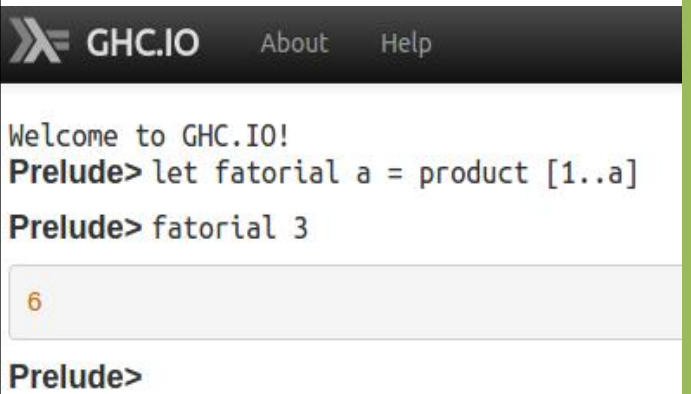


sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14

Programação Funcional

Criação de Tipos

Matheus Carvalho Viana
matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação
UFSJ
Universidade Federal de São João del-Rei

Introdução

Haskell já oferece uma série de tipos que podem ser utilizados para resolver diversas categorias de problemas.

Desde os tipos simples, como os tipos primitivos e as listas, até os tipos compostos, como tuplas e as funções.

Os tipos em Haskell são organizados em hierarquias definidas por classes de tipos. Ou seja, uma classe de tipos possui um ou mais tipos subordinados a ela. Assim todas as operações e funções aplicáveis a uma classe, podem ser aplicadas a todos os tipos que pertencem a essa classe.

Introdução

Já vimos as principais classes de tipos do Haskell e que elas são necessárias na definição de funções polimórficas.

Por exemplo, a função max é uma função polimórfica que funciona para tipos pertencentes à classe Ord.

```
max :: Ord a => a -> a -> a
max x y = if x > y then x else y
```

Além da rica biblioteca de tipos oferecida pela linguagem, Haskell também fornece a possibilidade de criamos nossas próprias classes de tipos e, também, criar novos tipos de dados que podem pertencer a classes de tipos já existentes ou classes que criamos.

Classes de Tipos

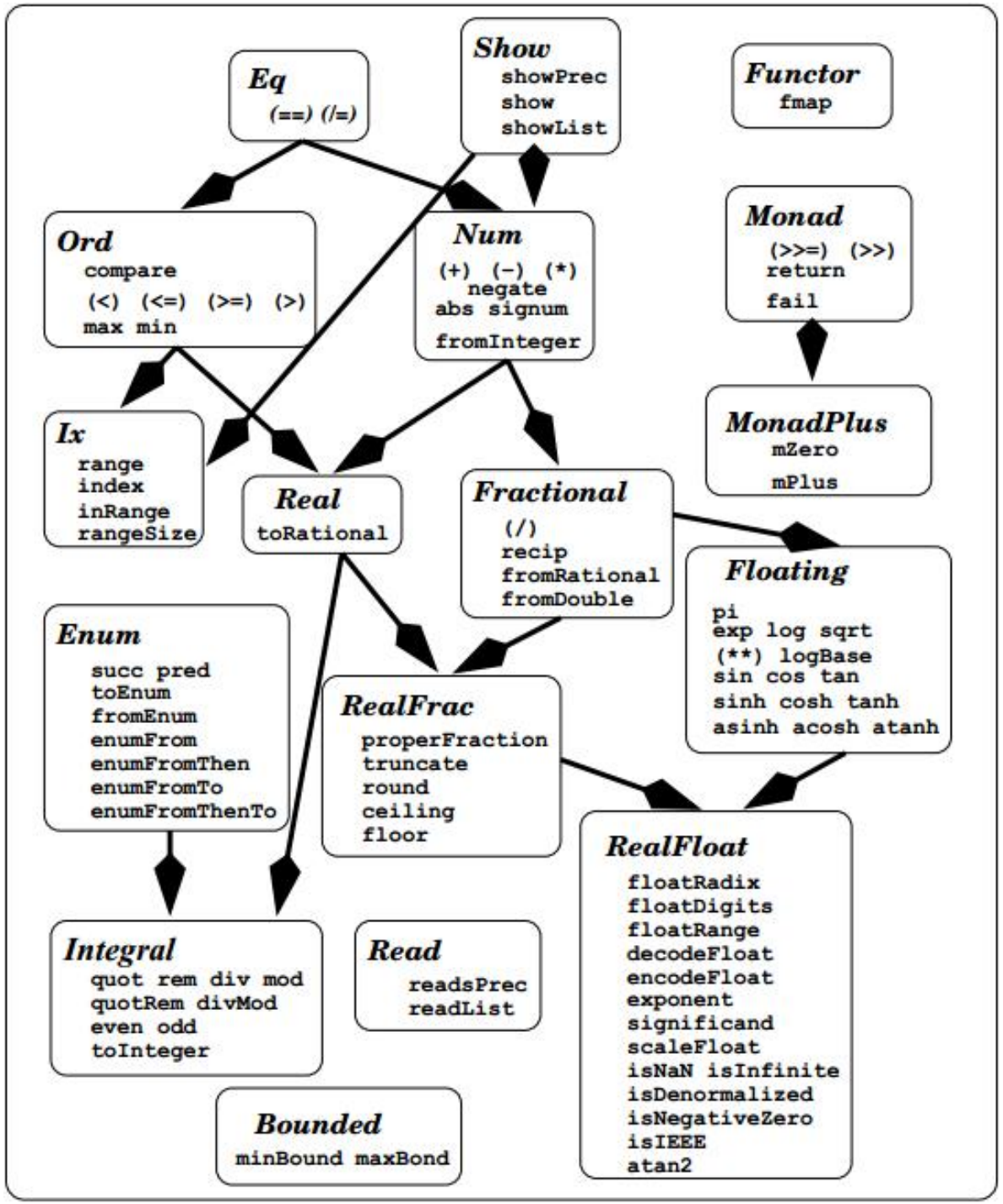
Os tipos sobre os quais uma função pode atuar formam uma coleção de tipos chamada ***classe de tipos*** **(type class)** em Haskell.

Quando um tipo pertence a uma classe, diz-se que ele é uma ***instância*** da classe.

As classes de tipos de Haskell servem para duas coisas:

1. Definir uma hierarquia entre os tipos;
2. Definir uma interface com um conjunto de operações que devem ser implementadas por todas as suas instâncias.

Classes de Tipos Predefinidas



Classes de Tipos

Para criar um nova classe de tipos usa-se a palavra reservada **class**.

Por exemplo, a classe **Visible** transforma cada valor em um **String** e dá a ele um tamanho.

```
class Visible t where
  toString :: t -> String
  size :: t -> Int
```

A definição inclui o nome da classe (**Visible**) e uma ou mais *assinaturas,* que são as funções que compõem a classe juntamente com seus tipos.

Instâncias das Classes de Tipos

Para pertencer a uma classe de tipos, um tipo tem que implementar todas as funções dessa classe. Por exemplo, para pertencer à classe **Visible**, um tipo tem que implementar as funções toString e size.

Por exemplo, para indicar que o tipo **Char** seja uma instância da classe **Visible**, deve-se fazer a seguinte declaração:

```
instance Visible Char where
  toString ch = [ch]
  size _ = 1
```

Essa implementação mostra que um caractere deve ser transformado em uma String de tamanho 1 para que ele seja visível.

Instâncias das Classes de Tipos

Agora é possível aplicar as funções da classe Visible para o tipo Char.

```
*Main> toString '5'
"5"
*Main> size '5'
1
```

Instâncias das Classes de Tipos

De forma similar, para declararmos o tipo **Bool** como uma instância da classe **Visible**, faz-se:

```
instance Visible Bool where
  toString True = "True"
  toString False = "False"
  size _ = 1
```

Para que o tipo **Bool** seja uma instância da classe **Eq**, faz-se:

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
  a /= b = not (a == b)
```

Tipos Algébricos

Já foram vistas várias formas de modelar dados em Haskell.

Por exemplo: funções de alta ordem, polimorfismo, sobrecarga e as classes de tipos.

Também foi visto que os dados podem ser modelados por meio dos seguintes tipos:

- Tipos básicos (primitivos): **Int, Integer, Float, Double, Bool, Char** e listas;
- Tipos compostos: **tuplas (t**1**, t**2**, ..., t**n) e funções (**t**1 -> **t**2**),** onde **t1** e **t2** são tipos.

Tipos Algébricos

Estas facilidades já mostram um grande poder de expressividade e de abstração oferecidos pela linguagem.

No entanto, Haskell oferece uma forma especial para a construção de novos tipos de dados, chamado de **tipo algébrico**, visando modelar situações peculiares, como:

- enumerações;
- tipos compostos (semelhante ao struct de C);
- tipos abstratos de dados.

Tipos Algébricos

Um tipo algébrico permite maior poder de abstração, necessário para modelar estruturas de dados complexas.

Um tipo algébrico define tanto o nome do novo tipo quanto os elementos deste novo tipo.

Cada elemento é nomeado por uma expressão formulada em função dos construtores do tipo. Nomes diferentes denotam elementos também distintos.

Tipos Algébricos

A sintaxe da declaração de um tipo algébrico de dados é:

```
data <nometipo> = <Val1> | <Val2> | ... | <Valn>
```

`<nometipo>` é chamado de **construtor do tipo.**

`<Val1>`, `<Val2>`, etc. são os **valores do tipo**.

Cada valor do tipo é composto de um **construtor de valor** seguido, ou não, de **componentes do valor**.

O primeiro e o último valor definem os limites mínimo e máximo do tipo.

Tipos Algébricos

Por exemplo, a definição do tipo Booleano seria da seguinte forma:

```
data Booleano = Falso | Verdadeiro
```

Nesse caso, `Booleano` é o construtor do tipo e `Falso` e `Verdadeiro` são os valores aceitos por esse tipo.

Cada valor do tipo `Booleano` é composto apenas de um construtor, pois não há componentes.

Tipos Algébricos

No próximo exemplo, `Forma` é o construtor do tipo e `Circulo` e `Retangulo` são os construtores dos valores.

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

Cada valor `Circulo` possui três componentes do tipo `Float`, representando as coordenadas X e Y do centro e o raio.

Cada valor `Retangulo` possui quatro componentes do tipo `Float`, representando as coordenadas X e Y dos pontos superior esquerdo e inferior direito.

Tipos Algébricos

Outra maneira de criar o tipo Forma seria usando tuplas.

```
type Circulo = (Float, Float, Float)
```

Nesse caso, um círculo poderia ser `(43.1, 55.0, 10.4)`.

Parace bom, mas é pouco legível. Três valores em uma tupla também podem representar um vetor 3D ou qualquer outra coisa.

Além disso, usando tuplas, `Circulo` passou a ser um tipo ao invés de um valor do tipo `Forma`.

Nessas situações, um tipo algébrico é mais adequado.

Tipos Algébricos

Na verdade, construtores de valores são funções cujo tipo de retorno é o tipo a qual pertencem.

```
Prelude> :load "algebrico.hs"
[1 of 1] Compiling Main             ( algebrico.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t Circulo
Circulo :: Float -> Float -> Float -> Forma
*Main> :t Retangulo
Retangulo :: Float -> Float -> Float -> Float -> Forma
*Main>
```

Tipos Algébricos

Uma vez criado o tipo, funções podem ser construídas.

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

Para usar essa função no GHCi:

```
> surface (Circulo 1 2 3)
28.274334
```

Observações:
Note que é obrigatório o uso de parênteses nos valores dos tipos algébricos.
O operador $ elimina a necessidade de parênteses nos parâmetros das chamadas de funções.
Por exemplo: `abs $ x2 – x1` é o mesmo que `abs (x2-x1)`.

Tipos Algébricos

Para fazer um tipo algébrico pertencer a uma classe de tipos, é necessário usar o comando **deriving** na criação desse tipo algébrico.

Por exemplo, ao tentar imprimir um valor do tipo `Forma` no GHCi ocasiona em erro, pois o interpretador ainda não sabe como exibir um valor do tipo `Forma` como uma `String`.

```
*Main> Circulo 1 2 3

<interactive>:15:1:
    No instance for (Show Forma) arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
*Main>
```

Tipos Algébricos

Em Haskell, para um tipo conseguir se exibido na tela, é necessário que ele pertença à classe de tipos **Show**.

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

Agora ele funciona:

```
*Main> Circulo 1 2 3
Circulo 1.0 2.0 3.0
```

O mesmo deve ser feito com o tipo Booleano.

```
data Booleano = Falso | Verdadeiro
  deriving (Show)
```

Tipos Algébricos

Repare que não foi necessário implementar uma versão da função `show`, definida na classe de tipos `Show`, para o tipo `Forma`.

Isso corre porque as classes de tipos fornecidas pelo Haskell possuem uma implementação padrão para as suas funções.

```
data DiaSemana =
  Domingo | Segunda | Terca | Quarta | Quinta | Sexta | Sabado
  deriving (Eq, Show)
```

```
*Main> Segunda
Segunda
*Main> Segunda == Segunda
True
```

Tipos Algébricos

Considere o tipo `Pessoa` com nome, idade e altura:

```
data Pessoa = Pessoa String Int Float deriving (Eq, Show)
```

Note que para o tipo `Pessoa` ser uma instância de `Eq` e de `Show`, os componentes dos seus valores também devem ser.

Só é permitido derivar as classes `Eq`, `Ord`, `Enum`, `Ix`, `Bounded`, `Read` e `Show`.

Tipos Algébricos

Considere novamente o tipo `Pessoa`:

```
data Pessoa = Pessoa String Int Float deriving (Eq, Show)
```

Para acessar os componentes, podem ser criadas funções como:

```
nome :: Pessoa -> String
nome (Pessoa n _ _) = n

idade:: Pessoa -> Int
idade (Pessoa _ i _) = i

altura:: Pessoa -> Float
altura (Pessoa _ _ a) = a
```

```
*Main> nome (Pessoa "JOAO" 20 1.80)
"JOAO"
```

Isso funciona, mas é muito trabalhoso para escrever.

Tipos Algébricos

Existe uma maneira melhor de fazer isso, chamada de **sintaxe de registro**.

```
data Pessoa =
  Pessoa {nome :: String, idade :: Int, altura :: Float}
  deriving (Eq, Show)
```

Além de facilitar a escrita, essa sintaxe permite indicar nomes para os componentes de um tipo, aumentando a legibilidade do código.

O funcionamento é o mesmo:

```
*Main> nome (Pessoa "JOAO" 20 1.80)
"JOAO"
```

Tipos Algébricos Polimórficos

As definições de tipos algébricos podem conter tipos variáveis.
Por exemplo, o tipo Par aceita dois componentes de qualquer tipo.

```
data Pares a = Par a a
```

A função igualPar recebe um par e verifica se seus valores são iguais ou não.

```
igualPar :: Eq a => Pares a -> Bool
igualPar (Par x y) = (x == y)
```

```
*Main> igualPar (Par 1 1)
True
*Main> igualPar (Par 1 2)
False
```

Exercícios

1. Como você colocaria os tipos **Int** e **Pares a** como instâncias da classe **Visible**?
2. Crie um tipo algébrico Ponto com as coordenadas X e Y. Modifique o tipo Forma levando em consideração o tipo Ponto. Modifique também a função surface.
3. Crie uma função que recebe dois pontos e retorna um retângulo.
4. Crie uma função que recebe um ponto e uma forma e retorna Verdadeiro se o ponto se encontra dentro da forma ou Falso caso não se encontre. Use o tipo Booleano como retorno.
5. Crie uma função desloca que recebe uma Forma e dois valores Float que correspondem ao deslocamento nos eixos X e Y e retorna a Forma com os eixos dos seus pontos reajustados.