Haskell/The Functor class

In this chapter, we will introduce the important `Functor` type class.

Motivation

In Other data structures, we saw operations that apply to all elements of some grouped value. The prime example is `map` which works on lists. Another example we worked through was the following Tree datatype:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving (Show)
```

The map function we wrote for Tree was:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f (Leaf x) = Leaf (f x)
treeMap f (Branch left right) = Branch (treeMap f left) (treeMap f right)
```

As discussed before, we can conceivably define a map-style function for any arbitrary data structure.

When we first introduced `map` in Lists II, we went through the process of taking a very specific function for list elements and generalizing to show how `map` combines any appropriate function with all sorts of lists. Now, we will generalize still *further*. Instead of map-for-lists and map-for-trees and other distinct maps, how about a general concept of maps for all sorts of mappable types?

Introducing `Functor`

`Functor` is a Prelude class for types which can be mapped over. It has a single method, called `fmap`. The class is defined as follows:

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

The usage of the type variable `f` can look a little strange at first. Here, `f` is a parametrized data type; in the signature of `fmap`, `f` takes `a` as a type parameter in one of its appearances and `b` in the other. Let's consider an instance of `Functor`: By replacing `f` with `Maybe` we get the following signature for `fmap`...

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

... which fits the natural definition:

```
instance  Functor Maybe  where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)
```

(Incidentally, this definition is in Prelude; so we didn't really need to implement `maybeMap` for that example in the "Other data structures" chapter.)

The `Functor` instance for lists (also in Prelude) is simple:

```
instance  Functor []  where
    fmap = map
```

... and if we replace `f` with `[]` in the `fmap` signature, we get the familiar type of `map`.

So, `fmap` is a generalization of `map` for any parametrized data type.[1]

Naturally, we can provide `Functor` instances for our own data types. In particular, `treeMap` can be promptly relocated to an instance:

```
instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch left right) = Branch (fmap f left) (fmap f right)
```

Here's a quick demo of `fmap` in action with the instances above (to reproduce it, you only need to load the `data` and `instance` declarations for `Tree`; the others are already in Prelude):

```
*Main> fmap (2*) [1,2,3,4]
[2,4,6,8]
*Main> fmap (2*) (Just 1)
Just 2
*Main> fmap (fmap (2*)) [Just 1, Just 2, Just 3, Nothing]
[Just 2, Just 4, Just 6, Nothing]
*Main> fmap (2*) (Branch (Branch (Leaf 1) (Leaf 2)) (Branch (Leaf 3) (Leaf 4)))
Branch (Branch (Leaf 2) (Leaf 4)) (Branch (Leaf 6) (Leaf 8))
```

> *Note*
>
> Beyond `[]` and `Maybe`, there are many other `Functor` instances already defined. Those made available from the

Prelude can are listed in the Data.Functor (http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-Functor.html) module.

The functor laws

When providing a new instance of `Functor`, you should ensure it satisfies the two functor laws. There is nothing mysterious about these laws; their role is to guarantee `fmap` behaves sanely and actually performs a mapping operation (as opposed to some other nonsense). [2] The first law is:

```
fmap id  =  id
```

`id` is the identity function, which returns its argument unaltered. The first law states that mapping `id` over a functorial value must return the functorial value unchanged.

Next, the second law:

```
fmap (g . f)  =  fmap g . fmap f
```

It states that it should not matter whether we map a composed function or first map one function and then the other (assuming the application order remains the same in both cases).

What did we gain?

At this point, we can ask what benefit we get from the extra layer of generalization brought by the `Functor` class. There are two significant advantages:

- The availability of the `fmap` method relieves us from having to recall, read, and write a plethora of differently named mapping methods (`maybeMap`, `treeMap`, `weirdMap`, *ad infinitum*). As a consequence, code becomes both cleaner and easier to understand. On spotting a use of `fmap`, we instantly have a general idea of what is going on.[3] Thanks to the guarantees given by the functor laws, this general idea is surprisingly precise.
- Using the type class system, we can write `fmap`-based algorithms which work out of the box with *any* functor - be it `[]`, `Maybe`, `Tree` or whichever you need. Indeed, a number of useful classes in the core libraries inherit from `Functor`.

Type classes make it possible to create general solutions to whole categories of problems. Depending on what you use Haskell for, you may not need to define new classes often, but you will certainly be *using* type classes all the time. Many of the most powerful features and sophisticated capabilities of Haskell rely on type classes (residing either in the standard libraries or elsewhere). From this point on, classes will be a prominent presence in our studies.

Notes

1. Data structures provide the most intuitive examples; however, there are functors which cannot reasonably be seen as data structures. A commonplace metaphor consists in thinking of functors as containers; like all metaphors, however, it can be stretched only so far.
2. Some examples of nonsense that the laws rule out: removing or adding elements from a list, reversing a list, changing a `Just`-value into a `Nothing`.
3. This is analogous to the gain in clarity provided by replacing explicit recursive algorithms on lists with implementations based on higher-order functions.