Programación Declarativa 2020-2
Mónadas 102 `Maybe`

Javier Enríquez Mendoza

Facultad de Ciencias UNAM

21 de mayo de 2020

Maybe

En muchos lenguajes existe un valor `null` que se adapta a cualquier tipo y lanza un error

Lo cual es una barbaridad pues interrumpe la ejecución del programa

Incluso su inventor Sir Tony Hoare se disculpo en 2009 por la creación de `null` diciendo que era un *billion dollar mistake*

En programación funcional se tiene una alternativa mucho mas elegante, el tipo `Maybe`

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


Es bellísimo...

Personas

Un clásico uso de `Maybe` es para validar la entrada del usuario

Tomemos como ejemplo el siguiente `record` para definir a una persona

```
type Name = String
data Person = Person {name :: Name, age :: Int}
```

Validacones

Antes de crear un registro de persona nos gustaría estar seguros de que la información proporcionada es la adecuada

Para eso se definen funciones de validación para estos campos, obviamente usando `Maybe`

```
valName :: String -> Maybe Name
valAge :: Int -> Maybe Int
```

Validar persona

Para validar a una persona se tienen que componer las funciones anteriores para que valide ambos campos

```
valPerson :: String -> Int -> Maybe Person
valPerson name age =
 case valName name of
  Nothing    -> Nothing
  Just name' -> case valAge age of
    Noting    -> Nothing
    Just age' -> Just (Person name' age')
```

¿No que Haskell era muy elegante?

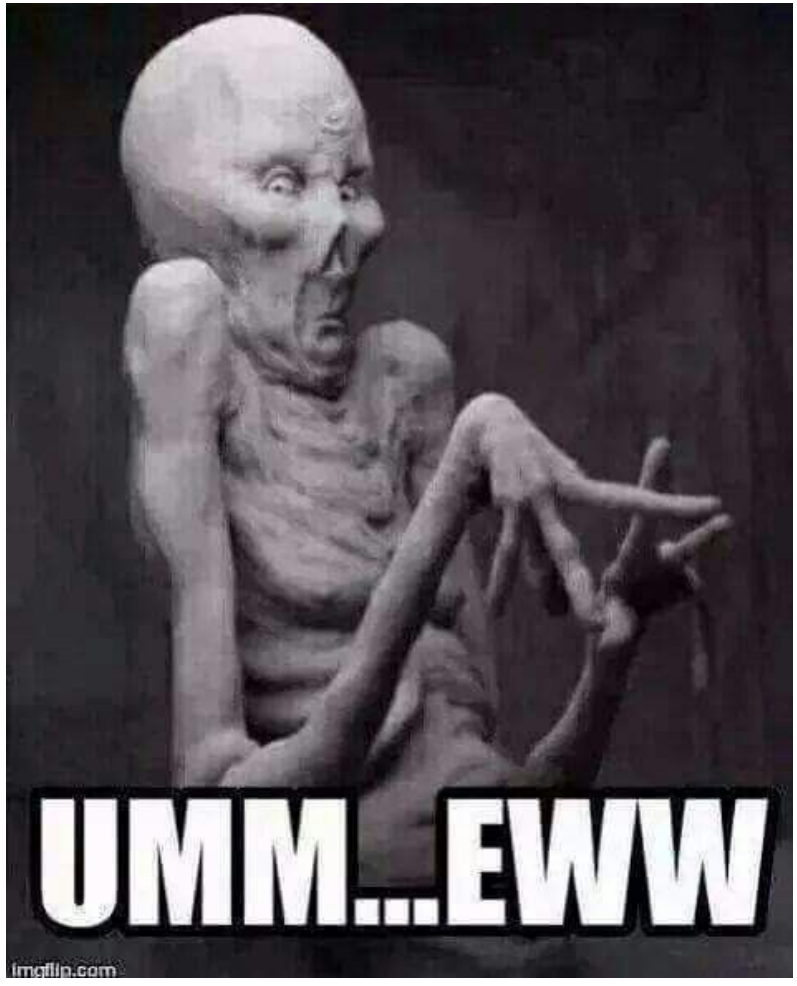
Problemas

La definición de la validación de personas es fea, pero ese no es el único problema

La validación también es muy poco general y difícilmente escalable

Si se agregan mas capos a `Person` la definición sigue ramificándose

La peor parte es que no hacemos mas que repetir el mismo código por cada campo


UMM...EWW
imgflip.com

Refactorización

Tratemos d refactorizar la función

El patrón es sencillo de abstraer

```
case v of
 Nothing -> Nothing
 Just v' -> nextAction ... v' ...
```

Refactorización

Utilicemos el poder de las funciones de orden superior para convertir este patrón en una función

```
then_ :: Maybe a -> (a -> Maybe b) -> Maybe b
then_ v g = case v of
                Nothing -> Nothing
                Just v' -> g v'
```

¡Podemos mejorar! usemos la pattern matching

```
then_ :: Maybe a -> (a -> Maybe b) -> Maybe b
then_ Nothing _  = Nothing
then_ (Just v) g = g v
```

valPerson

Reescribamos `valPerson` usando `then_`

```
valPerson :: String -> Int -> Maybe Person
valperson name age =
        valName name `then` \name' ->
        valAge age `then` \age' ->
        Just (Person name' age')
```

Mucho mejor


WAIT A MINUTE...
makeameme.org

Mónada

Esto también se ve como una mónada

Esta simulando el atrapar un error, un comportamiento imperativo

Y claramente se puede secuenciar cómputos, hasta se llama `then_` la función

¡¡Maybe es una mónada!!

La instancia de la clase `Monad` es:

```
instance Monad Maybe where
    return = Just
    (>>=) = then_
```

Reescribamos la función `valPerson` utilizando la notación de mónadas

```
valPerson :: String -> Int -> Maybe Person
valPerson name age =
        valName name >>= \name' ->
        valAge age >>= \age' ->
        return (Person name' age')
```

do

Ahora usemos la notación do

```
valPerson :: String -> Int -> Maybe Person
valPerson name age = do
        name' <- valName name
        age' <- valAge age
        return (Person name' age')
```


MAYBE IN A MONAD
HISTORY.COM