Tipos Algébricos

Linguagem Haskell

Maria Adriana Vidigal de Lima

Faculdade de Computação - UFU

Setembro - 2009

Fundamentos

- Além da definição de listas e tuplas, podem-se definir novos tipos de dados como:
  - o tipo dos meses: *Janeiro*, *Fevereiro*, ... , *Dezembro*
  - o tipo cujos elementos são inteiros ou strings: 1970 ou *mil novecentos e setenta*
  - o tipo para árvore de dados em que elementos são nós internos ou nós folha

Fundamentos

- A definição destes tipos complexos aumenta a **expressividade** da linguagem e possibilita adicionar estrutura aos valores a serem manipulados nos programas.
- O uso de tipos de dados próprios facilita a programação e proporciona maior segurança através da verificação de tipos (e classes de tipos).

Fundamentos

- Tipos algébricos são definidos começando-se pela palavra **data** seguida do nome do tipo e dos **construtores** do tipo.
- Em geral, um tipo algébrico pode ter um ou mais construtores, e cada construtor de dados pode ter zero ou mais argumentos.

Exemplo:

```
data ExNovoTipo  = Constr1 Tipo11 Tipo12
                 | Constr2 Tipo21
                 | Constr3 Tipo31 Tipo32 Tipo33
                 | Constr4
```

Nesta definição, o valor do tipo `ExNovoTipo` pode ser construído de quatro maneiras: usando `Constr1`, `Constr2`, `Constr3`, or `Constr4`. Dependendo do construtor utilizado, o tipo `ExNovoTipo` pode ou não conter outros valores.

Tipos Enumerados

A forma mais simples de definição de um tipo algébrico é pela enumeração dos elementos do tipo:

```
data Temperatura = Quente | Frio
data Estacao = Verao | Outono | Inverno | Primavera
data Meses = Jan | Fev | Mar | Abr | Mai | Jun | Jul
           | Ago | Set | Out | Nov | Dez
```

Tipos Enumerados

```
data Meses = Jan | Fev | Mar | Abr | Mai | Jun | Jul
           | Ago | Set | Out | Nov | Dez
```

- Construtor de Tipo: Meses
- 12 construtores de dados: Jan, Fev, Mar, Abr, ... Dez
- União disjunta: exatamente um de Jan, Fev, ... Dez
- Tipo enumerado - os construtores de dados não possuem argumentos

Tipos Enumerados

Quando um tipo algébrico é definido, algumas classes podem ser instanciadas diretamente através da palavra reservada **deriving**:

```
data Meses = Jan | Fev | Mar | Abr | Mai | Jun | Jul
           | Ago | Set | Out | Nov | Dez
    deriving (Eq, Show, Enum)
```

Desta maneira, podem-se realizar operações do tipo:

```
Haskell > Set
Set
Haskell > Jun == Jun
True
Haskell > [Jan .. Ago]
[Jan,Fev,Mar,Abr,Mai,Jun,Jul,Ago]
```

Tipos Enumerados

A definição de funções sobre tipos algébricos utiliza **correpondência de padrões**, considerando-se os nomes dos construtores e variáveis dos tipos determinados.

```
data Temperatura = Quente | Frio deriving (Show)
data Estacao = Verao | Outono | Inverno | Primavera

clima::Estacao->Temperatura
clima Inverno = Frio
clima _ = Quente
```

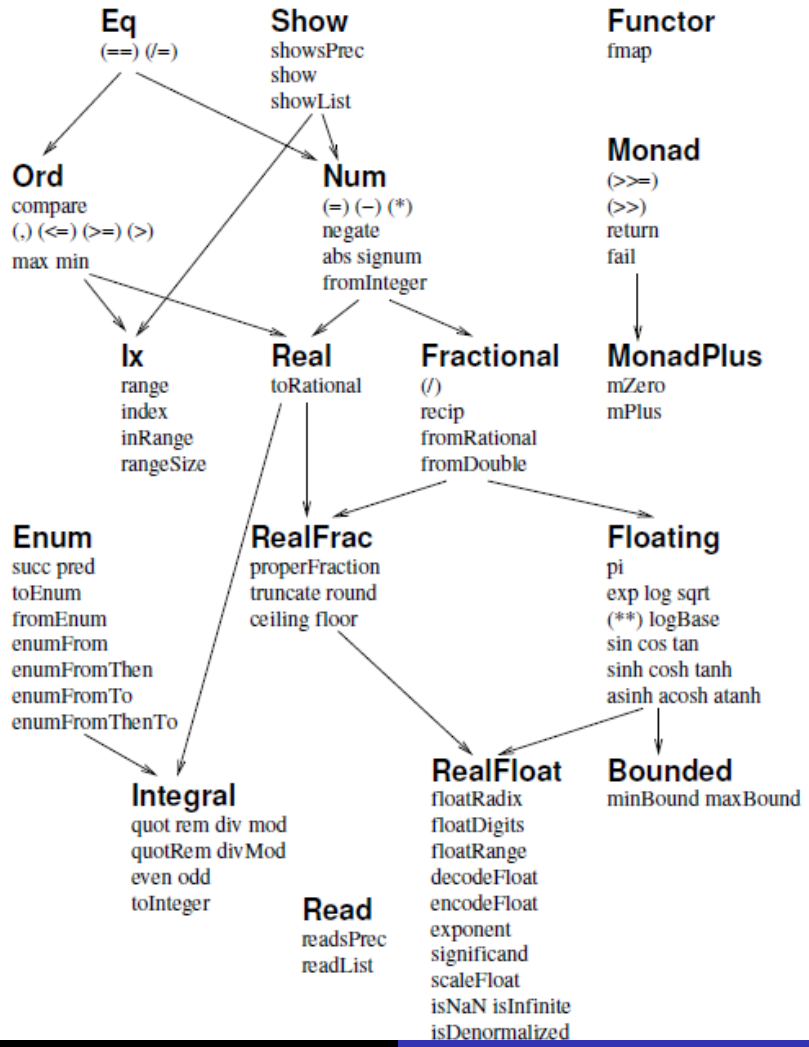
Tipos Enumerados

O tipo Booleano **Bool** é um tipo algébrico, e é definido da seguinte forma:

```
data Bool = False | True
          deriving
          (Eq, Read, Show, Enum, Ord, Bounded)
```

- O tipo booleano **Bool** é uma enumeração e suas operações básicas são && (and), || (or), e `not`.
- A derivação de `Read`, `Show`, `Eq`, `Ord`, `Enum` e `Bounded` permite o uso de operações definidas nestas classes.

Classes e Operações



Produto de Tipos

- Podem ser definidos tipos algébricos como sendo produto de tipos, sendo uma alternativa ao uso de tuplas.
- Este tipo permite a combinação de várias partes de uma informação num único item composto.

```
data Pessoa = Ind Nome Sobrenome AnoNascimento
              deriving (Show)
type Nome = String
type Sobrenome = String
type AnoNascimento = Int
```

Exemplos de valores para o tipo Pessoa poderiam ser:

```
Ind "Stephen" "Hawking" 1942
Ind "Albert" "Einstein" 1879
Ind "Isaac" "Newton" 1643
```

Produto de Tipos

```
data Pessoa = Ind String String Int deriving (Show)

p1,p2,p3::Pessoa
p1 = Ind "Stephen" "Hawking" 1942
p2 = Ind "Albert" "Einstein" 1879
p3 = Ind "Isaac" "Newton" 1643
```

Funções para a manipulação de informações sobre uma Pessoa:

```
primeiroNome :: Pessoa -> String
primeiroNome (Ind pNome _ _ ) = pNome

ultimoNome :: Pessoa -> String
ultimoNome (Ind _ uNome _ ) = uNome

anoNascimento :: Pessoa -> Int
anoNascimento (Ind _ _ vAno ) = vAno
```

Sintaxe de Registros

Ainda, o tipo **Pessoa** pode ser redefinido utilizando-se a sintaxe de registros:

```
data Pessoa = Ind { primNome :: String,
                    ultNome :: String,
                    anoNasc :: Int } deriving (Show)


p2::Pessoa
p2 = Ind {primNome="Stephen", ultNome="Hawking", anoNasc=1942}
```

Esta definição permite a realização das operações:

```
Haskell > primNome p2
"Stephen"
Haskell > anoNasc p2
1942
```

Alternativas

O tipo **Forma** define o círculo como uma tripla de números reais em que o primeiro e o segundo campos são coordenadas do ponto central e o terceiro campo é o valor do raio, e define o retângulo a partir de duas coordenadas.

```
data Forma = Circulo Float Float Float |
             Retangulo Float Float Float Float
     deriving Show

area:: Forma -> Float
area (Circulo _ _ r) = pi * r ^ 2
area (Retangulo x1 y1 x2 y2) = (abs (x2 - x1)) *
                               (abs (y2 - y1))
```

Alternativas

Pode-se refinar a definição de **Forma** a partir da criação de um tipo **Ponto**:

```
data Ponto = Pt Float Float deriving (Show)
data Forma = Circulo Pt Float |
             Retangulo Pt Pt deriving (Show)

area:: Forma -> Float
area (Circulo _ r) = pi * r ^ 2
area (Retangulo (Pt x1 y1) (Pt x2 y2)) = (abs $ x2 - x1) *
                                         (abs $ y2 - y1)
```

Obs: A expressão `(abs $ x2 - x1)` equivale a `(abs (x2 - x1))`

Alternativas

Outras funções usando os tipos **Forma** e **Ponto**:

```
-- mover um objeto Forma sendo 'a' e 'b' valores para a
-- movimentação nos eixos x e y
move:: Forma -> Float -> Float -> Forma
move (Circulo (Pt x y) r) a b = Circulo (Pt (x+a) (y+b)) r
move (Retangulo (Pt x1 y1) (Pt x2 y2)) a b =
      Retangulo (Pt (x1+a) (y1+b)) (Pt (x2+a) (y2+b))

-- criar um círculo na coordenada (0,0)
baseCirculo :: Float -> Forma
baseCirculo r = Circulo (Pt 0 0) r

-- criar um retângulo partindo da coordenada (0,0) até (x,y)
baseRetang :: Float -> Float -> Forma
baseRetang x y = Retangulo (Pt 0 0) (Pt x y)
```

Tipos Recursivos

Tipos são frequentemente descritos em termos de si mesmos:

```
data Expr = Lit Int |
            Add Expr Expr |
            Sub Expr Expr
```

```
2         =>  Lit 2
2+3       =>  Add (Lit 2) (Lit 3)
(3-1)+2   =>  Add (Sub (Lit 3) (Lit 1)) (Lit 2)
```

Dada uma expressão (do tipo Expr) pode-se querer:

- avaliá-la
- convertê-la numa string para que possa ser exibida
- estimar seu tamanho (contar o número de operadores)

Tipos Recursivos

Para tipos recursivos, as funções são definidas usando recursão primitiva: o formato recursivo da função segue o formato recursivo da definição do tipo:

```
data Expr = Lit Int |
            Add Expr Expr |
            Sub Expr Expr

eval::Expr->Int
eval (Lit n) = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)

> eval (Add (Lit 2) (Lit 3))
5
```

Tipos Polimórficos

Tipos algébricos **polimórficos** podem conter variáveis de tipo:

```
data GCarro a b c = GCar {mont::a, modelo::b, ano::c} deriving (Show)
```

Assim, `a`, `b` e `c` são variáveis de tipo e podem assumir tipos diversos em diferentes declarações:

```
c5::(Num a) => GCarro String String a
c5 = GCar {mont="Ford", ano=1967, modelo="Mustang"}

c6:: GCarro String String String
c6 = GCar {ano="sessenta e sete", mont="Ford", modelo="Maverick"}
```

Tipos Polimórficos

Para o tipo `GCarro` pode-se definir a função `frase` seguinte (para quando o ano for numérico):

```
data GCarro a b c = GCar {mont::a, modelo::b, ano::c} deriving (Show)

c5::(Num a) => GCarro String String a
c5 = GCar {mont="Ford", ano=1967, modelo="Mustang"}

frase :: (Show a) => GCarro String String a -> String
frase (GCar {mont=c, modelo=m, ano=y}) =
         "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

Bibliografia

1. *Haskell - Uma abordagem prática*. Cláudio César de Sá e Márcio Ferreira da Silva. Novatec, 2006.
2. *Haskell - The craft of functional programming*. Simon Thompson. Pearson, 1999.