Programação Funcional em Haskell

José Romildo Malaquias

BCC222: Programação Funcional

Universidade Federal de Ouro Preto
Departamento de Computação

5 de fevereiro de 2018

Sumário

1 Valores em um Contexto **1-1**
1.1 Valores encapsulados 1-1
1.2 Aplicação de função 1-2
1.3 Funtores 1-2
1.4 Funtores aplicativos 1-3
1.5 Mônadas 1-4

1 Valores em um Contexto

Resumo

É comum que programas operem com valores dentro de um contexto, como por exemplo um componente de uma estrutura de dados, o resultado de uma computação que pode suceder ou falhar, o retorno de uma ação de entrada e saída, o resultado de uma função, etc.

Funtores, funtores aplicativos, funtores alternativos e mônadas são abstrações que permitem manipular valores contextualizados.

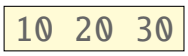
Sumário

1.1 Valores encapsulados 1-1
1.2 Aplicação de função 1-2
1.3 Funtores 1-2
1.4 Funtores aplicativos 1-3
1.5 Mônadas 1-4

1.1 Valores encapsulados

Em muitas aplicações é comum a necessidade de se realizar operações com valores *contidos* em estruturas de dados ou calculados em alguma forma de computação. Dizemos que estes valores estão **encapsulados**. Informalmente dizemos que os valores encapsulados estão *dentro de uma **caixinha***. Se a estrutura não contem nenhum valor dizemos que *a caixnha está vazia*.

São exemplos de valores encapsulados:

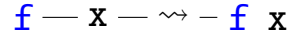
- os elementos de uma lista, tais como
  - `[10, 20, 30]` 
  - `["bom"]` 
  - `[]` 
- o valor opcional em uma estrutura *maybe*, tal como
  - `Just 23` 
  - `Just "bom"` 
  - `Nothing` 
- o retorno de uma ação de entrada e saída
  - `getLine` (retorna a próxima linha da entrada padrão) 

– `getArgs` (retorna os argumentos da linha de comando) `["-c", "prog.o"]`

– `return True` (retorna o valor `True`) `True`

Vamos aprender como operar com valores encapsulados de forma generalizada e abstrata.

1.2 Aplicação de função



Já sabemos como aplicar uma função em valores simples:

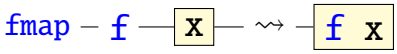
```
even 2     ⇝ True
abs (7-9)  ⇝ 2
```

Podemos também usar o operador binário `$` (precedência 0, associativo à direita) e eliminar alguns parênteses desnecessários:

```
even $ 2   ⇝ True
abs $ 7-9  ⇝ 2
```

1.3 Funtores

Quando queremos aplicar uma função a valores encapsulados, usamos a função `fmap`. O resultado da aplicação da função também será encapsulado. As estruturas e computações que possuem `fmap` são chamadas de **funtores**.



O operador binário `<$>`, com precedência 4 e associatividade à esquerda, é idêntico à função `fmap`.

```
fmap even (Just 2)     ⇝ Just True
fmap abs [7-4, 0 5-1]  ⇝ [3, 0, 4]
fmap length getLine    ⇝ ação que extrai e retorna o tamanho da próxima linha
fmap sqrt Nothing      ⇝ Nothing
fmap sin []            ⇝ []
even <$> return 7      ⇝ ação que retorna False
odd <$> [2, 3, 4]      ⇝ [False, True, False]
```

`fmap` é introduzida na classe `Functor` da biblioteca padrão:

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

Uma estrutura *maybe* é um funtor:

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

Uma estrutura lista também é um funtor:

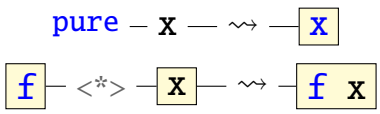
```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = x : fmap f xs
```

Uma ação de entrada e saída também é um funtor:

```
instance Functor IO where
  fmap f acao = do x <- acao
                   return (f x)
```

1.4 Funtores aplicativos

Em algumas situações queremos aplicar uma função a um valor onde ambos estão encapsulados. O resultado da aplicação também será encapsulado. A função <*> permite fazer tal aplicação. Ele é um operador binário associativo à esquerda com precedência 4. As estruturas e computações que possuem <*> são chamadas de **funtores aplicativos**. Elas possuem também uma função chamada pure para encapsular um dado valor.




```
pure even <*> Just 2             ⇝ Just True
pure abs <*> [7-4, 0 5-1]        ⇝ [3, 0, 4]
pure (==) <*> Just 2 <*> Just 3  ⇝ Just False
(==) <$> Just 2 <*> Just 3       ⇝ Just 5
div <$> Just 2 <*> Nothing       ⇝ Nothing
max <$> Nothing <*> Just "belo"  ⇝ Nothing
elem <$> Nothing <*> Nothing     ⇝ Nothing
pure (abs) <*> [8, -5, -1]       ⇝ [8, 5, 1]
[(<0)] <*> [8, -5, -1]           ⇝ [False, True, True]
[(<0), even] <*> [8, -5, -1]     ⇝ [False, True, True, False, True, False]
(+) <$> readLn <*> readLn        ⇝ ação que obtém duas linhas e retorna sua soma
```

As funções pure e (<*>) são introduzidas na classe **Applicative** da biblioteca padrão:

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

Uma estrutura *maybe* é um funtor aplicativo:

```
instance Applicative Maybe where
  pure x = Just x

  Nothing <*> _             = Nothing
  Just f  <*> maybeSomething = fmap f maybeSomething
```

Uma estrutura lista também é um funtor aplicativo:

```haskell
instance Applicative [] where
  pure x = [x]

  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

Uma ação de entrada e saída também é um funtor:

```haskell
instance Applicative IO where
  pure x = return x

  acao1 <*> acao2 = do f <- acao1
                       x <- acao2
                       return (f x)
```

1.5 Mônadas

Podemos compor duas estruturas ou comptações que são funtores aplicativos sequencialmente, de forma que o valor encapsulado na primeira pode ser usado para montar a segunda do sequenciamento. Ou seja, a segunda estrutura ou computação depende dos valores encapsulados na primeira. Tais estruturas ou computações são chamadas de **mônadas**.

O sequenciamento monádico pode ser obtido através de funções definidas na classe `Monad` da biblioteca padrão:

- `>>=` é um operador binário infixo associativo à esquerda e com precedência 1, que combina sequencialmente duas estruturas ou computações
  - o primeiro operando é a primeira estrutura ou computação
  - o segundo operando é *uma função* que, quando aplicada a um valor, resulta na segunda estrutura ou computação da sequência
  - o resultado da operaçao é a segunda estrutura ou computação, sendo que a segunda estrutura ou computação é obtida pela aplicação do segundo operando no valor encapsulado pelo primeiro operando

- `>>` é um operador binário infixo associativo à esquerda e com precedência 1, que combina sequencialmente duas estruturas ou computações
  - o primeiro operando é a primeira estrutura ou computação
  - o segundo operando é a segunda estrutura ou computação da sequência
  - o resultado da operaçao é a segudna estrutura ou computação

  Observe que o valor encapsulado na primeira estrutura ou computação é completamente ignorado.

- `return` é uma função que encapsula um valor. Na grande maioria dos casos coincide com a função `pure`.

Exemplos:

```haskell
return 5 :: [Int]
⇝ [5]

putStr "nome: " » getLine
⇝ ação que exibe uma mensagem e retorna a próxima linha

getLine >= \x -> print (length x)
⇝ ação que lê a próxima linha e exibe o seu tamanho

getLine >= (print .  length)
⇝ ação que lê a próxima linha e exibe o seu tamanho

readLn >= \x -> readLn -> \y -> return (x+y)
⇝ ação que lê a próxima linha e exibe o seu tamanho
```

As funções `return` , `(>=)` e `(»)` são introduzidas na classe **Monad** da biblioteca padrão:

```haskell
class (Applicative m) => Monad m where
  return :: a -> f a
  (>=)  :: m a -> (a -> m b) -> m b
  (»)   :: m a -> m b -> m b

  return = pure

  p » q = p >= (\_ -> q)
```

Uma estrutura *maybe* é uma mônada

```haskell
instance Monad Maybe where
  Nothing >= _ = Nothing
  Just x  >= f = f x
```

Uma estrutura lista também é uma mônada:

```haskell
instance Monad [] where
  xs >= f = [ y | x <- xs, y <- f x ]
```

Uma ação de entrada e saída também é uma mônada

```haskell
instance Monads IO where
  return = operação de um tipo abstrato

  acao1 >= operação de um tipo abstrato
```