α and β at our disposal, can we define a natural transformation from $G \circ F$ to $G' \circ F'$? Let me sketch the construction.



As usual, we start with an object a in $\mathbf{C}$. Its image splits into two objects in $\mathbf{D}$: Fa and $F'a$. There is also a morphism, a component of α, connecting these two objects:

$$\alpha_a :: Fa \to F'a$$

When going from **D** to **E**, these two objects split further into four objects: $G(Fa)$, $G'(Fa)$, $G(F'a)$, $G'(F'a)$. We also have four morphisms forming a square. Two of these morphisms are the components of the natural transformation β:

$$\beta_{Fa} :: G(Fa) \to G'(Fa)$$
$$\beta_{F'a} :: G(F'a) \to G'(F'a)$$

The other two are the images of α_a under the two functors (functors map morphisms):

$$G\alpha_a :: G(Fa) \to G(F'a)$$
$$G'\alpha_a :: G'(Fa) \to G'(F'a)$$

That's a lot of morphisms. Our goal is to find a morphism that goes from $G(Fa)$ to $G'(F'a)$, a candidate for the component of a natural transformation connecting the two functors $G \circ F$ and $G' \circ F'$. In fact there's not one but two paths we can take from $G(Fa)$ to $G'(F'a)$:

$$G'\alpha_a \circ \beta_{Fa}$$
$$\beta_{F'a} \circ G\alpha_a$$

Luckily for us, they are equal, because the square we have formed turns out to be the naturality square for β.

We have just defined a component of a natural transformation from $G \circ F$ to $G' \circ F'$. The proof of naturality for this transformation is pretty straightforward, provided you have enough patience.

We call this natural transformation the *horizontal composition* of α and β:

$$\beta \circ \alpha :: G \circ F \to G' \circ F'$$

Again, following Mac Lane I use the small circle for horizontal composition, although you may also encounter star in its place.

Here's a categorical rule of thumb: Every time you have composition, you should look for a category. We have vertical composition of natural transformations, and it's part of the functor category. But what about the horizontal composition? What category does that live in?

The way to figure this out is to look at **Cat** sideways. Look at natural transformations not as arrows between functors but as arrows between categories. A natural transformation sits between two categories, the ones that are connected by the functors it transforms. We can think of it as connecting these two categories.

Let's focus on two objects of **Cat** — categories **C** and **D**. There is a set of natural transformations that go between functors that connect **C** to **D**. These natural transformations are our new arrows from **C** to **D**. By the same token, there are natural transformations going between functors that connect **D** to **E**, which we can treat as new arrows going from **D** to **E**. Horizontal composition is the composition of these arrows.

We also have an identity arrow going from **C** to **C**. It's the identity natural transformation that maps the identity functor on **C** to itself. Notice that the identity for horizontal composition is also the identity for vertical composition, but not vice versa.

Finally, the two compositions satisfy the interchange law:

$$(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$$

I will quote Saunders Mac Lane here: The reader may enjoy writing down the evident diagrams needed to prove this fact.

There is one more piece of notation that might come in handy in the future. In this new sideways interpretation of **Cat** there are two ways of getting from object to object: using a functor or using a natural transformation. We can, however, re-interpret the functor arrow as a special kind of natural transformation: the identity natural transformation acting on this functor. So you'll often see this notation:

$$F \circ \alpha$$

where F is a functor from **D** to **E**, and α is a natural transformation between two functors going from **C** to **D**. Since you can't compose a functor with a natural transformation, this is interpreted as a horizontal composition of the identity natural transformation 1_F after α.

Similarly:

$$\alpha \circ F$$

is a horizontal composition of α after 1_F.

10.5 Conclusion

This concludes the first part of the book. We've learned the basic vocabulary of category theory. You may think of objects and categories as nouns; and morphisms, functors, and natural transformations as verbs. Morphisms connect objects, functors connect categories, natural transformations connect functors.

But we've also seen that, what appears as an action at one level of abstraction, becomes an object at the next level. A set of morphisms turns into a function object. As an object, it can be a source or a target of another morphism. That's the idea behind higher order functions.

A functor maps objects to objects, so we can use it as a type constructor, or a parametric type. A functor also maps morphisms, so it is a higher order function — `fmap`. There are some simple functors, like `Const`, product, and coproduct, that can be used to generate a large variety of algebraic data types. Function types are also functorial, both covariant and contravariant, and can be used to extend algebraic data types.

Functors may be looked upon as objects in the functor category. As such, they become sources and targets of morphisms: natural transformations. A natural transformation is a special type of polymorphic function.

10.6 Challenges

1. Define a natural transformation from the `Maybe` functor to the list functor. Prove the naturality condition for it.
2. Define at least two different natural transformations between `Reader ()` and the list functor. How many different lists of `()` are there?
3. Continue the previous exercise with `Reader Bool` and `Maybe`.
4. Show that horizontal composition of natural transformation satisfies the naturality condition (hint: use components). It's a good exercise in diagram chasing.
5. Write a short essay about how you may enjoy writing down the evident diagrams needed to prove the interchange law.
6. Create a few test cases for the opposite naturality condition of transformations between different `Op` functors. Here's one choice:

```
op :: Op Bool Int
op = Op (\x -> x > 0)
```

and

```
f :: String -> Int
f x = read x
```

Part Two