Programação Funcional
14ª Aula — Classes de tipos revisitadas

Sandra Alves
DCC/FCUP

2019/20

Classes de tipos I

As classes de tipos agrupam tipos de valores que suportam operações comuns.

Eq igualdade (`==`, `/=`)

Ord ordem total (`<`, `>`, `<=`, `>=`)

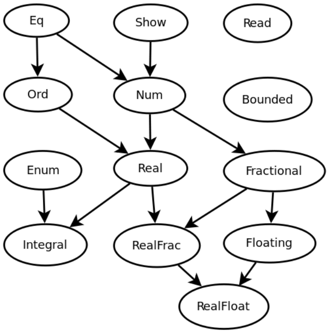
Show conversão para texto (`show`)

Read conversão de texto (`read`)

Num aritmética (`+`, `-`, `*`)

Integral divisão inteira (`div`, `mod`)

Fractional divisão fracionária (`/`)



Classes de tipos II

Atenção:

- as classes de tipos não correspondem às classes da programação com objectos!
- as classes de tipos estão mais próximas do conceito de *interfaces* em Java.

Polimorfismo e sobrecarga I

Em programação funcional dizemos que uma definição que pode ser usada com valores de vários tipos admite um tipo polimórfico.

Exemplo: a função `length`.

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

(Em programação com objectos dir-se-ia que length tem um tipo genérico.)

Polimorfismo e sobrecarga II

As classes de tipos são usadas para implementar a sobrecarga de operadores, isto é, utilização do *mesmo símbolo* para operações semelhantes mas com *diferentes definições*.

```
(==) :: Int -> Int -> Bool         -- tipo Int
(==) :: Float -> Float -> Bool     -- tipo Float
(==) :: String -> String -> Bool   -- tipo String
⋮
```

```
(==) :: Eq a => a -> a -> Bool     -- tipo mais geral
```

Definições de classes

Na definição de uma classe enumerando os nomes e tipos das operações associadas (*métodos*).

Exemplo (do prelúdio-padrão)

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

Apenas declaramos os métodos e tipos — não definimos a implementação para tipos concretos.

Declarações de instâncias

Definimos uma implementação duma classe para tipos concretos usando uma declaração *instance*.

Exemplo: igualdade entre boleanos (no prelúdio-padrão).

```
instance Eq Bool where
   True  == True  = True
   False == False = True
   _     == _     = False        -- todos os outros casos
```

Para `Int` ou `Float` a igualdade é definida usando uma operação primitiva em vez de encaixe de padrões.

Declarar instâncias para novos tipos

Podemos definir implementações das operações das classes para novos tipos de dados. Exemplo:

```
data Nat = Zero | Succ Nat

instance Eq Nat where
   Zero   == Zero   = True                -- caso base
   Succ x == Succ y = x==y                -- caso recursivo
   _      == _      = False               -- diferentes
```

Como `Nat é um tipo recursivo, a definição de igualdade também é recursiva.`

Instâncias derivadas

Em alternativa, podemos derivar automaticamente instâncias de classes quando definimos um novo tipo.

```
data Nat = Zero | Succ Nat
           deriving (Eq)
```

A igualdade derivada é *sintática*, isto é, dois termos são iguais se têm os mesmos construtores e sub-expressões iguais.

A igualdade derivada é equivalente à definição no *slide* anterior.

Instâncias derivadas

Podemos derivar instâncias para: `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, ou `Read`

Se C é da class `Bounded`, o tipo tem que ser uma enumeração (todos os construtores são atómicos) ou ter apenas um construtor.

- A classe `Bounded` define os métodos `minBound` e `maxBound`, que devolvem os elementos minimos e máximos de um tipo (respectivamente, o primeiro e último elemento na enumeração)

Se C é da class `Enum`, o tipo tem que ser uma enumeração.

- Os construtores atómicos são numerado da esquerda para a direita com índices de 0 a $n - 1$

A Class Enum

Relembremos os métodos da classe `Enum`

```
class  Enum a  where
  succ, pred     :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]                -- [n..]
  enumFromThen   :: a -> a -> [a]           -- [n,n'..]
  enumFromTo     :: a -> a -> [a]           -- [n..m]
  enumFromThenTo :: a -> a -> a -> [a]      -- [n,n'..m]
```

Implementações padrão I

Podemos declarar implementações padrão para alguns métodos. Tais implementações são usadas quando uma instância não as definir explicitamente.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool        -- dois métodos
    x == y = not (x/=y)                 -- implementação padrão
    x /= y = not (x==y)                 -- implementação padrão
```

Desta forma não necessitamos de implementar as duas operações: definimos == ou /= e a outra operação fica definida pela implementação padrão.

Implementações padrão II

Sejam `lastCon` e `firstCon` respectivamente o último e o primeiro construtores na enumeração do tipo.

```
enumFrom x            = enumFromTo x lastCon
enumFromThen x y      = enumFromThenTo x y bound
                        where
                          bound | fromEnum y >= fromEnum x = lastCon
                                | otherwise                = firstCon
enumFromTo x y        = map toEnum [fromEnum x .. fromEnum y]
enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Por exemplo

```
data  Semana = Segunda | Terça  | Quarta | Quinta | Sexta | Sabado | Domingo
               deriving (Enum)
```

Temos

```
  [Quarta..]  =  [Quarta, Quinta, Sexta, Sabado, Domingo]
  fromEnum Quinta =  3
```

Restrições de classes I

Podemos impor restrições de classes na declaração de uma nova classe.

Exemplo: tipos com *ordem total* devem também implementar *igualdade* (isto é, `Ord é uma sub-classe de Eq`).

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

Restrições de classes II

Também podemos impor restrições de classes na declaração de novas instâncias.

Exemplo: a igualdade entre listas é definida usando a igualdade entre os elementos das listas.

```
instance (Eq a) => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x==y && xs==ys
    _      == _      = False                  -- diferentes
```

Restrições de classes III

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

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

Estudo dum caso: números racionais

Vamos usar classes de tipos para implementar um tipo de dados para número racionais (isto é, frações de inteiros).

Temos de definir:

- instância de `Show para converter em texto`
- `instâncias de Eq e Ord para as operações de comparação`
- `instâncias de Num e Fractional para as operações aritméticas`

Números racionais I

Um número racional é um par (p, q):

- $p \in \mathbb{Z}$ é o *numerador*;
- $q \in \mathbb{Z} \setminus \{0\}$ é o *denominador*

Normalmente é apresentado como p/q.

Note que esta representação não é unica: há *pares diferentes* que representam o *mesmo número racional*, e.g. $(2, 3)$, $(4, 6)$ e $(6, 9)$ representam $2/3$.

Números racionais II

Em Haskell:

```
data Fraction = Frac Integer Integer
```

Podemos definir operações usando encaixe de padrões; e.g. obter o numerador e denominador duma fração:

```
num, denom :: Fraction -> Integer
num   (Frac p q) = p
denom (Frac p q) = q
```

Construir números racionais I

Vamos definir um operador infixo para construir números racionais.

```
(%) :: Integer -> Integer -> Fraction
```

Vantagens:

1. legibilidade (e.g. escrevemos `1%2` em vez de `Frac 1 2`);
2. permite ocular a representação;
3. permite normalizar a representação.

Construir números racionais II

Para reduzir à fração irredutível dividimos o numerador e denominador pelo *máximo divisor comum*; calculamos o m.d.c. usando o *algoritmo de Euclides*.

```
(%) :: Integer -> Integer -> Fraction
p % q
   | q==0      = error "%: division by zero"
   | q<0       = (-p) % (-q)
   | otherwise = Frac (p`div`d) (q`div`d)
   where d = mdc p q

mdc :: Integer -> Integer -> Integer
mdc a 0 = a
mdc a b = mdc b (a`mod`b)
```

Igualdade e conversão para texto

```
-- pré-condição: as frações são normalizadas
instance Eq Fraction where
    (Frac p q) == (Frac r s) = p==r && q==s

instance Show Fraction where
    show (Frac p q) = show p ++ ('%': show q)
```

Somar e multiplicar frações I

$$\frac{p}{q} + \frac{r}{s} = \frac{p \times s}{q \times s} + \frac{q \times r}{q \times s} \qquad \text{(denominador comum)}$$

$$= \frac{p \times s + q \times r}{q \times s}$$

$$\frac{p}{q} \times \frac{r}{s} = \frac{p \times r}{q \times s}$$

Somar e multiplicar frações II

```
instance Num Fraction where
    (Frac p q) + (Frac r s) = (p*s+q*r) % (q*s)
    (Frac p q) * (Frac r s) = (p*r) % (q*s)
    negate (Frac p q) = Frac (-p) q
    fromInteger n = Frac n 1
```

Comparação de fracções I

$$
\begin{aligned}
\frac{p}{q} \leq \frac{r}{s} &\iff \frac{p}{q} - \frac{r}{s} \leq 0 \\
&\iff \frac{p \times s}{q \times s} - \frac{q \times r}{q \times s} \leq 0 \\
&\iff \frac{p \times s - q \times r}{q \times s} \leq 0 \\
&\iff p \times s - q \times r \leq 0 \qquad \text{(porque } q > 0,\ s > 0\text{)}
\end{aligned}
$$

Comparação de fracções II

É suficiente definirmos um dos operadores de comparação (e.g. <=).

```
instance Ord Fraction where
    (Frac p q) <= (Frac r s) = p*s-q*r<=0
```

Os operadores <, >, >= ficam definidos apartir de <=, == e /= (ver a especificação no prelúdio-padrão).

Combinando as definições

```
module Fraction (Fraction, (%)) where
⋮
```

- Exportamos apenas o tipo `Fraction` e o operador `%` (ocultamos a representação interna)
- Todas as operações aritméticas etc. são exportadas pelas instâncias de classes de tipos
- Para um utilizador do módulo, `Fraction` comporta-se como um tipo pré-definido
- Mais geral: ver o módulo `Ratio na biblioteca padrão Haskell`