Criando e inspecionando dados

Felipe A. Lessa

Departamento de Ciência da Computação
Universidade de Brasília

Assuntos de hoje

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

Finalizando
- Vam bora!

Assuntos de hoje

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

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 Naipe = Paus | Copas | Espadas | Ouros
```

- Quando falarmos de classes veremos porque é útil distinguirmos mentalmente as enumerações do resto.

Assuntos de hoje

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

```
-- Lambdas
cabecas :: [[a]] → [a]
cabecas = map (\(x:xs) → x)

-- Lets
exemplo :: (a → (b,(c,d))) → (a → c)
exemplo f = \x → let (_, (c, _)) = f x in c
```

Observação

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

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

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

- Enquanto não falamos de classes, apenas adicionem "`deriving (Show)`" depois do último construtor.

Observação II

- Nossa `Formula` ficaria:

```
data Formula = Lit Char
             | Formula :/\: Formula
             | Formula :\/: Formula
             | Neg Formula
            deriving (Show)
```

- E então:

```
*Main> let f = Neg $ Lit 'a' :/\: Lit 'b'
*Main> f
Neg (Lit 'a' :/\: Lit 'b')
*Main> de_morgan f
Neg (Lit 'a') :\/: Neg (Lit 'b')
```

Nomeando expressões

- Você pode querer reutilizar algumas expressões:

```
inter xx@(x:xs) yy@(y:ys)
  = case x `compare` y of
      EQ → x : inter xs ys
      LT → inter xs yy
      GT → inter xx ys
inter _ _ = []
```

- Também funciona em qualquer nível:

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

Assuntos de hoje

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

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:

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

- Mas isso pode ser desejável:

```
frase :: String
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:

```
newtype Nome   = Nome String
newtype Cidade = Cidade String

pessoas :: [Nome]
pessoas =  [Nome "Roberto Campos"
           ,Cidade "Rio de Janeiro"]
```

Assuntos de hoje

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

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):

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

snd' :: Par a b → b
snd' (Com a b) = b

soma :: Par Int Int → Int
soma (a `Com` b) = a + b
```

Polimorfismo de novo? II

- Uma fórmula parametrizada pelo literal:

```haskell
data Formula a = Lit a
               | Formula a :\/: Formula a
               | Formula a :/\: Formula a
               | Neg (Formula a)
type OldFormula = Formula Char
```

- Árvore binária:

```haskell
data Arvore no folha
  = No no (Arvore no folha) (Arvore no folha)
  | Folha folha

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

Assuntos de hoje

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

Finalizando
- Vam bora!

Listas

- Podemos definir listas em Haskell assim:

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

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

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

Básicos

- Booleanos:

```haskell
data Bool = False | True
```

- Maybe:

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

- Either:

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

Sim, pattern matching funfa!

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

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

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

Assuntos de hoje

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

Finalizando
- Vam bora!

Dando nomes aos bois

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

```
type Nome = String
data Data
  = Data {dia :: Int, mes :: Int, ano :: Int}
  deriving (Show)
data Aniversario
  = Nascimento   {nome :: Nome
                 ,data_ :: Data}
  | Casamento    {marido :: Nome
                 ,esposa :: Nome
                 ,data_ :: Data}
  deriving (Show)
```

Seletores

- Com isso nós ganhamos seletores:

```
*Main> let d = Data 16 11 2007
*Main> let a = Nascimento "Joao␣Silva" d
*Main> nome a
"Joao␣Silva"
*Main> mes $ data_ a
11
*Main> marido a
*** Exception: No match in record selec...
*Main> :t esposa
esposa :: Aniversario → Nome
*Main> :t dia
dia :: Data → Int
```

Seletores II

- Podemos usar os seletores na construção:

```
*Main> let n = "Joao Silva"
*Main> let a = Nascimento {nome = n, data_ = d}
*Main> a
Nascimento {nome = "Joao Silva", data_ =
          Data {dia = 16, mes = 11, ano = 2007}}
```

Seletores III

- E na desconstrução:

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

- E na "atualização":

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

- Na verdade, o código acima é equivalente a

```
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`.

Assuntos de hoje

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:

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

inicial :: JogoDaVelha
jogadas :: JogoDaVelha → [JogoDaVelha]
arvoreDeJogo :: JogoDaVelha → Arvore
```

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