Haskell4Life: Monads

Sergiu Ivanov

sergiu.ivanov@lacl.fr

Slides and code examples available online:

http://lacl.fr/~sivanov/doku.php?id=en:haskell_for_life

Outline

1. Warm-up

2. Monads

Outline

1. Warm-up

2. Monads

Bad Question

What is pattern matching?

Case Expressions

```
case expression of
  pattern_1 -> expression_1
            ...
  pattern_n -> expression_n
```

Case Expressions

```
case expression of
  pattern_1 -> expression_1
            ...
  pattern_n -> expression_n
```

```
myLength [] = 0
myLength (_:xs) = 1 + myLength xs
```

is the same thing as

```
myLength list = case list of
  [] -> 0
  (_:xs) -> 1 + myLength xs
```

Pointfree Notation

```
mulByTwo :: [Int] -> [Int]
mulByTwo xs = map (*2) xs
```

Pointfree Notation

```
mulByTwo :: [Int] -> [Int]
mulByTwo xs = map (*2) xs
```

One can also write (and it is considered cleaner):

```
mulByTwo = map (*2)
```

Pointfree Notation

```
mulByTwo :: [Int] -> [Int]
mulByTwo xs = map (*2) xs
```

One can also write (and it is considered cleaner):

```
mulByTwo = map (*2)
```

To combine two such transformations :

```
lenMulByTwo = length . map (*2)
```

The same thing as

```
lenMulByTwo xs = length (map (*2) xs)
```

Outline

Chains of `Maybe` Functions

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

Chains of Maybe Functions

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

```
case personByName "John" of
  Nothing -> Nothing
  Just john ->
```

Chains of Maybe Functions

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

```
case personByName "John" of
  Nothing -> Nothing
  Just john ->
    case carByPerson john of
      Nothing -> Nothing
      Just johnsCar -> model johnsCar
```

Chains of Maybe Functions

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

Suppose we want to know the model of John's car.

```
case personByName "John" of
  Nothing -> Nothing
  Just john ->
    case carByPerson john of
      Nothing -> Nothing
      Just johnsCar -> model johnsCar
```

Imagine what happens if one has longer chains.

State

```
addPerson  ::  Person -> Database -> Database
addCar     ::  Car -> Database -> Database
```

State

```
addPerson  ::  Person -> Database -> Database
addCar     ::  Car -> Database -> Database
```

Modify something =

1. take old something
2. produce new something

This pattern arises very often.

What's in Common (Monads! \o/)

We want to do the same thing over and over between two function calls:

What's in Common (Monads! \o/)

We want to do the same thing over and over between two function calls:

- check whether the previous `Maybe`-call returned `Just`

What's in Common (Monads! \o/)

We want to do the same thing over and over between two function calls:

- check whether the previous `Maybe`-call returned `Just`
- take new something and put it into the next function.

What's in Common (Monads! \o/)

We want to do the same thing over and over between two function calls:

- check whether the previous `Maybe`-call returned `Just`
- take new something and put it into the next function.

Monads help factor out such patterns.

Popular Monads

Maybe	helps deal with chains of `Maybe` calls
List	helps deal with multiple possible outcomes of a computation
State	helps deal with states
Reader	provides a read-only state (e.g. configuration information)
Writer	provides a log-like functionality ($\sim$ write-only state)
IO	provides an interface with the outer world ($\sim$ the state of the world)

Monads More Formally (in Haskell)

A monad is a parameterised type.

Monads More Formally (in Haskell)

A monad is a parameterised type.

The property of being a monad is described by the `Monad` typeclass.

Monads More Formally (in Haskell)

A monad is a parameterised type.

The property of being a monad is described by the `Monad` typeclass.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b (bind)
  return :: a -> m a
```

- `(>>=)` describes how to combine two functions.
- `return` describes how to pack a value.

The `Maybe` Monad

In the case of the `Maybe` monad:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a
```

`(>>=)` unpacks a value `Maybe a` into a value of type `a`, feeds it into the function `a -> Maybe b`, and returns the result of this function.

Using the Maybe Monad

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

How do we find out the model of John's car?

Using the Maybe Monad

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

How do we find out the model of John's car?

```
return "John" >>= personByName
              >>= carByPerson >>= model
```

Using the Maybe Monad

```
personByName :: String -> Maybe Person
carByPerson :: Person -> Maybe Car
model :: Car -> Maybe String
```

How do we find out the model of John's car?

```
return "John" >>= personByName
              >>= carByPerson >>= model

personByName "John"
              >>= carByPerson >>= model
```

```
personByName "John" :: Maybe String
```

Syntactic Sugar: The `do` Notation

```
do
john <- personByName "John"
car <- carByPerson john
model car
```