

Criando e inspecionando dados

Felipe A. Lessa

Departamento de Ciência da Computação
Universidade de Brasília

Dados

- Definindo
- Desconstruindo
- Sinônimos
- Polimorfismo
- Alguns tipos comuns
- Nomes

Finalizando

- Vam bora!

Assuntos de hoje

1 Dados

- Definindo
- Desconstruindo
- Sinônimos
- Polimorfismo
- Alguns tipos comuns
- Nomes

2 Finalizando

- Vam bora!

Criando novos tipos

- É extremamente útil poder criar novos tipos.
- Em Haskell nós temos
 - **data** para criar tipos.
 - **type** para criar sinônimos.
 - e **newtype** que veremos depois.

Usando data

- Com **data** podemos fazer produtos e somas.
- Produtos são o “e” da lógica, agrupando dois dados.
- Somas são o “ou”, dando uma opção entre dois dados.
- Chega de teoria por hora, vamos ver na prática. =)

Usando data II

- Por exemplo:

```
1 data Aniversario
2   = Nascimento String Int Int Int
3   | Casamento String String Int Int Int
```

- Aqui estamos criando um novo tipo, chamado `Aniversario`.
- Esse novo tipo tem dois *construtores*:
 - O construtor `Nascimento`, com um nome e uma data.
 - O construtor `Casamento`, com o nome do marido, o nome da esposa e uma data.
- A barra vertical denota uma soma: ou você tem um `Nascimento`, ou você tem um `Casamento`.
- Cada construtor faz um produto dos tipos listados.

Usando data III

- Podemos agora criar dados do nosso novíssimo tipo.
- Por exemplo, o aniversário de Carlos Silva:

```
1 carlosSilva :: Aniversario
2 carlosSilva = Nascimento "Carlos_Silva" 28 5 1968
```

- E seu aniversário de casamento com Roberta Silva:

```
1 casamentoSilva :: Aniversario
2 casamentoSilva =
3   Casamento "Carlos_Silva" "Roberta_Silva" 3 12 1987
```

- Podemos também colocá-los numa lista:

```
1 aniversarios :: [Aniversario]
2 aniversarios = [carlosSilva, casamentoSilva]
```

Enumerações

- É possível ter tipos de dados tendo apenas somas, e nenhum produto (ou melhor, um produto vazio).
- A esses tipos damos o nome de *enumeração*, apesar de que a linguagem não faz distinções.
- Um exemplo de enumeração:

```
1 data Naipes = Paus | Copas | Espadas | Ouros
```
- Quando falarmos de classes veremos porque é útil distinguirmos mentalmente as enumerações do resto.

Assuntos de hoje

1

Dados

- Definindo
- **Desconstruindo**
- Sinônimos
- Polimorfismo
- Alguns tipos comuns
- Nomes

2

Finalizando

- Vam bora!

Pattern matching

- Ok, é muito legal e muito bonito criar tipos e dados, mas ainda não vimos como pegar de volta as informações.
- Para “desconstruir” um construtor temos *pattern matching*.
- Em poucas palavras: é do mesmo jeito que construindo =).

Pattern matching II

- Por exemplo, vamos mostrar um Aniversario:

```
1 showNiver :: Aniversario → String
2 showNiver (Aniversario nome d m a)
3   = nome ++ "_nasceu_em_" ++ showData d m a
4 showNiver (Casamento marido esposa d m a)
5   = marido ++ "_casou_com_" ++ esposa ++
6     "_em_" ++ showData d m a

8 showData :: Int → Int → Int → String
9 showData d m a =
10   let d' = show d; m' = show m; a' = show a
11   in d' ++ "/" ++ m' ++ "/" ++ a
```

Pattern matching III

- Isso funciona em qualquer profundidade:

```
1 data Formula = Lit Char
2             | Formula :/\: Formula
3             | Formula :\/: Formula
4             | Neg Formula

6 de_morgan :: Formula → Formula
7 de_morgan (Neg (a :/\: b)) = Neg a :\/: Neg b
8 de_morgan (Neg (a :\/: b)) = Neg a :/\: Neg b
9 de_morgan outra = outra
```

Pattern matching IV

- Também funciona em outros lugares:

```
1 -- Lambdas
```

```
2 cabecas :: [[a]] → [a]
```

```
3 cabecas = map (\(x:xs) → x)
```

```
5 -- Lets
```

```
6 exemplo :: (a → (b, (c, d))) → (a → c)
```

```
7 exemplo f = \x → let (_, (c, _)) = f x in c
```

Observação

- Da forma como construímos nossos tipos até agora, o interpretador não vai mostrá-los:

```
1 *Main> let f = Neg $ Lit 'a' :/\: Lit 'b'
2 *Main> f
```

```
4 <interactive>:1:0:
5   No instance for (Show Formula)
6     arising from use of 'print' at <interactive>:
7   Possible fix: add an instance declaration for
8   In the expression: print it
9   In a 'do' expression: print it
```

- Enquanto não falamos de classes, apenas adicionem “**deriving (Show)**” depois do último construtor.

Observação II

- Nossa Formula ficaria:

```
1 data Formula = Lit Char
2               | Formula :/\: Formula
3               | Formula :\/: Formula
4               | Neg Formula
5               deriving (Show)
```

- E então:

```
1 *Main> let f = Neg $ Lit 'a' :/\: Lit 'b'
2 *Main> f
3 Neg (Lit 'a' :/\: Lit 'b')
4 *Main> de_morgan f
5 Neg (Lit 'a') :\/: Neg (Lit 'b')
```

Nomeando expressões

- Você pode querer reutilizar algumas expressões:

```

1 inter xx@(x:xs) yy@(y:ys)
2   = case x `compare` y of
3       EQ → x : inter xs ys
4       LT → inter xs yy
5       GT → inter xx ys
6 inter _ _ = []

```

- Também funciona em qualquer nível:

```

1 exemplo (_:xx@(x:xs@(_:_)))
2   = if x then xx else xs

```


Assuntos de hoje

1

Dados

- Definindo
- Desconstruindo
- **Sinônimos**
- Polimorfismo
- Alguns tipos comuns
- Nomes

2

Finalizando

- Vam bora!

Meu nome é Well, Manuel.

- Já disse antes que **String** é sinônimo de **[Char]**.
- Beleza, mas como isso funciona? Simples!
- No **Prelude** existe uma declaração

```
1 type String = [Char]
```

- Isso é tudo! =)
- Sinônimos são bons porque não existe overhead.

Mais sobre sinônimos

- Note que ambos são intercambiáveis:

```
1 *Main> :t (++)  
2 (++) :: [a] → [a] → [a]  
3 *Main> let f = (++) :: [Char] → String → [Char]  
4 *Main> :t f  
5 f :: [Char] → String → [Char]
```

- Isso basicamente significa que entre sinônimos o compilador não irá trazer verificações estáticas.

Mais sobre sinônimos II

- Por exemplo, suponha:

```
1 type Nome = String
2 type Cidade = String

4 roberto :: Nome
5 roberto = "Roberto_Campos"

7 rio :: Cidade
8 rio = "Rio_de_Janeiro"
```

Mais sobre sinônimos III

- Então o código abaixo compila sem erro:

```
1 pessoas :: [Nome]
2 pessoas = [roberto, rio]
```

- Mas isso pode ser desejável:

```
1 frase :: String
2 frase = roberto ++ "_mora_no_" ++ rio
```

- Para isso existe **newtype**.

Sinônimo com construtor

- Um **newtype** define um sinônimo com construtor.
- Não existe overhead (internamente é um sinônimo), porém não pode ser confundido com o original.
- Por exemplo, `pessoas` abaixo produz um erro:

```
1 newtype Nome    = Nome String
2 newtype Cidade = Cidade String

4 pessoas :: [Nome]
5 pessoas = [Nome "Roberto_Campos"
6           ,Cidade "Rio_de_Janeiro"]
```

Assuntos de hoje

1

Dados

- Definindo
- Desconstruindo
- Sinônimos
- **Polimorfismo**
- Alguns tipos comuns
- Nomes

2

Finalizando

- Vam bora!

Polimorfismo de novo?

- Também podemos ter polimorfismo em tipos.
- Basta introduzir uma variável de tipo na declaração.
- Por exemplo, um par (equivalente a uma tupla de dois):

```
1 data Par a b = a `Com` b
```

```
3 snd' :: Par a b → b
```

```
4 snd' (Com a b) = b
```

```
6 soma :: Par Int Int → Int
```

```
7 soma (a `Com` b) = a + b
```


Polimorfismo de novo? II

- Uma fórmula parametrizada pelo literal:

```
1 data Formula a = Lit a
2               | Formula a :\/: Formula a
3               | Formula a :/\: Formula a
4               | Neg (Formula a)
5 type OldFormula = Formula Char
```

- Árvore binária:

```
1 data Arvore no folha
2   = No no (Arvore no folha) (Arvore no folha)
3   | Folha folha

5 arv :: Arvore Int String
6 arv = No 10 (Folha "Oi") (Folha "Tchau")
```

Assuntos de hoje

1

Dados

- Definindo
- Desconstruindo
- Sinônimos
- Polimorfismo
- **Alguns tipos comuns**
- Nomes

2

Finalizando

- Vam bora!

Listas

- Podemos definir listas em Haskell assim:

```
1 data Lista a = a `Cons` Lista a | Nil
```

- Em “pseudo-Haskell”, as listas que conhecemos são:

```
1 data [a] = a : [a] | []
```

Básicos

- Booleanos:

```
1 data Bool = False | True
```

- Maybe:

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

- Either:

```
1 data Either a b = Left a | Right b
```

Sim, pattern matching funfa!

- Sabendo as definições, podemos usar pattern matching.
- Por exemplo, com listas:

```
1 zip                                :: [a] → [b] → [(a,b)]
2 zip (x:xs) (y:ys)                 = (x,y) : zip xs ys
3 zip _ _                             = []
```

```
5 map                                :: (a → b) → [a] → [b]
6 map f (x:xs)                      = f x : map f xs
7 map f []                           = []
```

Assuntos de hoje

1

Dados

- Definindo
- Desconstruindo
- Sinônimos
- Polimorfismo
- Alguns tipos comuns
- **Nomes**

2

Finalizando

- Vam bora!

Dando nomes aos bois

- É possível dar nomes aos campos.
- Melhor explicar com um exemplo:

```
1 type Nome = String
2 data Data
3   = Data {dia :: Int, mes :: Int, ano :: Int}
4   deriving (Show)
5 data Aniversario
6   = Nascimento {nome :: Nome
7                 ,data_ :: Data}
8   | Casamento {marido :: Nome
9                ,esposa :: Nome
10               ,data_ :: Data}
11   deriving (Show)
```

Seletores

- Com isso nós ganhamos seletores:

```
1 *Main> let d = Data 16 11 2007
2 *Main> let a = Nascimento "Joao_Silva" d
3 *Main> nome a
4 "Joao_Silva"
5 *Main> mes $ data_ a
6 11
7 *Main> marido a
8 *** Exception: No match in record selec...
9 *Main> :t esposa
10 esposa :: Aniversario → Nome
11 *Main> :t dia
12 dia :: Data → Int
```


Seletores II

- Podemos usar os seletores na construção:

```
1 *Main> let n = "Joao_Silva"
2 *Main> let a = Nascimento {nome = n, data_ = d}
3 *Main> a
4 Nascimento {nome = "Joao_Silva", data_ =
5           Data {dia = 16, mes = 11, ano = 2007}}
```

Seletores III

- E na desconstrução:

```
1 anoNasc :: Aniversario → Int
2 anoNasc (Nascimento {data_ = Data {ano = a}})
3   = a
```

- E na “atualização”:

```
1 anoQueVem :: Data → Data
2 anoQueVem d@(Data {ano = a}) = d {ano = a+1}
```

- Na verdade, o código acima é equivalente a

```
1 anoQueVem (Data d m a) = Data d m (a+1)
```

- ... porém ele funcionará mesmo se, por exemplo, invertermos a ordem dos argumentos para a m d.

Dados

- Definindo
- Desconstruindo
- Sinônimos
- Polimorfismo
- Alguns tipos comuns
- Nomes

Finalizando

- Vam bora!

Estamos chegando lá!

- Vocês já são capazes de fazer muita coisa!
- Sugestão de exercício:

```
1 data Marcacao = Cruz | Bola
2 data JogoDaVelha = ... [[Maybe Marcacao]] ...
3 data ArvoreDeJogo = No JogoDaVelha [Arvore]
```

```
5 inicial :: JogoDaVelha
6 jogadas :: JogoDaVelha → [JogoDaVelha]
7 arvoreDeJogo :: JogoDaVelha → Arvore
```

- E não se esquecem de dar uma olhada no **Prelude!**