Why category theory matters for functional programmers

Eric Kidd

March 17, 2007

DRAFT: Please do not redistribute or post to aggregators quite yet—I want to finish it first!

At Dan Friedman's 60th birthday, Jonathan Sobel argued that computer scientists should understand category theory:

> "Turing machines can serve as a foundation for all of computing, [as can lambda calculus]. Anything one can do, the other can do. But they feel it different. [With Turing machines,] your program is just this giant state-transition rule with no structure to it. Whereas lambda calculus actually has the notion of functions. It's less primitive, but just as foundational."
>
> "Set theory is to Turing machines as category theory is to the lambda calculus." (from "Implementing Categorical Semantics")

I'd like to push Sobel's argument one step further: Everybody who programs in a functional language should know a bit of category theory.

1 Objects and arrows

In Haskell, we can convert a character to an ASCII value using the `ord` function:

```
ord 'x' == 120
```

We can declare the type of `ord` as follows:

```
ord :: Char -> Int
```

This would be read as "`ord` is a function mapping a `Char` to an `Int`." We can then map that integer to a Boolean value using `even`:

```
even 120 == True
```

The type declaration for `even` is similar to that of `ord`:

```
even :: Int -> Bool
```

Now that we have two functions, we're ready to do a bit of category theory. If you've ever seen a category theory textbook, you will have noticed a certain fondness for diagrams. And sure enough, we can draw our two type declarations as follows:

$$Char \xrightarrow{ord} Int \qquad\qquad Int \xrightarrow{even} Bool$$

In this diagram, `Char` is called an **object** and `ord` is called an **arrow**. This gives us the following relationships between category theory and functional programming:

Category theory	**Programming**
object	type
arrow	function

In Haskell, we can compose two functions together using the "`.`" operator to produce a new function.

```
even (ord 'x')   == True
(even . ord) 'x' == True
```

A composed function has the obvious type:

```
(even . ord) :: Char -> Bool
```

We can add this new function to our categorical diagram:

$$Char \xrightarrow{ord} Int \xrightarrow{even} Bool \quad (Char \xrightarrow{even \circ ord} Bool)$$

Note that some authors prefer to reverse $even \circ ord$, and instead write $ord;\ even$. The latter style "flows" in the same direction as the arrows. But for consistency with Haskell, we'll use the former style in this paper.

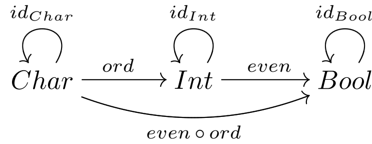
We'll need one more idea from Haskell: the identity function. Given a value of any type, `id` returns it unchanged.

```
id 'x'  == 'x'
id 120  == 120
id True == True
```

Because `id` is defined for every type, we need to use a type variable `a` in its declaration.

```
id :: a -> a
```

How do we add `id` to our diagram? We need to create one new arrow for each type, labelling it appropriately:



A **category** is any collection of objects and arrows such that

1. Every object has an identity arrow, and
2. Every path of two or more arrows has a corresponding "shortcut"—an arrow which goes straight to the ultimate destination with no stops along the way.

2 Functors

If functions are arrows, and types are objects, what are polymorphic types? In Haskell, a polymorphic type is any type which contains a variable:

```
data [a] = [] | a : [a]
```

This declaration can be read, "A list of type `a` is either the empty list, or a value of type `a` followed by a list of type `[a]`." But until we know what `a` is, we don't really have a type. Instead, we have a mechanism for turning any type `a` into the corresponding list type. This works a lot like a function at the type level, where `*` represents a concrete type:

```
[] :: * -> *
```

We can do something similar with functions. Any function `g` of type `a -> b` can be turned into a function `map g` of type `[a] -> [b]` by applying `g` to each element of the list:

```
map :: (a -> b) -> ([a] -> [b])
map g []     = []
map g (x:xs) = g x : map g xs
```

For example, we can apply `map` to `even`, yielding a function which works on lists:

```
(map even) :: [Int] -> [Bool]
```

Using `map even`, we can transform a list of integers to a list of Boolean values:

```
map even [1,2,3] == [False,True,False]
```

But `map` isn't limited to lists. In fact, we can define a version of `map` for any polymorphic type. For example, `Maybe a` represents a value which might or might not be present:

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

Our `map` function checks to see whether a value is present, and if so, applies `g` to it:

```
mapMaybe :: (a -> b) -> (Maybe a -> Maybe b)
mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)
```

Obviously, it would be nice to generalize this pattern a bit. We have some polymorphic type `f`, and a matching version of `map`. Using Haskell's type classes, we can specify how the two are related:

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

instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap = mapMaybe
```

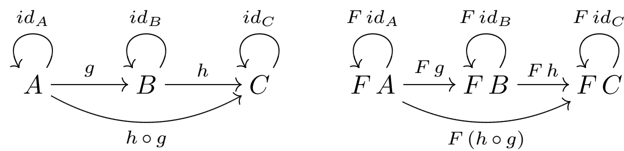
A `Functor` is a function at the type level, mapping one concrete type to another. It also provides support for "upgrading" functions to the new type.

Using these declarations, we can write code which works with both our functors.

```
fmap even [1,2,3]  == [False,True,False]
fmap even (Just 2) == Just True
```

What do functors look like in category theory? They're almost identical, except for the notation. In particular, we replace `fmap` with the type name. Instead of writing `mapMaybe even`, we write $Maybe\ even$, and rely on context to tell types and functions apart.

We can take our earlier diagram of a category, and apply `F` to every object and arrow:



The diagram on the right is *almost* a category. We need two more identities:

```
fmap id == id
fmap (h . g) == fmap h . fmap g
```

If we apply these to the right-hand diagram, we get:

$$F\,A \xrightarrow{F\,g} F\,B \xrightarrow{F\,h} F\,C \qquad id_{F\,A},\ id_{F\,B},\ id_{F\,C},\quad F\,h \circ F\,g : F\,A \to F\,C$$

This new diagram is once again a category, with objects of the form $F\,A$ and arrows of the form $F\,g$! So a functor is actually a transformation between two categories, or a transformation from a category onto a smaller part of itself.

Let's look at one more functor before proceeding. We can transform any object onto itself using the functor Id, where $Id\,A = A$ and $Id\,g = g$. In Haskell, we can represent Id as:

```
type Id a = a
fmapId g = g
```

Unfortunately, we can't make `Id` an instance of `Functor` because of limitations in Haskell's type system.

3 Natural transformations

There's one interesting property of `fmap` that we haven't investigated yet. It commutes with certain functions. For example, it doesn't matter whether we apply `fmap even` before or after we reverse the order of a list. We always get the same result:

```
fmap even . reverse == reverse . fmap even
```

This equation should catch our eye, because `fmap even` and `reverse` are both arrows. So what we're seeing here is a diagram with two equivalent paths:

$$\begin{array}{ccc} L\,Int & \xrightarrow{reverse_{Int}} & L\,Int \\ \downarrow{\scriptstyle L\,even} & & \downarrow{\scriptstyle L\,even} \\ L\,Bool & \xrightarrow[reverse_{Bool}]{} & L\,Bool \end{array}$$

We can find other functions with similar diagrams. For example, `head` returns the first element of a list:

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

As before, it doesn't matter whether we call `head` before or after calling `even`.

```
even . head == head . fmap even
```

But this isn't quite like our previous example—`fmap` only appears on one side of the equation. We can make it fit the pattern better using `fmapId`:

```
fmapId even . head == head . fmap even
```

This gives us the following diagram:

$$\begin{array}{ccc} L\,Int & \xrightarrow{head_{Int}} & Id\,Int \\ \downarrow{\scriptstyle L\,even} & & \downarrow{\scriptstyle Id\,even} \\ L\,Bool & \xrightarrow[head_{Bool}]{} & Id\,Bool \end{array}$$

Now that we have two diagrams, we can generalize them by replacing the object and arrow names with variables:

$$\begin{array}{ccc} F\,A & \xrightarrow{\alpha_A} & G\,A \\ \downarrow{\scriptstyle F\,h} & & \downarrow{\scriptstyle G\,h} \\ F\,B & \xrightarrow[\alpha_B]{} & G\,B \end{array}$$

If this diagram is valid for every function $h: A \rightarrow B$, then we call α a **natural transformation**. As it turns out, any polymorphic function with a single type variable `a` is automatically a natural transformation. The tricky part is figuring out what our functors `F` and `G` should be!

As Wadler showed in "Theorems for Free," natural transformations can be used to optimize functional programs [**?**]. Every time we see a polymorphic function next to `fmap` (which happens surprisingly often), we can prove certain optimizations safe with help from our diagrams.

Much day-to-day work in category theory has this flavor: We look for common patterns in diagrams, and see if we can use those patterns to solve whole families of problems at once.

4 Duals

Cite: Comprehending Queries.

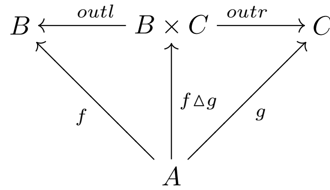
```
outl (x,y) = x
outr (x,y) = y

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

(even `split` chr) 120 == (True,'x')
```

As usual, we've discarded the values.

```
f == outl . (f ‘split‘ g)
g == outr . (f ‘split‘ g)
```

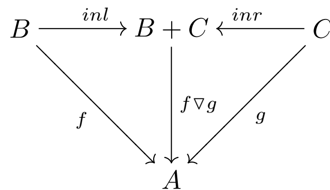



```
data Either a b = Left a | Right b

inl x = Left  x
inr x = Right x

junc :: (b -> a) -> (c -> a) -> (Either b c -> a)
(f ‘junc‘ g) (Left  x) = f x
(f ‘junc‘ g) (Right x) = g x

((==0) ‘junc‘ null) (Left  2)  == False
((==0) ‘junc‘ null) (Right []) == True
```



What are $\times$ and $+$? Bifunctors!

5 Further reading

6 Conclusion

Category theory	**Programming**
object	type
arrow	function
functor	polymorphic type
natural transformation	polymorphic function