Programação Funcional e Lógica

Tipos de Dados em Haskell

Prof. Ricardo Couto A. da Rocha
rcarocha@ufg.br
UFG – Regional de Catalão

- *Baseado no curso "Introduction to Haskell" de Brent Yorgey (University of Pennsylvania) E*
- *"Introduction to Functional Programming using Haskell". Richard Bird.*

Leitura de Referência

Aprender Haskell será um grande bem para você - Livro-tutorial online (tradução do livro de Miran Lipovaca)

- **Capítulo 8: Criando seus próprios tipos e typeclasses**

`http://haskell.tailorfontela.com.br/making-our-own-types-and-typeclasses`

Tipos Enumeração

- Haskell permite a criação de tipos enumerados:

```
data DiaDaSemana = Domingo | Segunda | Terca  | Quarta
                 | Quinta | Sexta | Sabado
   deriving Show
```

- **deriving Show** é necessário para indicar ao compilador GHC a geração de código para converter **DiaDaSemana** em String.
  - *Isso será discutido mais tarde.*

```
dia :: DiaDaSemana
dia = Quarta
```

```
listaDeDiasAula :: [DiaDaSemana]
listaDeDiasAula = [Quarta, Sexta]
```

Enumerações

- Funções com casamento de padrões

```
eDiaUtil :: DiaDaSemana -> Bool
eDiaUtil Domingo = False
eDiaUtil Sabado  = False
eDiaUtil Segunda = True
eDiaUtil Terca   = True
eDiaUtil Quarta  = True
eDiaUtil Quinta  = True
eDiaUtil Sexta   = True
```

- Ou de uma maneira mais simples

```
eDiaUtil :: DiaDaSemana -> Bool
eDiaUtil Domingo = False
eDiaUtil Sabado  = False
eDiaUtil _ = True
```

Tipos além de Enumerações

- Enumerações são um caso especial de tipos de dados algébricos mais gerais oferecidos por Haskell.

```
data MedicaoDouble = Falha
                   | OK Double
  deriving Show
```

- **MedicaoDouble** tem dois construtores, um deles com um **Double** como argumento.

```
medicao1 = Falha
medicao2 = OK 55.6
```

- Função geradora de medições

```
safeDiv :: Double -> Double -> MedicaoDouble
safeDiv _ 0 = Falha
safeDiv x y = OK (x / y)
```

Tipos de Dados

- Função geradora de medições

```
safeDiv :: Double -> Double -> MedicaoDouble
safeDiv _ 0 = Falha
safeDiv x y = OK (x / y)
```

- Uso com casamento de padrões

```
falhaParaMedicao :: MedicaoDouble -> Double
falhaParaMedicao Falha = 0
falhaParaMedicao (OK d)  = d
```

Tipos de Dados

- Construtores de dados podem ter mais de um argumento.

```
data Pessoa = Pessoa String Int DiaDaSemana
  deriving Show
aluno1 :: Pessoa
aluno1 = Pessoa "Lucas" 19 Sexta
```

```
aluno2 :: Pessoa
aluno2  = Pessoa "Dimas" 20 Quarta
```

```
getIdade :: Pessoa -> Int
getIdade (Pessoa _ a _) = a
```

- O construtor de tipo e de dado são chamados **Pessoa**, mas são coisas diferentes. É idiomático em Haskell e comum o uso dessa abordagem.

Tipos de Dados Algébricos

- Tipos de dados algébricos podem ter mais de um construtor.

```
data Pessoa = Pessoa String Int
            | Aluno String Int String
            | Professor String Int String
            | Desconhecido
```

- Construtores de tipos e dados sempre devem iniciar com maiúsculas.

Casamento de Padrões

- Em geral, casamento de padrões tenta encontrar um construtor ao qual um valor é baseado.

```
foo (Pessoa a b) = ...
foo (Aluno a b c)    = ...
foo (Professor a b c) = ...
foo Desconhecido      = ...
```

- Dois requisitos necessários:

  1) Indicar os nomes dos construtores usados

  2) Usar parênteses para indicar padrões que envolvem mais de um tipo de construtor.

Padrões e Aninhamento

- Padrões podem ser aninhados

```
data Pessoa = Pessoa String Int DiaDaSemana
  deriving Show
aluno1 = Aluno "Ana" 19 Sabado
aluno2 = Aluno "Ronaldo" 34 Terca
```

```
cumprimentaChegada :: Pessoa -> String
cumprimentaChegada (Aluno n _ Sabado) = n ++
                    ", wow, você é uma pessoa dedicada!"
cumprimentaChegada (Aluno n _ _) = n ++ ", bom dia!"
```

```
*Main> cumprimentaChegada aluno1
"Ana, wow, você é uma pessoa dedicada!"
*Main> cumprimentaChegada aluno2
"Ronaldo, bom dia!"
```

Sintaxe de Registro

- A estrutura de um dado pode perder em legibilidade

```
data Pessoa = Pessoa String Int DiaDaSemana
  deriving Show
aluno1 = Aluno "Ana" 19 Sabado
aluno2 = Aluno "Ronaldo" 34 Terca
```

- Para obter os diversos atributos deveríamos implementar funções próprias

```
nome :: Pessoa → String
nome (Pessoa nome _ _) = nome
idade :: Pessoa  -> Int
idade (Pessoa _ idade _) = idade
```

Sintaxe de Registro

- Sintaxe de Registro oferece um mecanismo alternativo de definição da estrutura dos dados e geração das funções de acesso aos atributos

```
data Pessoa = Pessoa { nome :: String,
                       idade :: Int,
                       entrada ::
DiaDaSemana
                     } deriving Show
```

- Funções `nome`, `idade` e `entrada` são automaticamente geradas pelo Haskell

Sintaxe de Registro

- A sintaxe de registro oferece ainda um mecanismo alternativo de inicialização de dados

```
p :: Pessoa
p = Aluno {nome=”Marcio”, idade=30,
entrada=Segunda}
```

Expressões Case e Padrões

- Em Haskell, a construção **case** é fundamental para realização de casamento de padrões.

```
case exp of
  pat1 -> exp1
  pat2 -> exp2
  ...
```

- **exp** é avaliado quanto a cada um dos padrões, retornando o respectivo padrão casado. Qual será o valor de **expressao** abaixo? Procure entender o significado de cada linha.

```
expressao = case "Hello" of
              []      -> 3
              ('H':s) -> length s
              _       -> 7
```

Expressões Case

- Substitua a definição da função abaixo, usando casamento de padrões com **case**.

```
falhaParaMedicao :: MedicaoDouble -> Double
falhaParaMedicao Falha = 0
falhaParaMedicao (OK d)  = d
```

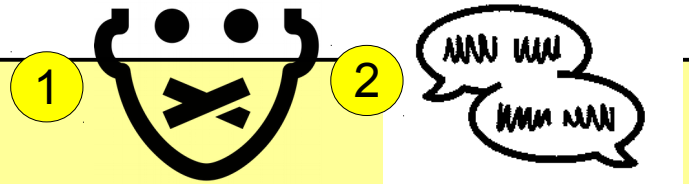
```
falhaParaMedicao :: MedicaoDouble -> Double
falhaParaMedicao x = case x of
                       Falha -> 0
                       (OK d)  -> d
```

Tipos de Dados Recursivos

- Tipos de dados podem ser recursivos, ou seja, definidos em termos deles mesmos.
- Exemplo de uma definição de lista de inteiros.

```
data IntList = Empty | Cons Int IntList
```

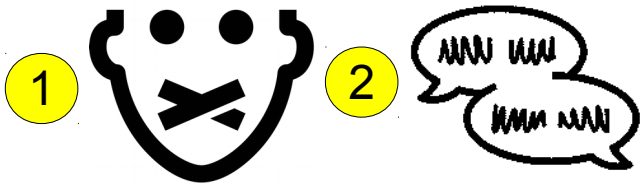
- Tipos Haskell são construídos de maneira similar, mas usando uma sintaxe especial (com `:` e `[]`).
- Construções recursivas usualmente manipulam tipos recursivos. O que faz essa função abaixo? Pensem em silêncio e depois discutiremos.




```
intListProd :: IntList -> Int
intListProd Empty      = 1
intListProd (Cons x l) = x * intListProd l
```

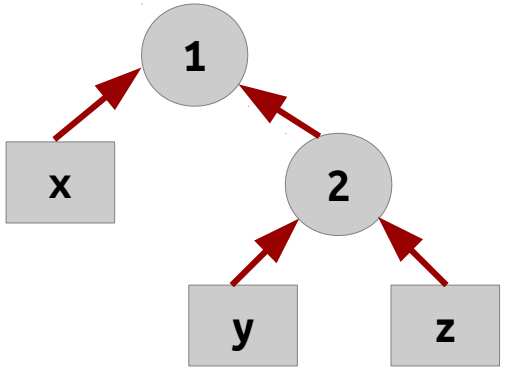
Tipos de Dados Recursivos

- Um tipo de dado árvore binária, com um inteiro (`Int`) armazenado em cada nó interno e uma character (`Char`) em cada folha.
- Pensem em silêncio e depois discutiremos.




```
data Tree = Leaf Char
          | Node Tree Int Tree
  deriving Show
```

- Escreva um código de uma árvore com a seguinte organização




```
tree :: Tree
tree = Node (Leaf 'x')
            1
            (Node (Leaf 'y') 2 (Leaf 'z'))
```

Checagem de Tipos

- **Checagem de tipos** → verificação se os operandos de um operador são de tipos compatíveis
- Tipo compatível é um tipo que ou:
  - Pode ser legalmente aplicado ao operador
  - Ou as regras da linguagem permitem sua conversão implícita para um tipo legal.
- Uma linguagem é **fortemente tipada** se os erros de tipos são sempre detectados (em tempo de compilação ou execução)
- Identificação do uso incorreto de variáveis e valores, na compilação ou execução
  - Todos os operandos tem tipos bem definidos
- Se as amarrações são estáticas, a verificação pode ser toda estática (i.e., na compilação) → **checagem estática de tipos**

Tipos

- Haskell é uma linguagem fortemente tipada.
- O universo de valores é particionado em uma coleção organizada chamada de tipos.
  - **Integer**: coleção de número inteiros
  - **Int**: coleção de números inteiros de **-2.147.483.648** a **2.147.483.647**
- Durante avaliação de uma expressão, os tipos precisam atender àqueles previamente especificados na declaração das funções ou operadores.

Tipos Polimórficos

- Considere o operador de composição funcional (.)

```
(f . x) y = f(x y)
```

- Considere a seguinte composição

```
square . square :: Integer -> Integer
sqrt . square :: Integer -> Float
```

- Os tipos envolvidos são diferentes

```
(.) :: (Integer→Integer) → (Integer→Integer) →
       (Integer->Integer)
(.) :: (Integer→Float) → (Integer→Integer) →
       (Integer->Float)
```

- Virtualmente impossível especificar todos os possíveis tipos para (.) e nem é necessário: Haskell oferece tipos polimórficos.

Tipos Polimórficos

- Especificação do operador (.)

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

- **a**, **b** e **c** são variáveis de tipo, que podem ser associadas a qualquer tipo, mas a relação deve permacener garantida.
  - O valor de retorno da segunda função necessariamente deve ser do mesmo tipo da entrada da primeira
- Exemplo:

Funções Polimórficas

- Considere a definição de uma lista de Integer

```
data IntList = Empty | Cons Int IntList
   deriving Show
```

- Implemente uma função que gera o dobro de uma lista

```
duplica :: Int -> Int
duplica x = 2 * x
```

- Implemente uma função que duplica todos os elementos de uma IntList

```
duplicaTodos :: IntList -> IntList
duplicaTodos Empty = Empty
duplicaTodos (Cons x xs) = Cons (2 * x) (duplicaTodos xs)
```

- **Generalização 1**: Implemente uma função mapListInt que aplica uma função a todos os elementos de uma IntList.

```
mapListInt :: (Int->Int) -> IntList -> IntList
mapListInt f Empty -> Empty
mapListInt f (Cons x xs) -> Cons (f x) (mapListInt f xs)
```

Funções Polimórficas

- Generalize o tipo lista e a função map para se aplicarem a um tipo qualquer.

```
data Lista = Empty | Cons a Lista
  deriving Show
```

- Qual é o problema da definição acima?

```
data Lista t = Empty | Cons t (Lista t)
  deriving Show
```

- Generalize a função mapLista

```
mapLista :: (a->b) -> Lista a -> Lista b
mapLista _ Empty -> Empty
mapLista f (Cons x xs) = Cons (f x) (mapLista f xs)
```

- Generalize a função filterLista (mapeia um elemento em um booleano)

```
filterLista :: (a->Bool) -> Lista a -> Lista a
filterLista _ Empty -> Empty
filterLista f (Cons x xs)
    | f x = Cons x (filterLista f xs)
    | otherwise = filterLista f xs
```

Classe de Tipos

- Operador de (*) pode ser aplicado a inteiros, gerando um inteiro ou a números reais, gerando um número real. Considere as seguintes definições do operador:

```
(*) :: Integer -> Integer -> Integer
(*) :: Float -> Float -> Float
```

- Essas duas definições são abrangentes?
- Qual é o problema com a definição abaixo?

```
(*) :: a -> a -> a
```

- Veja especificação do operador (*)

```
(*) :: Num a => a -> a -> a
```

- Num especifica uma classe de tipos. A especificação indica que a necessariamente deve estar na classe de tipos estabelecida por Num.

Classe de Tipos

- Especificação do operador de igualdade (==)

```
(==) :: Eq a => a -> a -> Bool
```

- A classe de tipos **Eq** especifica que o tipo a precisa ser comparável.
- A classe **Eq** é definida da seguinte maneira

```
class Eq a where
 (==), (/=) :: a -> a -> Bool
```

- Declaração estabelece que:
  - Classe **Eq** contém duas funções membro (==) e (/=)

Classe de Tipos

- Ao declarar um certo tipo que precise fazer parte da classe **Eq**, precisamos
  - Criar uma declaração de instância (instance) e
  - Especificar a definição dos operadores (==) e (/=)
- Exemplo

```
instance Eq Bool where
   (x == y) = (x && y) || (not x && not y)
   (x /= y) = not (x == y)
```

Classe de Tipos

- Podemos aplicar os operadores de (<), (<=), (>=) e (>) a valores booleanos. É possível estabelecer uma ordem entre

```
class (Eq a) => Ord a where
    (<),(<=),(>=),(>) :: a -> a -> Bool
    (x<=y) = (x<y) || (x==y)
    (x>=y) = (x>y) || (x==y)
    (x>y) = not (x <= y)
```

- Um tipo comparável (classe Eq) é automaticamente um tipo ordenável (classe Ord)?
- **Não**! É necessário especificar a operação (<).

```
instance Ord Bool where
    False < False = False
    False < True  = True
    True  < False = False
    True  < True  = False
```

Revisitando Enumeração

- Tipo enumerado **DiaDaSemana**

```
data DiaDaSemana = Domingo | Segunda | Terca  | Quarta
                 | Quinta | Sexta | Sabado
   deriving Show
```

- **deriving Show** é necessário para indicar ao compilador GHC a geração de código para converter **DiaDaSemana** em String.

```
dia :: DiaDaSemana
dia = Quarta
```

```
listaDeDiasAula :: [DiaDaSemana]
listaDeDiasAula = [Quarta, Sexta]
```

Classe Enum

- Classe Enum

```
class Enum a where
    toEnum :: a -> Int
    fromEnum :: Int -> a
```

- Para a função fromEnum vale a relação:
  - `fromEnum(toEnum x) = x`
  - para todo x. Relação não pode ser expressada em Haskell, pois é não construtiva.
- O programador precisa oferecer uma definição construtiva de toEnum.

DiaDaSemana classe de Enum

```
instance Enum DiaDaSemana where
   toEnum Domingo = 0
```

```
instance Eq DiaDaSemana where
   (x == y) = (toEnum x == toEnum y)
```

```
instance Ord DiaDaSemana where
   (x < y) = (toEnum x < toEnum y)
```

```
diaUtil :: DiaDaSemana -> Bool
diaDescanso :: DiaDaSemana -> Bool
```

```
proximoDia :: DiaDaSemana -> DiaDaSemana
proximoDia d = fromEnum ((toEnum d + 1) mod 7)
```

Derivação Automática

- A palavra reservada deriving solicita a derivação automática de código da instância de classe
  - Compilador sabe como gerar o código (quando for possível e definido)
- Exemplo

```
dataDiaDaSemana = ....
    deriving (Eq, Ord, Enum)
```

Sinônimos de Tipos

- Sinônimos de tipos especificam nomes alternativos com os quais um tipo pode ser especificado
- Exemplo para a solução da raiz de equação de segundo grau

```
raizes :: (Float, Float, Float) ->
          (Float, Float)
```

- Redefinição usando sinônimos

```
type Coefs = (Float, Float, Float)
type Raizes = (Float, Float)
raizes :: Coefs → Raizes
```

List Comprehensions

- Notação alternativa, legível e concisa de aplicação de map e filter em lista.
- A notação `[x * x | x <- [1..5], odd x]` deve ser lida como
  - Dada a lista `[1..5]`
  - Para cada elemento `x`, tal que `odd x` (`x` é impar)
  - Gere uma lista com elemento `x*x`
- Retorna `[1,9,25]`
- Equivalente a

```
map (^2) (filter odd [1..5])
```

List Comprehensions

1) Gerar uma lista com os números pares de 1 até 100
2) Gerar uma lista com os números pares e divisíveis por três de 1 até 100
3) Gerar uma lista de *"Ping"* e *"Pong"* a partir de uma lista de inteiros, gerando *"Ping"* para um número par e *"pong"* para um ímpar.
4) Gerar uma lista de todos os pares (tuplas) possíveis entre duas listas, desde que nenhum dos valores do par seja maior que 100 ou menor que -100.
   - Para `l1 = [1,100,101]` e `l2=[-200,0,50]` deve gerar `[(1,0),(1,50),(100,0),(100,50)]`