Functors

CIS 194 Week 9
18 March 2013

Suggested reading:

- Learn You a Haskell, **The Functor typeclass**
- **The Typeclassopedia**

Motivation

Over the past weeks we have seen a number of functions designed to "map" a function over every element of some sort of container. For example:

- `map :: (a -> b) -> [a] -> [b]`
- `treeMap :: (a -> b) -> Tree a -> Tree b`
- In Homework 5 many people ended up doing a similar thing when you had to somehow apply `eval :: ExprT -> Int` to a `Maybe ExprT` in order to get a `Maybe Int`.

  ```
  maybeEval :: (ExprT -> Int) -> Maybe ExprT -> Maybe Int

  maybeMap :: (a -> b) -> Maybe a -> Maybe b
  ```

There's a repeated pattern here, and as good Haskell programmers we want to know how to generalize it! So which parts are the same from example to example, and which parts are different?

The part that is different, of course, is the container being "mapped over":

```
thingMap :: (a -> b) -> f a -> f b
```

But what sort of things are these "containers"? Can we really assign a type variable like `f` to them?

A brief digression on kinds

Just as every expression has a type, types themselves have "types", called *kinds*. (Before you ask: no, there's not another level beyond kinds—not in Haskell at least.) In `ghci` we can ask about the kinds of types using `:kind`. For example, let's ask for the kind of `Int`:

```
Prelude> :k Int
Int :: *
```

We see that `Int` has kind `*`. In fact, every type which can actually serve as the type of some values has kind `*`.

```
Prelude> :k Bool
Bool :: *
Prelude> :k Char
Char :: *
Prelude> :k Maybe Int
Maybe Int :: *
```

If `Maybe Int` has kind `*`, then what about `Maybe`? Notice that there are no values of type `Maybe`. There are values of type `Maybe Int`, and of type `Maybe Bool`, but not of type `Maybe`. But `Maybe` is certainly a valid type-like-thing. So what is it? What kind does it have? Let's ask `ghci`:

```
Prelude> :k Maybe
Maybe :: * -> *
```

`ghci` tells us that `Maybe` has kind `* -> *`. `Maybe` is, in a sense, a *function on types* — we usually call it a *type constructor*. `Maybe` takes as input types of kind `*`, and produces another type of kind `*`. For example, it can take as input `Int :: *` and produce the new type `Maybe Int :: *`.

Are there other type constructors with kind `* -> *`? Sure. For example, `Tree`, or the list type constructor, written `[]`.

```
Prelude> :k []
[] :: * -> *
Prelude :k [] Int
[] Int :: *
Prelude> :k [Int]  -- special syntax for [] Int
[Int] :: *
Prelude> :k Tree
Tree :: * -> *
```

What about type constructors with other kinds? How about `JoinList` from Homework 7?

```
data JoinList m a = Empty
                  | Single m a
                  | Append m (JoinList m a) (JoinList m a)

Prelude> :k JoinList
JoinList :: * -> * -> *
```

This makes sense: `JoinList` expects *two* types as parameters and gives us back a new type. (Of course, it is curried, so we can also think of it as taking *one* type and giving back something of kind `* -> *`.) Here's another one:

```
Prelude> :k (->)
(->) :: * -> * -> *
```

Tthis tells us that the function type constructor takes two type arguments. Like any operator, we use it infix:

```
Prelude> :k Int -> Char
Int -> Char :: *
```

But we don't have to:

```
Prelude> :k (->) Int Char
(->) Int Char :: *
```

OK, what about this one?

```
data Funny f a = Funny a (f a)

Prelude> :k Funny
Funny :: (* -> *) -> * -> *
```

`Funny` takes two arguments, the first one a type of kind `* -> *`, and the second a type of kind `*`, and constructs a type. (How did GHCi know what the kind of `Funny` is? Well, it does *kind inference* just like it also does *type inference*.) `Funny` is a *higher-order* type constructor, in the same way that `map` is a *higher-order* function. Note that types can be partially applied too, just like functions:

```
Prelude> :k Funny Maybe
Funny Maybe :: * -> *
Prelude> :k Funny Maybe Int
Funny Maybe Int :: *
```

Functor

The essence of the mapping pattern we saw was a higher-order function with a type like

```
thingMap :: (a -> b) -> f a -> f b
```

where `f` is a type variable standing in for some type of kind `* -> *`. So, can we write a function of this type once and for all?

```
thingMap :: (a -> b) -> f a -> f b
thingMap h fa = ???
```

Well, not really. There's not much we can do if we don't know what `f` is. `thingMap` has to work differently for each particular `f`. The solution is to make a type class, which is traditionally called `Functor`:

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

(`Functor` is defined in the standard Prelude. Note that the name "functor" comes from category theory, and is *not* the same thing as functors in C++ (which are essentially first-class functions).) Now we can just implement this class in a way specific to each particular `f`. Note that the `Functor` class abstracts over types of kind `* -> *`. So it would make no sense to write

```
instance Functor Int where
  fmap = ...
```

wrong kind

Indeed, if we try, we get a very nice *kind mismatch error*:

```
[1 of 1] Compiling Main             ( 09-functors.lhs, interpreted )

09-functors.lhs:145:19:
    Kind mis-match
    The first argument of `Functor' should have kind `* -> *',
    but `Int' has kind `*'
    In the instance declaration for `Functor Int'
```

If we understand kinds, this error tells us exactly what is wrong.

However, it does make sense (kind-wise) to make a `Functor` instance for, say, `Maybe`. Let's do it. Following the types makes it almost trivial:

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

How about lists?

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs
  -- or just
  -- fmap = map
```

Easy peasy. What about `IO`? Could it make sense to create an instance of `Functor` for `IO`?

Sure. `fmap :: (a -> b) -> IO a -> IO b` results in the IO action which first runs the `IO a` action, then applies the function to transform the result before returning it. We can implement this without too much trouble:

```
instance Functor IO where
  fmap f ioa = ioa >>= (\a -> return (f a))
```

or even

```
instance Functor IO where
  fmap f ioa = ioa >>= (return . f)
```

Now let's try something a bit more mind-twisting:

```
instance Functor ((->) e) where
```

What!? Well, let's follow the types: if `f = (->) e` then we want

```
fmap :: (a -> b) -> (->) e a -> (->) e b
```

or, with `(->)` written infix:

```
fmap :: (a -> b) -> (e -> a) -> (e -> b)
```

Hmm, this type signature seems familiar…

```
instance Functor ((->) e) where
  fmap = (.)
```

Crazy! What does this mean? Well, one way to think of a value of type `(e -> a)` is as a "e-indexed container" with one value of `a` for each value of `e`. To map a function over every value in such a container corresponds

exactly to function composition: to pick an element out of the transformed container we first we apply the `(e -> a)` function to pick out an `a` from the original container, and then apply the `(a -> b)` function to transform the element we picked.

Generated 2013-03-21 14:41:58.488748