CSC324 Lecture 21

Last time

- We were introduced to the Either functor
- We will see a limitation of fmap when composing effects within a functor, and derive a new "design pattern" that addresses it.

The Either type: What should `fmap` do?

An `Either` is a functor, so we need to decide whether fmap operates within a `Left` or a `Right` context.

With Maybe, there was only one choice -- it had to operate on the `a` within the `Just a` -- but here we have two options (the `a` within the `Left` and the `b` within the `Right`)

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

fmap :: (a → b) → Either ? ? → Either ? ?
```

The Either type: What should `fmap` do?

Since the more typical case of an `Either` is to apply a series effect on valid data than invalid ones, by convention, fmap lifts computation into the `Right` and leaves the `Left` unaffected.

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

fmap :: (a → b) → Either e a → Either e b

fmap _ (Left x) = Left x
fmap f (Right x) = Right (f x)
```

fmap on a Right

```
*Main Text.ParserCombinators.Parsec> parseCSV "a,b\nc,d\n"
Right [["a","b"],["c","d"]]

*Main Text.ParserCombinators.Parsec> fmap concat (parseCSV "a,b\nc,d\n")
Right ["a","b","c","d"]
```

fmap on a Left

```
*Main Text.ParserCombinators.Parsec> parseCSV "a,b"
Left "Lecture 20" (line 1, column 4):
unexpected end of input
expecting "," or "\n"

*Main Text.ParserCombinators.Parsec> fmap concat (parseCSV "a,b")
Left "Lecture 20" (line 1, column 4):
unexpected end of input
expecting "," or "\n"
```

Composing effects within a context

Let's write an expression that:

- Given a list of integers,
- extracts the first element from the list
- produces one plus that element

Composing effects within a context

Let's write an expression that:

- Given a list of integers,
- extracts the first element from the list
- produces one plus that element

Obviously, a list of integers may not have a first element, so that suggests the final expression should be a `Maybe Int`.

Composing effects within a context

Let's write an expression that:

- Given a list of integers,
- extracts the first element from the list
- produces one plus that element

The context in which the effect will take place

The effect to take place within the context

Obviously, a list of integers may not have a first element, so that suggests the final expression should be a `Maybe Int`.

Uncons: "the opposite of cons"

If you've looked at the quiz sample questions, you've been introduced to unfold.

Recall the type signature of cons:

```
*Main Data.List> :t (:)
(:) :: a -> [a] -> [a]
*Main Data.List>
```

How could we write a function "uncons" that "undoes a cons"?

Uncons: "the opposite of cons"

`uncons` produces the arguments that, when applied to `cons`, produces the original list.

```haskell
uncons :: [a] -> Maybe (a, [a])
uncons [] = Nothing
uncons (x : xs) = Just (x, xs)
```

What happens if no such arguments to `cons` exist? It produces `Nothing`.

Uncons: "the opposite of cons"

`uncons` produces the arguments that, when applied to `cons`, produces the original list.

What happens if no such arguments to `cons` exist? It produces `Nothing`.

```haskell
uncons :: [a] -> Maybe (a, [a])
uncons [] = Nothing
uncons (x : xs) = Just (x, xs)
```

```
*Main> uncons []
Nothing
*Main> uncons [1,2,3]
Just (1,[2,3])
*Main> (:) 1 [2,3]
[1,2,3]
*Main>
```

Uncons: "the opposite of cons"

Because uncons returns a Maybe, if we want to do anything with the extracted head and tail of the list, those **effects** will have to take place **within the context** of the **Maybe.**

```haskell
uncons :: [a] -> Maybe (a, [a])
uncons [] = Nothing
uncons (x : xs) = Just (x, xs)
```

```
*Main> uncons []
Nothing
*Main> uncons [1,2,3]
Just (1,[2,3])
*Main> (:) 1 [2,3]
[1,2,3]
*Main>
```

Composing effects within a context

```
Prelude> import Data.List
Prelude Data.List> :t uncons
uncons :: [a] -> Maybe (a, [a])
Prelude Data.List> uncons [1,2,3]
Just (1,[2,3])
Prelude Data.List> fmap fst (uncons [1,2,3])
Just 1
Prelude Data.List> fmap (+1) (fmap fst (uncons [1,2,3]))
Just 2
Prelude Data.List> (fmap (+1) . (fmap fst)) (uncons [1,2,3])
Just 2
Prelude Data.List> fmap ((+1) . (fst)) (uncons [1,2,3])
Just 2
Prelude Data.List>
```

Composing effects within a context

The effect to take place within the context: the function that composes extracting the first from a tuple and adding 1 to it

The context in which the effect will take place: an instance of a `Maybe (Int, [Int])`

```
> fmap ((+1) . fst) (uncons [1,2,3])
Just 2
```

Composing effects within a context

Let's write an expression that:

- Given a list of integers,
- extracts the **2nd** element from the list
- produces one plus that element

The context in which the effect will take place

The effect to take place within the context

Obviously, a list of integers may not have a **second** element, so that suggests the final expression should be a `Maybe Int`.

Composing effects within a context

We would like to use `uncons` in the following way:

- If the list is empty, produce `Nothing`
- Get the first and rest of the list from `uncons`, and apply `uncons` to the rest of the list
- If the rest of the list is empty, produce `Nothing`
- Else: produce the `Just` of first of the rest of the list (that is, the 2nd element)

Failable operation

Failable operation

If the inner operation fails, the whole operation should be seen to fail!

Composing effects within a context

```
Prelude Data.List> uncons [1,2,3]
Just (1,[2,3])
Prelude Data.List> fmap snd (uncons [1,2,3])
Just [2,3]
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

What went wrong?

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

The inner fmap has signature:

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

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

The inner fmap has signature:

```
fmap :: ((Int,[Int]) -> [Int]) -> Maybe (Int, [Int]) -> Maybe [Int]
```

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

The outer fmap has signature:

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

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

The outer fmap has signature:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: ([Int] -> Maybe (Int, [Int])) -> Maybe [Int] -> Maybe (Maybe (Int, [Int]))
```

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: ([Int] -> Maybe (Int, [Int])) -> Maybe [Int] -> Maybe (Maybe (Int, [Int]))
```

We wanted to **compose** the effects into a single context, but instead we **nested** a context within a context!

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: ([Int] -> Maybe (Int, [Int])) -> Maybe [Int] -> Maybe (Maybe (Int, [Int]))
```

What is the type signature we want instead?

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: ([Int] -> Maybe (Int, [Int])) -> Maybe [Int] -> Maybe (Maybe (Int, [Int]))
```

What is the type signature we want instead? We don't want the resulting b to be wrapped into its own Maybe...

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

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

Where have we seen this type signature before

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

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

Conmapenate, yet again!!

The cause of, and solution to, all of life's problems

Composing effects within a context

```
Prelude Data.List> fmap uncons (fmap snd (uncons [1,2,3]))
Just (Just (2,[3]))
```

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

For clarity's sake, let's call this function `andThen`, signifying that **after** we've lifted an `a` into a `Maybe a`, **then** we'll do some operation to produce a `Maybe b`.

implementing andThen

Surprisingly straightforward!

The function we pass in is a function that is like a "constructor" of `Maybe`s

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing  _ = Nothing
andThen (Just a) f = f a
```

implementing andThen

Surprisingly straightforward!

```haskell
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing  _ = Nothing
andThen (Just a) f = f a
```

(pause and ponder: was there a free theorem that uniquely determined the implementation of this function given its type signature?)

```
*Main> uncons [10,20,30]
Just (10,[20,30])
*Main> (uncons [10,20,30]) `andThen` (\ x -> Just (snd x))
Just [20,30]
*Main> (uncons [10,20,30]) `andThen` (Just . snd)
Just [20,30]
*Main> (uncons [10,20,30]) `andThen` (Just . snd) `andThen` uncons
Just (20,[30])
*Main> (uncons [10,20,30]) `andThen` (Just . snd) `andThen` uncons `andThen` (Just . (+1) . fst)
Just 21
*Main>
```

```
*Main> uncons [10]
Just (10,[])
*Main> uncons [10] `andThen` (Just . snd)
Just []
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons
Nothing
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons `andThen` (Just . (+1) . fst)
Nothing
*Main>
```

This uncons call produced a Nothing...

```
*Main> uncons [10]
Just (10,[])
*Main> uncons [10] `andThen` (Just . snd)
Just []
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons
Nothing
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons `andThen` (Just . (+1) . fst)
Nothing
*Main>
```

This uncons call produced a Nothing...

```
*Main> uncons [10]
Just (10,[])
*Main> uncons [10] `andThen` (Just . snd)
Just []
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons
Nothing
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons `andThen` (Just . (+1) . fst)
Nothing
*Main>
```

...but we can still chain together effects with andThen without the whole expression breaking

Nothing vs `NULL` pointers

You may come across half-baked Medium posts explaining datatypes such as `Maybe` in terms of producing Nothing is "kind of like producing NULL in C/Java/etc, `undefined` in JavaScript, or `nil` in Go…"

Nothing vs `NULL` pointers

In Go, functions return not a single value but a tuple of the value and an optional error value (if it's `nil`, the first element is a valid return value)

```go
resp, err := http.Get(genEndpoint)
if err != nil {
    return err
}
data, err := ioutil.ReadAll(resp.Body)
if err != nil {
    return err
}
```

In Java, NULL is often used to indicate an absent return value (like looking non-existent things up in hash tables)

```java
Map<Integer, String> map = new HashMap<>();
map.put(1, "Amir");
map.put(2, "Beth");
map.put(3, null);

String name = (String)map.get(3);  // null
name = (String)map.get(4);  // null
```

Nothing vs NULL pointers

If you have a function that returns NULL on "the error case", you still need to manually check for "the special sentinel value" and propagate the error up to the caller.

Here's a chain of failable function calls in Go; each one needs a "did this fail, and if so, return the error" check!

```go
func upgradeUser(endpoint, username string) error {
    getEndpoint := fmt.Sprintf("%s/oldusers/%s", endpoint, username)
    postEndpoint := fmt.Sprintf("%s/newusers/%s", endpoint, username)

    resp, err := http.Get(genEndpoint)
    if err != nil {
        return err
    }
    data, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        return err
    }
    olduser, err := user.NewFromJson(data)
    if err != nil {
        return err
    }
    newuser, err := user.NewUserFromUser(olduser),
    if err != nil {
        return err
    }
    buf, err := json.Marshal(newuser)
    if err != nil {
        return err
    }
    _, err = http.Post(
        postEndpoint,
        "application/json",
        bytes.NewBuffer(buf),
    )
    return err
}
```

This uncons call produced an "error value"

```
*Main> uncons [10]
Just (10,[])
*Main> uncons [10] `andThen` (Just . snd)
Just []
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons
Nothing
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons `andThen` (Just . (+1) . fst)
Nothing
*Main>
```

...but we can still chain together effects with `andThen` without the whole expression breaking

andThen in the real world:

https://monkey.org/~marius/funsrv.pdf

This is a paper that describes the architecture of Twitter's backend servers

Twitter makes heavy use of statically-typed languages (Scala in particular), so let's see what lessons they have to offer us…

(full disclosure: I worked on this software, so I'm biased to thinking it's good…)

andThen in the real world:

The functionality that a Twitter web server performs when a client connects to it is a series of **independent concerns:**

- **Logging the client connection**
- **Parsing the HTTP request**
- **Data sanitization**
- **Routing the request to an appropriate backend service**

Independent concerns refers to each of these things being orthogonal (logging has nothing to do with data sanitization, etc) and in a large company are often worked on by different people on different teams.

Composing fns with `andThen` in the real world:

sally: Finagle itself uses filters heavily; our frontend web servers—reverse HTTP proxies through which all of our external traffic flows—use a large stack of filters to implement different aspects of its responsibilities. This is an excerpt from its current configuration:

```
recordHandletime     andThen
traceRequest         andThen
collectJvmStats      andThen
parseRequest         andThen
logRequest           andThen
recordClientStats    andThen
sanitize             andThen
respondToHealthCheck andThen
applyTrafficControl  andThen
virtualHostServer
```

Here's the piece of code in Twitter's web frontend that handles all of those independent pieces:

The server is a series of functions, composed together with `andThen`!

Composing fns with `andThen` in the real world:

sally: Finagle itself uses filters heavily; our frontend web servers—reverse HTTP proxies through which all of our external traffic flows—use a large stack of filters to implement different aspects of its responsibilities. This is an excerpt from its current configuration:

```
recordHandletime      andThen
traceRequest          andThen
collectJvmStats       andThen
parseRequest          andThen
logRequest            andThen
recordClientStats     andThen
sanitize              andThen
respondToHealthCheck  andThen
applyTrafficControl   andThen
virtualHostServer
```

If the call to, for instance, parseRequest fails and produces an error result…

...as we would expect from the behaviour of functors that we've seen so far, execution terminates for that request and all other functions aren't called.

Composing fns with `andThen` in the real world:


Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

"[They] **help enhance modularity and reusability,** and they have also proved **valuable for testing**. It is quite simple to unit test each of [them] in isolation without any set up"

"Furthermore, they encourage programmers to separate functionality into **independent modules with clean boundaries**, which generally leads to better design and reuse."

A typeclass for `andThen`

As you may have been able to guess by now, there's nothing special about Maybe in our use of andThen:

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

A typeclass for `andThen`

One could just as easily imagine an equivalent function that operates on Either types (and maybe even more types that we'll encounter next week in class!)

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen :: Either e a -> (a -> Either e b) -> Either e b
…
…
```

A typeclass for `andThen`

One could just as easily imagine an equivalent function that operates on Either types (and maybe even more types that we'll encounter next week in class!)

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen :: Either e a -> (a -> Either e b) -> Either e b
…
…
andThen :: x a -> (a -> x b) -> x b
```

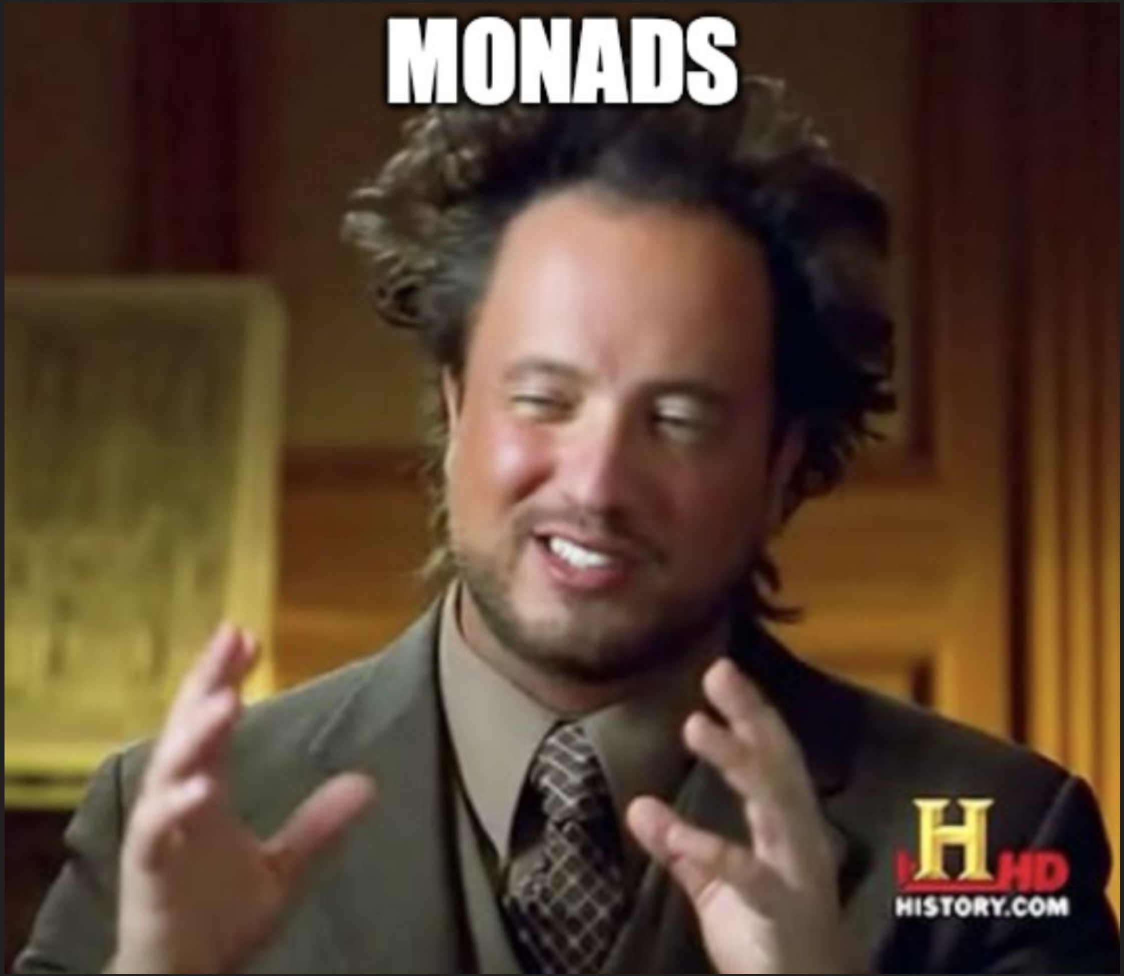
These type variables unify with particular types (ie. resulting in a `Either String [Int]`)...

A typeclass for `andThen`

One could just as easily imagine an equivalent function that operates on Either types (and maybe even more types that we'll encounter next week in class!)

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen :: Either e a -> (a -> Either e b) -> Either e b
…
…
andThen :: x a -> (a -> x b) -> x b
```

Whereas x unifies against a **type constructor**. x requires another type (the type variable) in order to become a type.


MONADS
H HD
HISTORY.COM

A typeclass for `andThen`: Monad

A **monad** is a typeclass that might be defined in the following way:

```
-- Not exactly the Haskell implementation
class Functor m => Monad m where
    bind :: m a -> (a -> m b) -> m b
```

In Monad-Land, `andThen` is called "bind" because it "combines" the functor computational context to the effectful-function to produce another context.

A typeclass for `andThen`: Monad

A **monad** is a typeclass that might be defined in the following way:

```
-- Not exactly the Haskell implementation
class Functor m => Monad m where
    bind :: m a -> (a -> m b) -> m b
```

Is this all we need for a Monad? Let's look at our example from a few minutes ago.

Our claim was: as it's part of a typeclass, `andThen` should operate on any possible monad instance...

```
*Main> uncons [10]
Just (10,[])
*Main> uncons [10] `andThen` (Just . snd)
Just []
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons
Nothing
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons `andThen` (Just . (+1) . fst)
Nothing
*Main>
```

Our claim was: as it's part of a typeclass, `andThen` should operate on any possible monad instance...

```
*Main> uncons [10]
Just (10,[])
*Main> uncons [10] `andThen` (Just . snd)
Just []
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons
Nothing
*Main> uncons [10] `andThen` (Just . snd) `andThen` uncons `andThen` (Just . (+1) . fst)
Nothing
*Main> 
```

We are using `Just` here to lift the incremented Num into a Maybe Num… but this is specific to this particular monad instance. We'd have to rethink this piece of code every time we wanted to lift a value into an appropriate monad.

A typeclass for `andThen`: Monad

A **monad** is a typeclass that might be defined in the following way:

```
-- Not exactly the Haskell implementation
class Functor m => Monad m where
    return :: a -> m a
    bind :: m a -> (a -> m b) -> m b
```

`return` returns a "pure value" of type `a` in `m`'s computational context.

A typeclass for `andThen`: Monad

A **monad** is a typeclass that might be defined in the following way:

```
-- Not exactly the Haskell implementation
class Functor m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

In Haskell, bind is named with this funny-looking operator. (Maybe it helps to imagine it as "the monad is being fed righwards, ">>", into the effectful function?)

>>= and return in action

```
Prelude Data.List> uncons [1,2,3]
Just (1,[2,3])
Prelude Data.List> (uncons [1,2,3]) >>= (Just . snd)
Just [2,3]
Prelude Data.List> (uncons [1,2,3]) >>= (return . snd)
Just [2,3]
```

Next week

- We will learn the **monad laws**, which, in a similar way to the functor laws, put algebraic constraints on the relationship between `(>>=)` and `return`.
- We will learn about **do-notation**, which is a nicer way of chaining monadic operations than these huge chains of `(>>=)` function calls.

- We'll see a monad that lets us mimic **mutable state!**
- We'll see a monad that lets us perform general side-effecting operations!!

Good luck studying; see you on Monday for the quiz!

do-notation

Because of the power that comes from the **monad laws** (a similar set of algebraic rules to the functor laws that we will discuss on Wednesday), Haskell is able to provide a richer form of syntactic sugar to apply and chain effects to monads.

do