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.

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

[1] Ignoring "size" issues.

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

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 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.

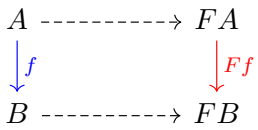
In fact, it's easier to imagine a mapping *between* categories. Every object in the source category is mapped to an object in the target category. If A is an object in $\mathcal{C}$, there is a corrsponding object FA in $\mathcal{D}$.

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

$$f : A \to B$$

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

$$Ff : FA \to FB$$



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.

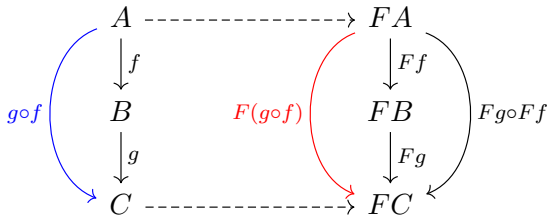
The structure of a category is defined by arrows and their composition. Therefore a functor must preserve composition. What is composed in one category:

$$h = g \circ f$$

should remain composed in the second category:

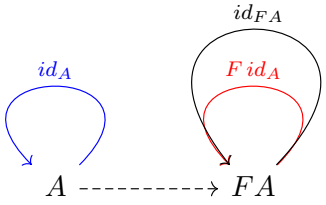
$$Fh = F(g \circ f) = Fg \circ Ff$$

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.

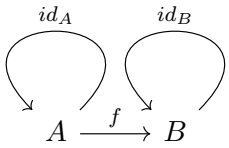
A *constant functor* Δ_C is an example of a functor that maps all objects from the source category to a single object C in the target category, and all arrows from the source category to a single identity arrow id_C.

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 $\mathcal{C}^{op} \times \mathcal{C}$ to $\mathcal{C}$. Functors from $\mathcal{C}^{op} \times \mathcal{C}$ 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.