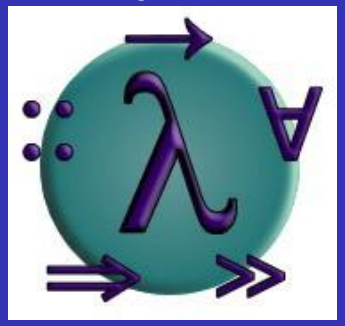
Programação Funcional



Capítulo 12

Classes de Tipos

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2012.1

1 Classes de tipos

Tópicos

Polimorfismo paramétrico

- Já vimos que o sistema de tipos de Haskell incorpora **tipos polimórficos**, isto é, tipos com variáveis quantificadas universalmente (de forma implícita).
- **Exemplos**:
  - Para qualquer tipo `a`, `[a]` é o tipo das listas cujos elementos são do tipo `a`.
  - Dada a declaração

    ```
    data ArvBin a = Vazia | No (ArvBin a) a (ArvBin a)
    ```

    para qualquer tipo `a`, `ArvBin a` é o tipo das árvores binárias com nós do tipo `a`.
- As variáveis de tipo podem ser vista como **parâmetros** dos construtores de tipos e podem ser substituídas por tipos concretos.
- Esta forma de polimorfismo tem o nome de **polimorfismo paramétrico**.

Polimorfismo paramétrico (cont.)

- **Exemplo**:

```
length        :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs
```

```
length [5.6, 7.1, 2.0, 3.8]  ⇝ 4
length ['a', 'b', 'c']       ⇝ 3
length [(3,True), (7,False)] ⇝ 2

:t length                    ⇝ length :: [a] -> Int
```

O tipo

```
[a] -> Int
```

não é mais do que uma abreviatura de

```
∀ a . [a] -> Int
```

ou seja

> *Para todo tipo* `a`*,* `[a] -> Int` *é o tipo das funções com domínio* `[a]` *e contradomínio* `Int`*.*

Polimorfismo *ad hoc* (sobrecarga)

- Haskell incorpora ainda uma outra forma de polimorfismo que é a **sobrecarga de funções**.
- Um mesmo identificador de função pode ser usado para designar funções computacionalmente distintas.
- A esta característica também se chama **polimorfismo *ad hoc***.

Polimorfismo *ad hoc* (sobrecarga) (cont.)

- **Exemplos**:
  - O operador `(+)` tem sido usado para somar, tanto valores inteiros como valores fracionários.
  - O operador `(==)` pode ser usado para comparar inteiros, caractres, listas de inteiros, strings, booleanos, ...
  - Afinal, qual é o tipo de `(+)`? E de `(==)`?
  - A sugestão

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

não serve, pois são tipos demasiado genéricos e fariam com que fossem aceitas expressões como

```
'a' + 'b'
True + False
"está" + "errado"
div == mod
```

e estas expressões resultariam em erro, pois estas operações não estão definidas para trabalhar com valores destes tipos.

Polimorfismo *ad hoc* (sobrecarga) (cont.)

- Em Haskell esta situação é resolvida através de **tipos qualificados** (*qualified types*), fazendo uso da noção de **classe de tipos**.

Tipos qualificados

- Conceitualmente um **tipo qualificado** pode ser visto como um tipo polimórfico, só que, em vez da quantificação universal da forma

  para todo tipo a, . . .

  vai-se poder dizer

  para todo tipo a que pertence à classe C, . . .

- Uma **classe** pode ser vista como um conjunto de tipos.

Tipos qualificados (cont.)

- **Exemplo**:
  Sendo `Num` uma classe (a classe dos números) que tem como elementos os tipos:

  `Int`, **`Integer`**, **`Float`**, **`Double`**, **`Rational`**, ...,

  pode-se dar a `(+)` o tipo

  $$\forall a \in \texttt{Num}.a \to a \to a$$

  o que em Haskell é escrito como:

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

  e lê-se

  *para todo o tipo `a` que pertence à classe **`Num`**, `(+)` tem tipo `a -> a -> a`.*

Tipos qualificados (cont.)

- Desta forma uma classe surge como uma forma de classificar tipos quanto às funcionalidades a ele associadas. Neste sentido as classes podem ser vistas como os tipos dos tipos.
- Os tipos que pertencem a uma classe são chamados de **instâncias** da classe.
- A capacidade de qualificar tipos polimórficos é uma característica inovadora de Haskell.

Classes e Instâncias

- Uma **classe** estabelece um conjunto de assinaturas de funções (os **métodos** da classe).
- Deve-se definir os métodos de uma classe para cada um dos tipos que são instâncias desta classe.

Classes e Instâncias (cont.)

- **Exemplo**:
  - Considere a seguinte **declaração de classe** simplificada:

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
```

Todo tipo a da classe **Num** deve ter as operações (+) e (*) definidas.

Classes e Instâncias (cont.)

- Para declarar `Int` e `Float` como elementos da classe `Num`, tem que se fazer as seguintes **declarações de instância**:

```
instance Num Int where
  (+) = primPlusInt
  (*) = primMulInt
```

```
instance Num Float where
  (+) = primPlusFloat
  (*) = primMulFloat
```

- Neste caso as funções `primPlusInt`, `primMulInt`, `primPlusFloat` e `primMulFloat` são funções primitivas da linguagem.
- Se `x::Int` e `y::Int`, então `x + y` ≡ `primPlusInt x y`.
- Se `x::Float` e `y::Float`, então `x + y` ≡ `primPlusFloat x y`.

Tipo Principal

- O **tipo principal** de uma expressão ou de uma função é o tipo mais geral que lhe é possível associar, de forma que todas as possíveis instâncias desse tipo constituam ainda tipos válidos para a expressão ou função.
- Qualquer expressão ou função válida tem um tipo principal único.
- Haskell **infere** sempre o tipo principal das expressões e funções, mas é sempre possível associar tipos mais específicos (que são instâncias do tipo principal).
- **Exemplo**: O tipo principal inferido pelo haskell para o operador `(+)` é

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

mas,

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```

também são tipos válidos, dado que tanto `Int` como `Float` são instâncias da classe `Num`, e portanto podem substituir a variável de tipo `a`.

Tipo Principal (cont.)

- Note que `Num a` não é um tipo, mas antes uma restrição sobre um tipo. Diz-se que `Num a` é o **contexto** para o tip apresentado.
- **Exemplo**:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

  - O tipo principal da função `sum` é

```
sum :: Num a => [a] -> a
```

  - `sum :: [a] -> a` seria um tipo demasiado geral. Porquê?
  - Qual será o tipo principal da função `product`?

Definições padrão

- Considere a função pré-definida `elem`:

```
elem _ []     = False
elem x (y:ys) = (x == y) || elem x ys
```

  - Qual será o seu tipo?
  - É necessário que `(==)` esteja definido para o tipo dos elementos da lista.
  - A classe pre-definida **`Eq`** é formada pelos tipos para os quais existem operações de comparação de igualdade e desigualdade:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- Minimal complete difinition: (==) or (/=)
  x == y = not (x /= y)
  x /= y = not (x == y)
```

  - Esta classe introduz as funções `(==)` e `(/=)`, e também fornece definições padrão para estes métodos, chamados **métodos *default***.

Definições padrão (cont.)

- Caso a definição de uma função seja omitida numa declaração de instância, o sistema assume a definição padrão feita na classe.
- Se existir uma nova definição do método na declaração de instância, esta definição será usada.

Exemplos de instâncias

- O tipo `Cor` é uma instância da classe `Eq` com `(==)` definido como segue:

```
data Cor = Azul | Verde | Amarelo | Vermelho

instance Eq Cor where
  Azul     == Azul     = True
  Verde    == Verde    = True
  Amarelo  == Amarelo  = True
  Vermelho == Vermelho = True
  _        == _        = False
```

O método `(/=)` utiliza a definição padrão dada na classe `Eq`.

Exemplos de instâncias (cont.)

- O tipo **PontoCor** abaixo também pode ser declarado como instância da classe **Eq**:

```
data PontoCor = Pt Double Double Cor

instance Eq PontoCor where
  (Pt x1 y1 c1) == (Pt x2 y2 c2) = (x1 == x2) &&
                                   (y1 == y2) &&
                                   (c1 == c2)
```

Exemplos de instâncias (cont.)

- O tipo **Nat** também pode ser declarado como instância da classe **Eq**:

```
data Nat = Zero | Suc Nat

instance Eq Nat where
  Zero    == Zero    = True
  (Suc m) == (Suc n) = m == n
  _       == _       = False
```

Instâncias com restrições

- Considere a seguinte definição de tipo para árvores binárias:

```
data ArvBin a = Vazia
              | No (ArvBin a) a (ArvBin a)
```

- Como podemos fazer o teste de igualdade para árvores binárias?
- Duas árvores são iguais se tiverem a mesma estrutura (a mesma forma) e se os valores que estão nos nós também forem iguais.
- Portanto, para fazer o teste de igualdade para o tipo `ArvBin a`, necessariamente tem que se saber como testar a igualdade entre os valores que estão nos nós.
- Só poderemos declarar `ArbBin a` como instância da classe `Eq` se `a` também for uma instância de `Eq`.
- Este tipo de **restrição** pode ser colocado na declaração de instância.

```
instance (Eq a) => Eq (ArbBin a) where
  Vazia          == Vazia          = True
  (No e1 x1 d1) == (No e2 x2 d2) = x1 == x2 && e1 == e2
                                            && d1 == d2
  _              == _              = False
```

Derivação de instâncias

- Os testes de igualdade definidos nos exemplos anteriores implementam a **igualdade estrutural**: dois valores são iguais quando resultam da aplicação do mesmo construtor de dados a argumentos também iguais.
- Nestes casos o compilador pode gerar sozinho a definição da função a partir da definição do tipo.
- Para tanto basta acrescentar a instrução **`deriving Eq`** no final da declaração do tipo:

```
data ArvBin a = Vazia
              | No (ArvBin a) a (ArvBin a)
              deriving (Eq)
```

- Instâncias de algumas outras classes também podem ser derivadas automaticamente.

Herança

- O sistema de classes de Haskell também suporta a noção de **herança**, onde uma classe pode herdar todos os métodos de uma outra classe, e ao mesmo tempo ter seus próprios métodos.
- **Exemplo**: a classe `Ord`:

```
class (Eq a) => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max             :: a -> a -> a
```

- `Eq` é uma **superclasse** de `Ord`.
- `Ord` é uma **subclasse** de `Eq`.
- `Ord` **herda** todos os métodos de `Eq`.
- Todo tipo que é instância de `Ord` tem que ser necessariamente instância de `Eq`.
- Haskell suporta **herança múltipla**: uma classe pode ter mais do que uma superclasse.

A classe **show**

- Define métodos para conversão de um valor para string.
- `Show` pode ser derivada.
- Definição completa mínima: `showsPrec` ou `show`.

```
type ShowS = String -> String
```

```
class  Show a  where
    show      :: a   -> String
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS
```

```
shows :: (Show a) => a -> ShowS
shows =  showsPrec 0
```

A classe **show** (cont.)

- Exemplo:

```
class Horario = AM Int Int Int
              | PM Int Int Int

instance Show Horario where
  show (AM h m s) = show h ++ ":" ++ show m ++ ":" ++ show s
                    ++ " am"
  show (PM h m s) = show h ++ ":" ++ show m ++ ":" ++ show s
                    ++ " pm"
```

A classe Eq

- Define igualdade (==) e desigualdade (/=).
- Todos os tipos básicos exportados por `Prelude` são instâncias de **Eq**.
- **Eq** pode ser derivada para qualquer tipo cujos constituintes são instâncias de **Eq**.
- Definição completa mínima: == ou /=.

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

A classe `Ord`

- Tipos com ordenação total.
- **`Ord`** pode ser derivada para qualquer tipo cujos constituintes são instâncias de **`Ord`**. A ordenação dos valores é determinada pela ordem dos construtores na declaração do tipo.
- Definição completa mínima: `compare` ou `<=`.
- `compare` pode ser mais eficiente para tipos complexos.

```haskell
data Ordering = LT | EQ | GT
```

```haskell
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

A classe Enum

- Define operações em tipos sequencialmante ordenados (enumerações).
- **Enum** pode ser derivada para qualquer tipo enumerado (os construtores de dados são todos constantes). Os construtores são numerados da esquerda para a direita começando com 0.
- Definição completa mínima: `toEnum` e `fromEnum`.

```
class  Enum a   where
    succ                :: a -> a
    pred                :: a -> a
    toEnum              :: Int -> a
    fromEnum            :: a -> Int
    enumFrom            :: a -> [a]
    enumFromThen        :: a -> a -> [a]
    enumFromTo          :: a -> a -> [a]
    enumFromThenTo      :: a -> a -> a -> [a]
```

- As operações da classe **Enum** permitem construir sequências aritméticas.

A classe Enum (cont.)

```
take 5 (enumFrom 'c')        ⇒ "cdefg"
take 5 (enumFromThen 7 10)   ⇒ [7,10,13,16,19]
enumFromTo 'A' 'Z'           ⇒ "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
enumFromThenTo 5 10 38       ⇒ [5,10,15,20,25,30,35]
```

- As sequências aritméticas são abreviações sintáticas para estas operações:

```
take 5 ['c'..]     ⇒ "cdefg"
take 5 [7, 10 ..]  ⇒ [7,10,13,16,19]
['A' .. 'Z']       ⇒ "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
[5,10 .. 38]       ⇒ [5,10,15,20,25,30,35]
```

A classe Num

- Define operações numéricas básicas.
- `Num` não pode ser derivada.
- Definição completa mínima: todos, exceto `negate` ou `(-)`.

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)           :: a -> a -> a
    negate                  :: a -> a
    abs                     :: a -> a
    signum                  :: a -> a
    fromInteger             :: Integer -> a
```

A classe `Num` (cont.)

- Um literal inteiro representa a aplicação da função `fromInteger` ao valor apropriado do tipo **`Integer`**. Portanto o tipo destes literais é `(Num a) => a`.
- Exemplo: `35` é na verdade `fromInteger 35`

```
:t 35
35 :: Num a => a
```

```
35 :: Double   ⇒ 35.0
35 :: Rational ⇒ 35 % 1
```

A classe Real

- Nem todos os números podem ser comparados usando $<$. Exemplo: os números complexos.
- A classe **Real** é formada pelos tipos numéricos para os quais $<$ faz sentido.
- **Real** não pode ser derivada.
- Definição completa mínima: `toRational`.

```
class  (Num a, Ord a) => Real a  where
    toRational           ::  a -> Rational
```

- A idéia é que todo número real de precisão finita pode ser expresso como uma razão de dois números inteiros de precisão arbitrária.

A classe `Integral`

- Números integrais, suportando divisão integral.
- **`Integral`** não pode ser derivada.
- Definição completa mínima: `quotRem` e `toInteger`.

```
class  (Real a, Enum a) => Integral a  where
    quot                    :: a -> a -> a
    rem                     :: a -> a -> a
    div                     :: a -> a -> a
    mod                     :: a -> a -> a
    quotRem                 :: a -> a -> (a,a)
    divMod                  :: a -> a -> (a,a)
    toInteger               :: a -> Integer
```

A classe `Fractional`

- Números fracionários, suportando divisão real.
- **`Fractional`** não pode ser derivada.
- Definição completa mínima: `fromRational` e (`recip` ou `(/)`).

```
class  (Num a) => Fractional a  where
    (/)                 :: a -> a -> a
    recip               :: a -> a
    fromRational        :: Rational -> a
```

- A função de conversão `fromRational` é usada para literais de ponto flutuante.
- Exemplo: `35.1414` é na verdade `fromRational 35.1414`

```
:t 35.7
35.7 :: Fractional a => a
```

```
35.7 :: Float    ⇒ 35.7
35.7 :: Double   ⇒ 35.7
35.7 :: Rational ⇒ 357 % 10
```

A classe `Floating`

- Funções trigonométricas e hiperbólicas, e funções relacionadas.
- **`Floating`** não pode ser derivada.
- Definição completa mínima: `pi`, `exp`, `log`, `sin`, `cos`, `sinh`, `cosh`, `asin`, `acos`, `atan`, `asinh`, `acosh`, `atanh`.

```
class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a
```

A classe `RealFrac`

- Funções para extração de componentes de uma fração.
- **`RealFrac`** não pode ser derivada.
- Definição completa mínima: `properFraction`.

```
class  (Real a, Fractional a) => RealFrac a  where
    properFraction      :: (Integral b) => a -> (b,a)
    truncate            :: (Integral b) => a -> b
    round               :: (Integral b) => a -> b
    ceiling             :: (Integral b) => a -> b
    floor               :: (Integral b) => a -> b
```

A classe `RealFloat`

- Funções para acesso aos componentes de um número em ponto flutuante de forma eficiente e independente da arquitetura do computador.
- **`RealFloat`** não pode ser derivada.
- Definição completa mínima: `exponent`, `significand`, `scaleFloat` e `atan2`.

A classe `RealFloat` (cont.)

```
class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix          :: a -> Integer
    floatDigits         :: a -> Int
    floatRange          :: a -> (Int,Int)
    decodeFloat         :: a -> (Integer,Int)
    encodeFloat         :: Integer -> Int -> a
    exponent            :: a -> Int
    significand         :: a -> a
    scaleFloat          :: Int -> a -> a
    isNaN               :: a -> Bool
    isInfinite          :: a -> Bool
    isDenormalized      :: a -> Bool
    isNegativeZero      :: a -> Bool
    isIEEE              :: a -> Bool
    atan2               :: a -> a -> a
```

Exercícios

Exercício 1

Complete as seguintes declarações de instância:

1. `instance (Ord a, Ord b) => Ord (a,b) where ...`
2. `instance (Ord a) => Ord [a] where ...`

onde pares e listas devem ser ordenadas lexicographicamente, como palavras em um dicionário.

Exercícios (cont.)

Exercício 2

Exercícios (cont.)

Considere o tipo

```
data ArvBin a = Vazia | No (ArvBin a) a (ArvBin a)
```

para representar árvores binárias de busca.

1. Defina uma função que verifica se uma árvore binária é vazia ou não.
2. Defina uma função que recebe um valor e uma árvore binária e insere o valor na árvore binária mantendo-a a ordenada, resultando na nova árvore assim obtida.
3. Defina uma função que recebe um valor e uma árvore binária e verifica se o valor é um elemento da árvore.
4. Modifique a definição do tipo para que sejam criadas automaticamente istâncias desse tipo para as classes **Read** e **Show**.
5. Declare uma instância de **ArvBin** a para a classe **Eq**.
6. Declare uma instância de **ArvBin** a para a classe **Ord**.
7. Declare uma instância de **ArvBin** a para a classe **Functor**. A classe functor tem apenas um método chamado `fmap` que permite mapear uma função aos elementos de uma estrutura de dados, resultando em uma estrutura de dados similar contendo os resultados obtidos pela aplicação da função.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Fim