Introdução a Programação Funcional com Haskell

Fabrício Olivetti de França, Emílio Francesquini

29 de Setembro de 2018

Aplicando funções dentro de ADTs

Tipos Paramétricos

Na aula anterior vimos definições de tipos paramétricos, como:

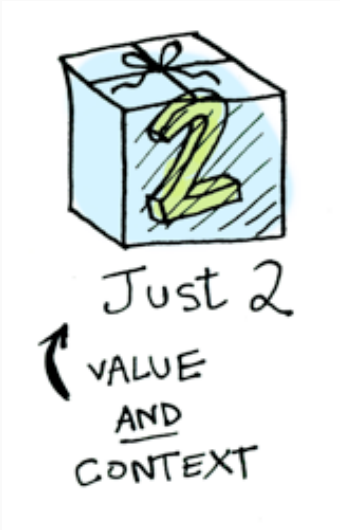
```haskell
data Maybe a = Nothing | Just a
```

```haskell
data List  a = a | a : List a
```

Tipos Paramétricos

Com isso podemos utilizar tipos primitivos dentro de contêiners:

```
['1', '2', '2'] :: [Char]
Just 30         :: Maybe Int
```



Tipos Paramétricos

Assumindo que eu tenho uma função `f :: a -> b` e um tipo `T a`, como faço para aplicá-la gerando um tipo `T b`?

Tipos Paramétricos

Por exemplo, quero converter uma lista de char contendo dígitos em inteiros, ou dobrar o valor de um `Maybe Int`.

Tipos Paramétricos

Para a lista utilizamos:

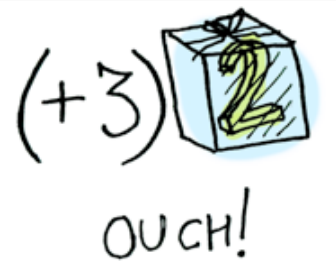
```
import Data.Char

> map digitToInt "123"
[1,2,3]
```

Tipos Paramétricos

E para um `Maybe Int`?

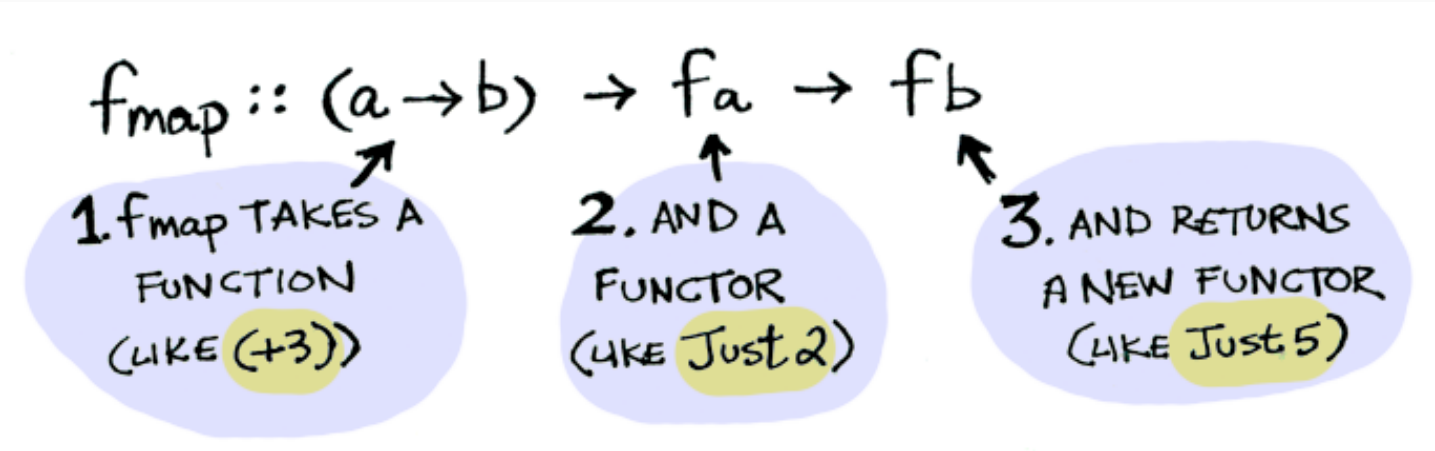
```
> map (+3) (Just 2)
erro!
```



Functors

A classe de tipos `Functor` define a função `fmap`, similar a `map`:

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



Functors

Todo tipo `f` que pertence a classe `Functor` pode fazer uso de `fmap`:

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

Functors

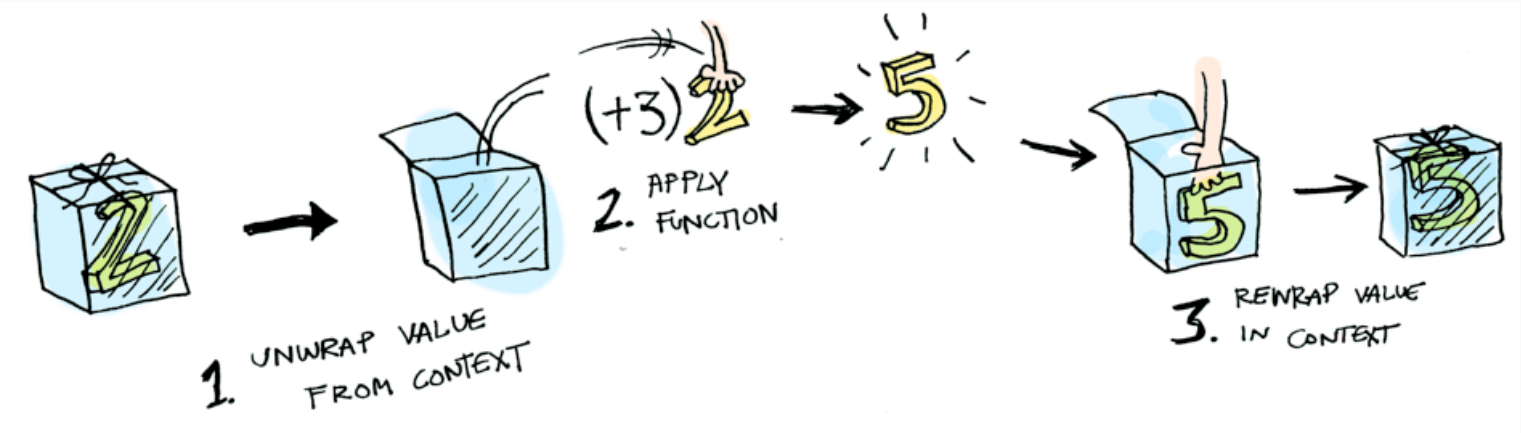
Para o tipo `Maybe`:

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

Functors

E com isso podemos fazer:

```
> fmap (+3) (Just 2)
Just 5
```




```
> fmap (+3) Nothing
Nothing
```

Aplicando funções de múltiplos argumentos

Applicative

E se precisarmos fazer:

```
> (Just 3) + (Just 2)
Erro!
```

Applicative

Para isso temos a classe `Applicative`:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

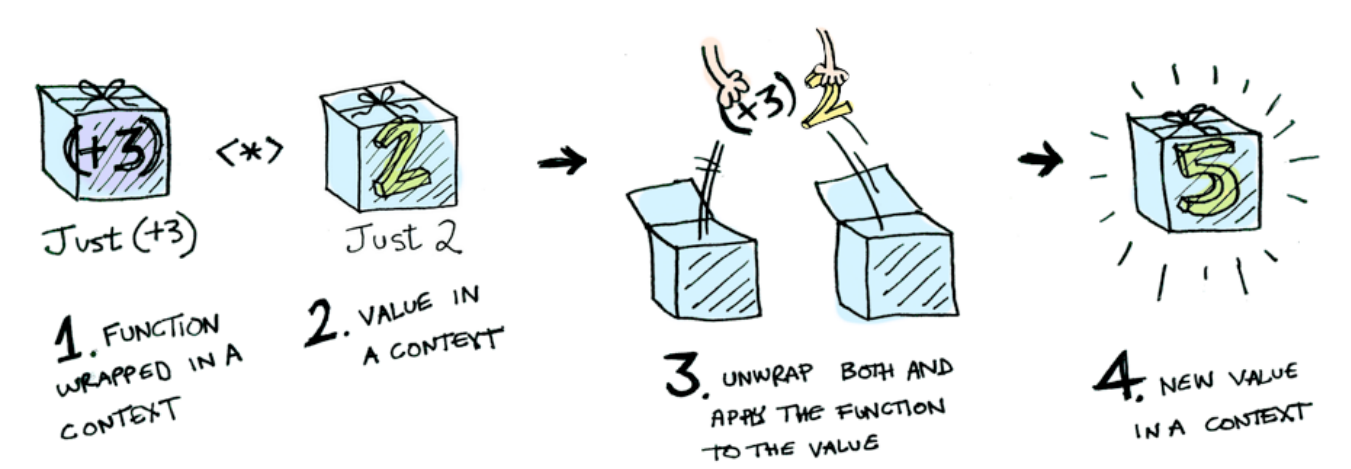
Applicative

Problema original:

```
> (Just 3) + (Just 2)
```

Applicatives nos permitem fazer o seguinte:

```
> Just (+3) <*> Just 2
> fmap (+) (Just 3) <*> Just 2
> Just (+) <*> Just 3 <*> Just 2
> pure (+) <*> Just 3 <*> Just 2
Just 5
```



Applicative em Listas

Para listas, o `Applicative` é definido como a combinação de todos os elementos das listas:

```
[(+), (*)] <*> [1,2] <*> [3,4]
> [1+3, 1+4, 2+3, 2+4, 1*3, 1*4, 2*3, 2*4]
> [4,5,5,6,3,4,6,8]
```

Sequenciando operações

Tratando o tipo `Maybe`

Digamos que temos um conjunto de funções do tipo *safe* que recebem um ou mais argumentos do tipo `Float` e retornam um `Maybe Float`:

```
safeDiv :: Float -> Float -> Maybe Float
safeLog :: Float -> Maybe Float
safeTan :: Float -> Maybe Float
```

Tratando o tipo `Maybe`

Como poderíamos aplicar essas funções em sequência?

```
safeTan . safeLog $ safeDiv 2 3
```

As assinaturas não batem!

- E se usarmos um Functor?

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

- E se usarmos um Applicative?

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Tratando o tipo `Maybe`

Como poderíamos aplicar essas funções em sequência?

```
safeTan . safeLog $ safeDiv 2 3
```

Precisamos de algo novo!

Monads

A classe `Monad` define o operador `bind` (`>>=`):

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

Monads

A função `bind` (`>>=`) aplica uma função que devolve um valor dentro de um contêiner (como fazem `safeTan`, `safeLog` e `safeDiv`) no valor dentro do contêiner.

Monads

Agora podemos fazer:

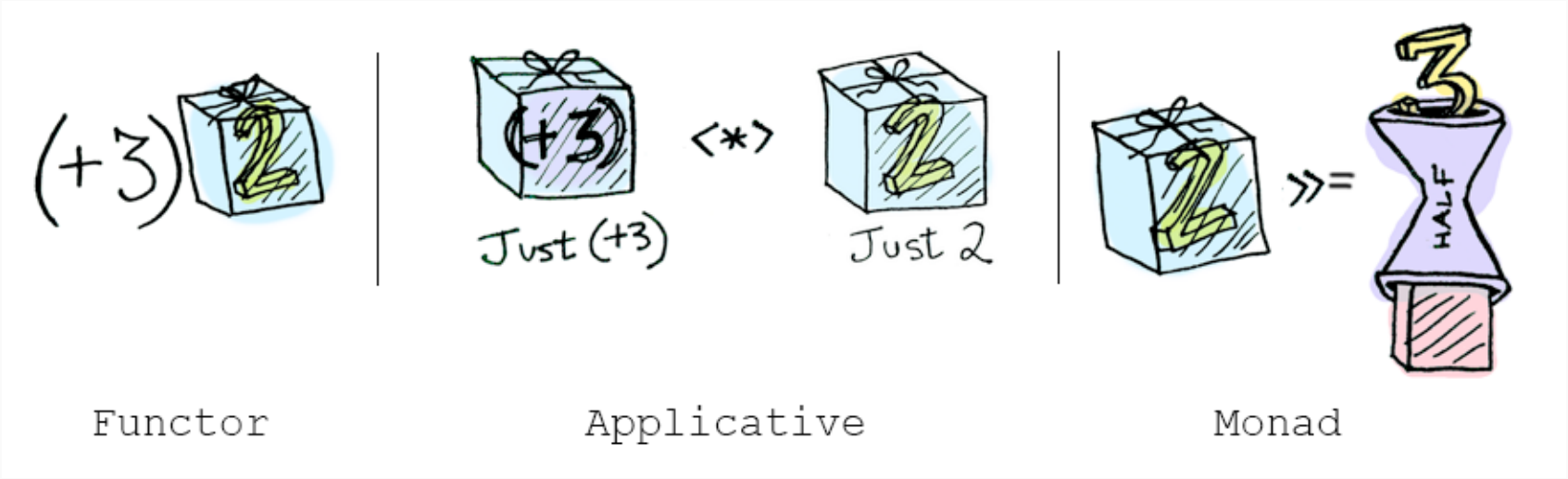
```
composicao x y = safeDiv x y >>= safeLog >>= safeTan
```

O resultado de `safeDiv` será extraído do `Maybe` e passado para `safeLog`, o mesmo processo se repete para `safeTan`.

Monads

```
> composicao 2 3
Just (-0.43)
> composicao 1 0
Nothing
> composicao 0 1
Nothing
```

Resumindo



- **Functor**: aplica uma função a um valor dentro de um contêiner com `fmap`
- **Applicative**: aplica uma função em um contêiner a um valor dentro de um contêiner com `<*>`
- **Monad**: aplica uma função que devolve um valor em um contêiner a um valor que já está em um contêiner com `>>=`

Entrada e Saída

Entrada e Saída

Para entrada e saída de dados precisamos garantir que a execução das funções sejam feitas em sequência. Considere a função `getChar` que captura um caracter do teclado:

```
doisChar = getChar . getChar
```

Entrada e Saída

Se a sintaxe anterior fosse aceita, ao digitar “ab”, dependendo da ordem de execução poderíamos receber “ab” ou “ba”.

Entrada e Saída

Para resolver esse problema utilizamos a classe `Monad` com uma notação diferente, a notação `do`:

```
doisChar :: IO String
doisChar = do
    c1 <- getChar
    c2 <- getChar
    return [c1,c2]
```

Entrada e Saída

Todas as instruções do bloco do serão executadas na ordem.

O operador `<-` retira o conteúdo do contêiner de tipo e armazena na variável a esquerda.

A função `return` devolve o conteúdo dentro do contêiner.

Entrada e Saída

Todo bloco do **deve** terminar com uma instrução que retorna o contêiner `Monad` que está sendo trabalhado.

Entrada e Saída

Outra função útil é `putChar`:

```
eco :: IO ()
eco = do c <- getChar
         putChar c
```

`IO ()` indica que ele executa uma ação de `IO` mas não retorna nada.

Entrada e Saída

Outras funções:

```
getLine  :: IO String
putStr   :: IO ()
putStrln :: IO ()
```

Leitura e Escrita de Arquivos

A leitura e escrita de arquivos é feita utilizando as funções:

```
readFile  :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Exemplo

Vamos criar um programa que lê um arquivo texto, elimina certas palavras contidas em uma lista e escreve uma nova versão do arquivo:

```
redata :: FilePath -> IO ()
redata fp = do
          conteudo <- readFile fp
          writeFile (fp ++ "_corrigido") $ filtra conteudo
```

Exemplo

```haskell
filtra :: String -> String
filtra s = unwords
         $ filter (`notElem` proibido)
         $ words s
  where
    proibido = ["reprovado", "baixas", "zero", "recuperação"]
```

Referências

Figuras retiradas de: http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html