Funtores aplicativos

Programação Funcional

Prof. Maycon Amaro

Na aula anterior...

- Vimos construtores de tipos (lista, opcional, tupla)

```
x :: Maybe Int
x = Just 5
```

- Vimos como aplicar funções a valores que estão *contextualizados* por um construtor de tipos

```
ghci> fmap (>0) (Just 5)
Just True
```

Mas e se a função também estiver em algum tipo de estrutura?

```haskell
f :: Maybe (Int -> Bool)
f a = a > 0

x :: Maybe Int
x = Just 5
```

Precisaríamos extrair a função para poder aplicar

```
extrair :: Maybe (Int -> Bool) -> Maybe Int -> Maybe Bool
extrair (Just f) (Just x) = Just (f x)
extrair Nothing  (Just x) = Nothing
extrair (Just f)  Nothing = Nothing
extrair Nothing   Nothing = Nothing
```

Este padrão é tão comum que pode ser generalizado.

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  GHC.Base.liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Note que apenas instâncias de `Functor` podem se tornar instâncias de `Applicative`. A biblioteca também oferece um sinônimo para `fmap` na forma do operador `<$>`.

Este é nosso operador de interesse

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

Dada uma função em uma estrutura e um valor na mesma estrutura, podemos manter a estrutura inalterada e realizar a aplicação

```
ghci> Just even <*> Just 5
Just False
ghci> [(+1)] <*> [1, 2, 3, 4, 5]
[2, 3, 4, 5, 6]
```

A função `pure` também é interessante

```
pure :: a -> f a
```

Nos permite colocar um valor ou função dentro de uma estrutura

```
ghci> fmap even (Just 5)
Just False
ghci> pure even <*> Just 5
Just False
```

Todos esses exemplos nos mostram que `Maybe` é um **funtor aplicativo**.

Listas são funtores aplicativos

Uma lista de funções aplicada a uma lista de valores aplica cada função a cada valor

```
ghci> [even, (>=0)] <*> [1, 2]
[False, True, True, True]
```

A composição de <*> também é muito interessante

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

O caso da Tupla

Vimos que tuplas com o primeiro tipo fixado são funtores

```
ghci> fmap even ("Amarelo", 5)
("Amarelo", False)
```

O que deveria acontecer se a função estivesse também numa tupla?

```
ghci> ("Vermelho", even) <*> ("Amarelo", 5)
(???, True)
```

Monóides

- Qualquer conjunto (ou tipo de dado) que possua uma operação associativa e um elemento identidade para esta operação, forma um **monóide**
- Os naturais formam um monóide com a soma e o 0.
- Também formam um monóide com a multiplicação e o 1.
- Listas formam um monóide com concatenação e a lista vazia.
- Em outras palavras, monóides são estruturas cujos valores podem ser *mesclados* de alguma forma.

O caso tupla

Se o primeiro tipo fixado da tupla for um Monóide, temos uma solução simples

```
ghci> ("Vermelho", even) <*> ("Amarelo", 5)
("VermelhoAmarelo", True)
```

De fato, o prelúdio define tupla como `Applicative` quando pelo menos o primeiro tipo é um `Monoid`

```
instance Monoid a => Applicative ((,) a)
  -- Defined in ‘GHC.Base’
```

Exemplo

Poderíamos criar um histórico de aplicações de função com isso!

```
ghci> ("Soma ", (+)) <*> ("1 ", 1) <*> ("2", 2)
("Soma 1 2", 3)
```

Observação

- Apesar dos naturais formarem um monóide sob ambas soma e produto, o prelúdio não define uma instância padrão
- Isto implicaria em escolher entre a soma ou o produto, e não há escolha óbvia
- Porém, a biblioteca `Data.Monoid` oferece as estruturas `Sum` e `Product` para encapsularem números sob uma instância de monóide.

```
ghci> (1, even) <*> (7, 3)
error...
ghci> (Sum 1, even) <*> (7, 3)
(Sum {getSum = 8}, False)
ghci> (Product 1, even) <*> (7, 3)
(Product {getProduct = 7},False)
```

Exemplo

Construir a tabela verdade dos operadores lógicos

```
ghci> (&&) <$> [True, False] <*> [True, False]
[True, False, False, False]
ghci> (||) <$> [True, False] <*> [True, False]
[True, True, True, False]
```

Dúvidas?