Functores

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```haskell
module Main where

main :: IO ()
main = return ()
```

Função map

- Pergunta: podemos definir `map` para outros tipos?

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x : xs) = f x : map f xs
```

Função `map`

- Tarefa: definir `map` para o tipo `Maybe`:

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

Função mapMay

- Passo 1: Definindo o tipo

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

Função mapMay

- Passo 2: Enumerando os casos

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

Função mapMay

- Passo 3: Definindo o caso para `Nothing`:

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

Função mapMay

- Passo 4: Definindo o caso para `Just`:

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

Next one!

- Definir `map` para árvores binárias:

```
data Tree a
  = Leaf
  | Node a (Tree a) (Tree a)
  deriving (Eq, Ord, Show)
```

Função `mapTree`

- Passo 1: Definindo o tipo

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

Função mapTree

- Passo 2: Enumerando os casos

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Leaf = _
mapTree f (Node x l r) = _
```

Função mapTree

- Passo 3: Definindo o caso base

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Leaf = Leaf
mapTree f (Node x l r) = _
```

Função mapTree

- Passo 4: Definindo o caso recursivo

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Leaf = Leaf
mapTree f (Node x l r)
   = Node (f x) (mapTree f l) (mapTree f r)
```

Padrão

- Observe o tipo das funções definidas:

```
mapMay :: (a -> b) -> Maybe a -> Maybe b
mapTree :: (a -> b) -> Tree a -> Tree b
```

- Qua o padrão?

Padrão

- Podemos generalizar o tipo das duas funções para

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

em que, `f = Maybe` para `mapMay` e `f = Tree` para `mapTree`.

Quiz

- Considere as seguintes funções:

```
foo :: Int -> Maybe
foo x = Nothing

baz :: String -> Tree
baz s = Leaf
```

Este código está correto? Será aceito pelo compilador?

Quiz

- O GHC nos responde:

```
• Expecting one more argument to ‘Maybe’
    Expected a type, but ‘Maybe’ has kind ‘* -> *’
• In the type ‘(String -> Maybe)’
```

Kinds

- Intuitivamente, kinds são tipos de "tipos".
  - Tipos simples como Bool, String, Int, [Int] são de kind `*`.
  - Construtores de tipos como `Maybe` e `Tree` possuem kind `* -> *`.

Kinds

- Tipos como Maybe e Tree são funções a nível de tipos.
  - Precisam de um parâmetro de tipo para serem tipos efetivamente.
- Podemos consultar o kind de um tipo usando o comando `:k` no ghci.

Exemplos

```
Prelude*> :k Int
Int :: *
Prelude*> :k []
[] :: * -> *
Prelude*> :k (,)
(,) :: * -> * -> *
Prelude*> :k Either
Either :: * -> * -> *
```

Exemplos

- Qual o kind de `GRose`?

```
data GRose f a
  = a :> f (GRose f a)
```

Exemplos

- Qual o kind de `GRose`?
  - Primeiramente, `GRose f a :: *`
  - Note que `f :: * -> *` e `a :: *`

```
data GRose f a
  = a :> f (GRose f a)
```

Exemplos

- Qual o kind de `GRose`?

```
Prelude*> :k GRose
GRose :: (* -> *) -> * -> *
```

Functor

- Tipos de kind `* -> *` e que possuem uma função similar a `map` são chamados de functores.

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

(<$>) :: (a -> b) -> f a -> f b
(<$>) = fmap
```

Functor

- Instância para `Maybe`

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

Functor

- Instância para `Tree`

```
instance Functor Tree where
  fmap _ Leaf = Leaf
  fmap f (Node x l r)
    = Node (f x) (f <$> l) (f <$> r)
```

Árvores

- Apresente a definição de Functor para o tipo:

```
data Rose a
  = Rose a [Rose a]
    deriving (Eq, Ord, Show)
```

Functor

- Instância para Rose

```
instance Functor Rose where
  fmap f (Rose x ts)
    = Rose (f x) ((fmap f) <$> ts)
```

Functor

- Idealmente, functores devem atender as seguintes propriedades:

```
fmap id == id
fmap (g . f) == fmap g . fmap f
```

- Veremos como provar esse tipo de propriedades nas próximas aulas.

Tarefa

- Validar entradas de e-mails em um formulário.
  - Email: deve conter o caractere `@`.
  - Corpo: String não vazia.

Modelagem

```haskell
type From = Address
type To = Address

data Address
   = Address String
     deriving (Eq, Ord, Show)
data Body
   = Body String
     deriving (Eq, Ord, Show)

data Email
   = Email From To Body
     deriving (Eq, Ord, Show)
```

Validação

- Testando se a string é vazia.

```
nonEmpty :: String -> Maybe String
nonEmpty []  = Nothing
nonEmpty s   = Just s
```

Validação

- Testando se uma string contém `@`:

```
contains :: Char -> String -> Maybe String
contains x xs
  | x `elem` xs = Just xs
  | otherwise   = Nothing
```

Validação

- Validando um endereço de e-mail
  - Vamos usar a instância de Functor para Maybe!

```
mkAddress :: String -> Maybe Address
mkAddress s = Address <$> contains '@' s
```

Validação

- Validando o corpo do e-mail

```
mkBody :: String -> Maybe Body
mkBody s = Body <$> nonEmpty s
```

Validação

- Validando o e-mail.

```
mkEmail :: String -> String -> String -> Maybe Email
mkEmail from to body
   = case mkAddress from of
       Nothing -> Nothing
       Just fromAddr ->
         case mkAddress to of
           Nothing -> Nothing
           Just toAddr ->
             case mkBody body of
               Nothing -> Nothing
               Just nBody ->
                   Just (Email fromAddr toAddr nBody)
```

Validação

- Código anterior de `mkEmail` é repleto de repetição.
- Lógica real de `mkEmail` poderia ser resumida em:

```
mkEmail from to body
    = Email (mkAddress from)
            (mkAddress to)
            (mkBody body)
```

Validação

- O tipo de `Email` é:

```
Email :: Address -> Address -> Body -> Email
```

Porém, os resultados de `mkAddress` e `mkBody` envolvem o tipo `Maybe`.

Validação

- Idealmente, deveríamos ser capazes de transformar

```
Email :: Address -> Address -> Body -> Email
```

em

```
Email :: Maybe Address ->
         Maybe Address ->
         Maybe Body -> Maybe Email
```

Validação

- Note que a diferença entre os tipos

```
Email :: Address -> Address -> Body -> Email
Email :: Maybe Address ->
         Maybe Address ->
         Maybe Body -> Maybe Email
```

pode ser resolvida por "inserir" a função `Email` em valores de tipo `Maybe`.

Functores Aplicativos

- A abstração de functor aplicativo permite a aplicação de uma função sobre valores contidos em uma "estrutura" de construtor de tipos.

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

Applicative Maybe

```haskell
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    _       <*> Nothing = Nothing
    (Just f) <*> (Just x) = Just (f x)
```

Exercício

- Defina uma instância de Applicative para listas e árvores binárias.

Applicative

- O operador <*> permite a aplicação de funções sobre construtores de tipos.
- Exemplo

```
foo :: Maybe Int
foo = (+) <$> (Just 2) <*> (Just 3)

Prelude*> foo
Just 5
```

Applicative

- Qual o tipo de `(+) <$>` ?
- Lembrando os tipos individuais:

```
(+) :: Num c => c -> c -> c
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

Applicative

- Note que basta fazer `a = c` e `b = c -> c`.
- Com isso, temos:

```
(+) <$> :: (Num c, Functor f) => f c -> f (c -> c)
```

Isso representa que ao aplicarmos essa expressão a um valor de tipo `f c`, o resultado terá tipo `f (c -> c)`.

Applicative

- Dessa forma, temos que `(+) <$> (Just 2)` possui o tipo

```
(+) <$> (Just 2):: (Num c, Functor f) => f (c -> c)
```

Applicative

- Note que podemos combinar a expressão `(+) <$> (Just 2)` com `Just 3` usando o operador `<*>`, cujo tipo é:

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

```
(+) <$> (Just 2) <*> (Just 3) :: Maybe Int
```

Validação

- Usando funções de Applicative, a implementação de mkEmail fica:

```
mkEmail' :: String -> String -> String -> Maybe Email
mkEmail' from to body
   = Email <$> mkAddress from <*>
               mkAddress to   <*>
               mkBody body
```

Validação

- O que fizemos para validar e-mails esta ok...
  - Porém, qual foi a causa da falha?
  - Não há como saber!

Validação

- Idealmente, devíamos armazenar erros obtidos durante a validação para exibí-los ao usuário.
- Usaremos um conjunto de tipos e funções para esse fim.

Validação

- Definindo um tipo para validadores.

```haskell
data Validation err a
   = Failure err
   | Success a
   deriving (Eq, Ord, Show)
```

Validação

- Validation é um Functor

```
instance Functor (Validation err) where
   fmap _ (Failure err) = Failure err
   fmap f (Success x)   = Success (f x)
```

Validação

- Próximo passo: Definir uma instância de Applicative de forma a combinar as falhas.
- Pergunta: Como combinar falhas?
  - Usaremos outra abstração: a de semi-grupo!

Semi-grupo

- Um semi-grupo é uma estrutura algébrica que é formada por um conjunto e uma operação binária associativa.
- Em Haskell, é uma classe de tipos que possui uma função binária que deve ser associativa.

Semi-grupo

- Definição de Semigroup

```
class Semigroup a where
   (<>) :: a -> a -> a
```

Semi-group

- Vamos exigir, na instância de Applicative, que a variável `err` em `Validation err a` seja uma instância de `Semigroup`.

Applicative

```haskell
instance Semigroup err => Applicative (Validation err) where
  pure = Success
  Failure e1 <*> b = Failure $ case b of
    Failure e2 -> e1 <> e2
    Success _ -> e1
  Success _  <*> Failure e2 =
    Failure e2
  Success f  <*> Success a  =
    Success (f a)
```

Erros

```haskell
data Error
  = MustNotBeEmpty
  | MustContain String
  deriving (Eq, Ord, Show)
```

Validação

```haskell
atString :: String -> Validation [Error] Address
atString s
    | '@' `elem` s = Success (Address s)
    | otherwise    = Failure [MustContain "@"]

nonEmptyString :: String -> Validation [Error] Body
nonEmptyString [] = Failure [MustNotBeEmpty]
nonEmptyString s  = Success (Body s)
```

Validação

```
email :: String -> String ->
         String -> Validation [Error] Email
email from to body = Email <$> atString from <*>
                               atString to   <*>
                               nonEmptyString body
```

Exercício

- Qual o kind do tipo a seguir? Apresente uma instância de `Functor` para esse tipo e mostre que essa satisfaz as propriedades anteriormente apresentadas.

```
data Toy a b =
    Output a b
  | Bell b
  | Done
```