CSC324 Lecture 20

Last time

We introduced **functors**, a polymorphic typeclass that implements a `fmap()` function

We generalised the idea of functors as "container data structures" to **effects being applied** within **computational contexts.**

Today:

- Formalizing the Maybe type as a functor
- The Either functor
- (if we have time): composing operations on functors

But first, a typo on Friday's slides

I told you an incorrect Functor law, which got copied and pasted to a bunch of slides from Friday.

The first functor law, actually, is: There must exist a **function `id :: a->a`** such that the functor is not transformed when it is mapped with this function.

```
fmap id x = id x          -- if we fmap id over a functor, it is the
                          -- same as just calling id on the functor.
```

But first, a typo on Friday's slides

I told you an incorrect Functor law, which got copied and pasted to a bunch of slides from Friday.

The first functor law, actually, is: There must exist a **function `id :: a->a`** such that the functor is not transformed when it is mapped with this function.

```
fmap id x = id x

fmap id = id  -- the curryed version, ahh, how nice
```

But first, a typo on Friday's slides

I told you an incorrect Functor law, which got copied and pasted to a bunch of slides from Friday.

The first functor law, actually, is: There must exist a **function `id :: a->a`** such that the functor is not transformed when it is mapped with this function.

```
fmap id x = id x
```

```
fmap id = id  -- the curryed version, ahh, how nice
```

The slides are fixed, but of course the recording has `id` be the second argument to map. Sorry.

Call me Maybe

The built-in option type in Haskell is called Maybe. It is defined as:

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

(to worked example)

What's optional in the type?

We saw two possibilities of implementing divAllBy:

- One implementation produces a `Maybe [Num]`

- One implementation produces a `[Maybe Num]`.

What can we say about the differences between these types?

What's optional in the type?

We saw two possibilities of implementing divAllBy:

- One implementation produces a `Maybe [Num]`
  "The return value is the list of divisions, unless any of them would have failed; in which case, we produce a `Nothing`"

- One implementation produces a `[Maybe Num]`.

What can we say about the differences between these types?

What's optional in the type?

We saw two possibilities of implementing divAllBy:

- One implementation produces a `Maybe [Num]`
  "The return value is the list of divisions, unless any of them would have failed; in which case, we produce a `Nothing`"

- One implementation produces a `[Maybe Num]`.
  "The return value is the list of, for each number, whether that division was successful or not"

What can we say about the differences between these types?

What's optional in the type?

We saw two possibilities of implementing divAllBy:

One implementation produces a `Maybe [Num]`

- If any number would have caused a division by zero, the whole expr is `Nothing`; if we get a `Just` back, we know every element in the input list was divided correctly.
- This no longer works lazily: the entire `[Num]` list must be materialised before we can say if it should be a `Just` of some list, or `Nothing`

What's optional in the type?

We saw two possibilities of implementing divAllBy:

One implementation produces a `[Maybe Num]`.

- We are guaranteed to get a list of values back, such that the length of the input list equals the length of the output list (so this works with infinite lists, too!)
- Each element in the list needs to be "unpacked" if it's a `Just` of some `Num`.

Divining a `fmap` from the data definition

Recall the type signature for Maybe's fmap:

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

Imagine that we hadn't seen how fmap() behaves, and had no idea what it *should* do. We only have the type signature and the Functor laws.

Can we figure out what it should do from these things alone?

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f m = ...
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's fmap:

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

We know that as Maybe is a sum type, we need to handle each data constructor independently. Let's do that in the usual way, with pattern matching:

The built-in option type in Haskell is called `Maybe`. It is defined as:

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

```
fmap f Nothing = …
fmap f Just x = …
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's map:

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

What can we say about what we can do on the right hand side of the first pattern match? We know it needs to be a `Maybe b`, but...

- We have nothing of type `b`…
- We have no `a` to turn into a `b` with `f`…
- Our only choice is to produce a Nothing!

The built-in option type in Haskell is called `Maybe`. It is defined as:

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

```
fmap f Nothing = …
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's map:

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

Could we do the same thing as before, where we simply produce a **Nothing** again?

This violates the first Functor law!
(recall: `map id = id`)

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Nothing
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's map:

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

What can we say about what map needs to do when we are mapping over a `Just` of something?

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = ...
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's map:

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

Could we do the same thing as before, where we simply produce a **Nothing** again?

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Nothing
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's map:

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

Could we do the same thing as before, where we simply produce a **Nothing** again?

This violates the first Functor law!
(recall: `fmap id = id`)
So this has to be a Just of something….

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Just ...
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's map:

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

We play the same game as before:

- We have nothing of type b …
- but we DO have x, of type a…
- and a function `f :: (a -> b)` ...

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Just ...
```

Divining a `fmap` from the data definition

Recall the type signature for Maybe's map:

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

We play the same game as before:

- We have nothing of type b …
- but we DO have x, of type a…
- and a function `f :: (a -> b)` ...

So the **only** thing we can do is apply `f` to `x`.

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Just (f x)
```

Free theorems

This is a beautiful example of **free theorems**, where the generality of a polymorphic datatype forces a unique implementation.

Because we don't know anything about types `a` and `b`, the only way to ever produce a `b` is `(f a)`, and the only way to have an a is to have it passed to us.

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Just (f x)
```

Free theorems

This is a beautiful example of **free theorems**, where the generality of a polymorphic datatype forces a unique implementation.

Is there still a unique implementation for fmap if it isn't polymorphic?

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data MaybeInt = Nothing
              | JustInt Int

fmap f Nothing = ...
fmap f JustInt i = ...
```

Free theorems

Is there still a unique implementation for fmap if it isn't polymorphic?

```
fmap::(int -> int) -> MaybeInt -> MaybeInt
```

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data MaybeInt = Nothing
              | JustInt Int

fmap f Nothing = ...
fmap f JustInt i = ...
```

Free theorems

Is there still a unique implementation for fmap if it isn't polymorphic?

```
fmap::(int -> int) -> MaybeInt -> MaybeInt
```

Because `int` is a concrete type, we can instantiate any number of valid instances of a `MaybeInt`: `JustInt 42`, `JustInt 99`, ...

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data MaybeInt = Nothing
              | JustInt Int

fmap f Nothing = ...
fmap f JustInt i = ...
```

Free theorems

Recall the type signature for Maybe's map:

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

But for a polymorphic `Maybe` we don't know the first thing about what a possible `a` or `b` is, so the only possible implementation forces applying `f` to `a`.

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Just (f x)
```

Free theorems

Another free theorem example: we said that the identity function has type signature `a -> a`

By a similar argument, the only polymorphic function `a->a` is the identity function! We have no idea how else to construct another `a`, so our only choice is to just produce the one we already have.

The built-in option type in Haskell is called `Maybe`. It is defined as:

```
data Maybe a = Nothing
             | Just a

fmap f Nothing = Nothing
fmap f Just x = Just (f x)
```

The Either type

We use Nothing to represent the absence of a valid value, but as Nothing is a nullary value constructor, when we get a Nothing we don't know what, exactly, went wrong

```
optAdd :: (Num a) => Maybe a -> Maybe a -> Maybe a
optMul :: (Num a) => Maybe a -> Maybe a -> Maybe a
optDiv :: (Num a) => Maybe a -> Maybe a -> Maybe a
```

If some huge expression involving optional arithmetic produces a `Nothing`, we have no idea of what invalid operation led to that `Nothing` being produced

The Either type

It would be nice to have a datatype with two value constructors:

- One value constructor that wraps a "valid" value
- One value constructor that wraps an "error" value (or something indicating what went wrong"

```
data Either ...
```

The Either type

Since the type of the valid result and the "error-reporting" result can be different, this needs to be a polymorphic

```
data Either a b = ...
```

The Either type

By convention, `Right` holds the "correct" value, (mnemonic: "right" and "correct" are synonyms), and `Left` wraps the error-describing value.

Note that this is more general than the "valid or error" usecase, but that's often what an Either is used for.

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

The Either type

```
-- Extracting comma-separated values using the Parsec parsing library
-- http://book.realworldhaskell.org/read/using-parsec.html
import Text.ParserCombinators.Parsec

parseCSV :: String -> Either ParseError [[String]]
```

The Either type

```
-- Extracting comma-separated values using the Parsec parsing library
-- http://book.realworldhaskell.org/read/using-parsec.html
import Text.ParserCombinators.Parsec

parseCSV :: String -> Either ParseError [[String]]

*Main Text.Parsec> parseCSV "a,b,c"
```

The Either type

```
-- Extracting comma-separated values using the Parsec parsing library
-- http://book.realworldhaskell.org/read/using-parsec.html
import Text.ParserCombinators.Parsec

parseCSV :: String -> Either ParseError [[String]]

*Main Text.Parsec> parseCSV "a,b,c"
Left "Lecture 20" (line 2, column 7):
unexpected end of input
expecting "," or "\n"

*Main Text.Parsec>
```

The Either type

```
-- Extracting comma-separated values using the Parsec parsing library
-- http://book.realworldhaskell.org/read/using-parsec.html
import Text.ParserCombinators.Parsec

parseCSV :: String -> Either ParseError [[String]]

*Main Text.Parsec> parseCSV "a,b,c"
Left "Lecture 20" (line 2, column 7):
unexpected end of input
expecting "," or "\n"

*Main Text.Parsec> parseCSV "a,b,c\nd,e,f\n"
Right [["a","b","c"],["d","e","f"]]
```