Construtores de Tipos e Funtores

Programação Funcional

Prof. Maycon Amaro

Lista é um tipo?

Tipo

- Classificação de valores, análogos à conjuntos.
- Tipo `Int` $= \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- Tipo `[Int]` $= \{$[], [0], [1], [0, 1], [-1], $\ldots\}$
- Tipo `[_]` $=$?

Lista é um construtor de tipo!

- Dado um tipo qualquer a, temos que [a] é um tipo.
- Ou seja, o operador [ ] serve para **construir** tipos a partir de outros tipos.
- O **kind** de um tipo é sempre constante (*). O **kind** de um construtor de tipos é análogo a uma função sobre tipos.

```
ghci> :k Int
Int :: *
ghci> :k []
[] :: * -> *
ghci> :k [Int]
[Int] :: *
```

- O **kind** de uma classe de tipos é análoga a uma função sobre um tipo que retorna uma restrição.

```
ghci> :k Eq
Eq :: * -> Constraint
```

- Ou seja, apenas tipos podem ser instâncias de `Eq` e das classes de tipos que vimos na aula anterior.
- Construtores de tipo não podem ser instâncias de `Eq`.

```
ghci> :k Eq []

<interactive> error:
    • Expected a type, but ‘[]’ has kind ‘* -> *’
```

Construtores de tipos como contextualizadores

- O construtor de tipos Lista representa uma estrutura de dados.
- Podemos pensar nisso como um *contexto*.
- O tipo `[Int]` é o tipo `Int` num *contexto* de lista.
- Como manipular dados contextualizados? Álgebra!

Funtores

O que é um funtor?

Um termo que vêm da Álgebra Abstrata, e é definido como um mapeamento entre duas estruturas.

Um mapeamento entre tipos é meramente uma função.

```
even :: Int -> Bool
```

No nosso caso, um Funtor é um mapeamento entre tipos que estão contextualizados.

```
evens :: [Int] -> [Bool]
```

Para listas essa função tem uma implementação muito simples

```
evens :: [Int] -> [Bool]
evens = map even
```

Este padrão é tão comum que pode ser generalizado.

A classe Funtor

- Em Haskell, funtores estão modelados como uma classe de tipos
- Mas ela aplica restrições à **construtores de tipos**

```
ghci> :info Functor
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
```

Para listas, `fmap` e `map` terão o mesmo comportamento

```
evens :: [Int] -> [Bool]
evens = fmap even
```

Porém, `fmap` funciona com qualquer construtor de tipo que seja instância de `Functor`.

Exemplo

Considere o seguinte construtor de tipos.

```
data Opcional a = Algo a | Nada
```

Ele oferece a chance de modelar que um valor é opcional. A função `even` com ele deveria levar em consideração quando não há valor

```
even' :: Opcional Int -> Opcional Bool
even' (Algo x) = Algo (even x)
even'  Nada    = Nada
```

Nosso construtor `Opcional` é um funtor, e por isso podemos facilitar a implementação desta e de muitas outras funções sobre ele:

```
instance Functor Opcional where
  fmap :: (a -> b) -> Opcional a -> Opcional b
  fmap f (Algo a) = Algo (f a)
  fmap f  Nada    = Nada

even' :: Opcional Int -> Opcional Bool
even' = fmap even
```

`Opcional`, `Algo` e `Nada` estão no prelúdio como `Maybe`, `Just` e `Nothing`. O prelúdio já possui a instância de `Functor` para `Maybe`.

Tuplas

Tuplas são construtores de tipos, construindo um novo tipo a partir do produto de dois outros tipos.

```
ghci> :k (,)
(,) :: * -> * -> *
```

Se fixarmos um dos tipos, tuplas se tornam um construtor de tipo com o mesmo **kind** de lista e opcional.

```
ghci> :k (,) Int
(,) Int :: * -> *
```

No prelúdio há instância de `Functor` para tuplas em que o primeiro tipo está fixado.

```
even' :: (Int, Int) -> (Int, Bool)
even' = fmap even
```

De forma análoga, também há instância de Functor para `Either` com o primeiro tipo fixado.

Propriedades de Funtores

Identidade

Utilizar `fmap` com a função de identidade deve se comportar exatamente como a função identidade.

`fmap id` $\equiv$ `id`

Exemplos

```
ghci> fmap id [2, 3, 4, 5, 6, 7]
[2,3,4,5,6,7]
ghci> fmap id (Just "Oi")
Just "Oi"
ghci> fmap id (4, False)
(4,False)
```

Composição

O `fmap` de uma composição de função é a composição dos `fmaps` de cada função.

$\texttt{fmap}\ (f \circ g) \equiv (\texttt{fmap}\ f) \circ (\texttt{fmap}\ g)$

Exemplo

```
ghci> fmap (not . even) [2, 4, 8, 1]
[False,False,False,True]
ghci> (fmap not . fmap even) [2, 4, 8, 1]
[False,False,False,True]
```

Aninhando Funtores

Se tivermos uma lista de tuplas, temos uma combinação de dois funtores

```
x :: [(String, Double)]
x = [("Bloom", 1.5), ("Flora", 5.3), ("Stella", 2.8)]
```

Uso composto de `fmaps`:

```
arred :: [(String, Double)] -> [(String, Int)]
arred = (fmap . fmap) round
```

Exemplo

```
ghci> arred x
[("Bloom",2),("Flora",5),("Stella",3)]
```

Versão ingênua

Mais complicada de entender, mais verbosa e só funciona para o específico caso de lista de tuplas.

```
arred :: [(String, Double)] -> [(String, Double)]
arred [] = []
arred ((s, d):xs) = (s, round d) : arred xs
```

Com composição de `fmaps`, podemos manter a mesma implementação mesmo se trocarmos os construtores de tipos envolvidos.

```
arred :: Maybe [Double] -> Maybe [Int]
arred = (fmap . fmap) round
```

Dúvidas?