Chapter 6

Function Types

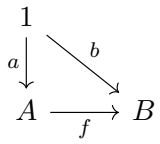
There is another kind of composition that is at the heart of functional programming. It happens when you pass a function as an argument to another function. The outer function can then use this argument as a pluggable part of its own machinery. It lets you implement, for instance, a generic sorting algorithm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

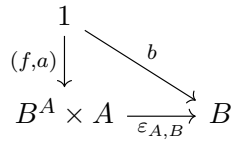
$$b \colon 1 \to B$$

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)

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

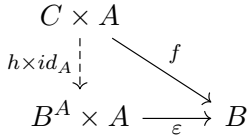
We can now follow this with function application to get to B

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

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*:

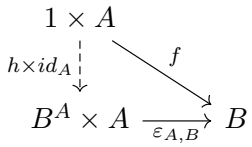
```
curry   :: ((c, a) -> b)   -> (c -> (a -> b))
```

```
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 consider C 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

$$ax^2 + bx + c$$

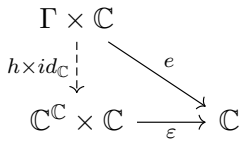
You may think of it as being parameterized by a triple of real numbers (a, b, c) and a variable x, taken to be, let's say, a complex number. The triple is an element of a product $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$. This product is the environment Γ for our expression.

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:

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

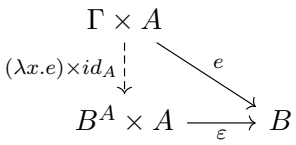
or, in Haskell, with the backslash replacing λ,

```
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

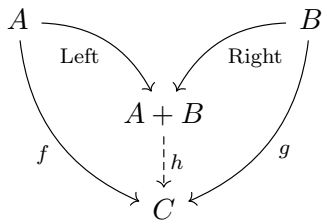
In logic, the function object corresponds to an implication. An arrow from the terminal object to the function object is the proof of an implication. Function application ε corresponds to what logicians call *modus ponens*: if you have a proof of the implication $A \Rightarrow B$ and a proof of A then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

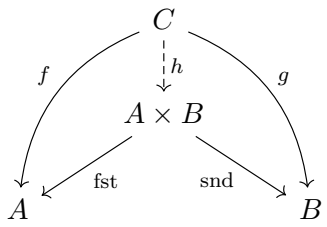
```
either                  :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)     =  f x
either _ g (Right y)    =  g y
```

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:

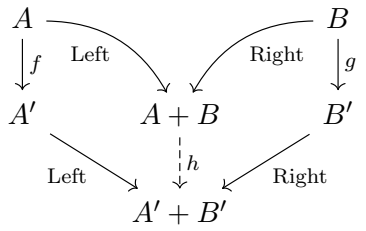
```
fork :: (c -> (a, b)) -> (c -> a, c -> b)
fork h = (fst . h, snd . h)
```

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:



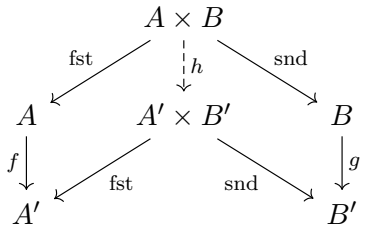
Reading this diagram we can immediately write h using `either`:

```
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`.

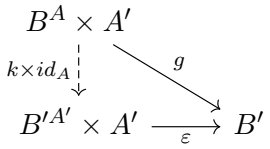
6.2 Functoriality of the Function Type

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.*