Classes e Tipos

Overloading

Em Haskell é possível usar o mesmo identificador para funções computacionalmente distintas. A isto chama-se sobrecarga de funções.

Ao nível do sistema de tipos a sobrecarga de funções é tratada introduzindo o conceito de classe e tipos qualificados. Exemplo:

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

```
> 3 + 2             -> a = Int   que pertence à classe Num
5
> 10.5 + 1.7        -> a = Float que pertence à classe Num
12.2
> 'a' + 'b'         -> Char não perntence à classe Num
error: ...
```

Classes e Instâncias

As classes são uma forma de classificar tipos quanto às funcionalidades que lhe estão associadas.

- Uma classe estabelece um conjunto de assinaturas de funções;
- Os tipos que são declarados como instâncias dessa classe têm que ter essas funções definidas

A declaração (simplificada) da classe Num

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

exige que todo o tipo a da classe Num tenha que ter as funções (+) e (*) definidas.

Para declarar *Int* e *Float* como pertencendo à classe Num, têm que se fazer as seguintes declarações de instância:

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

 instance Num Float where
```

```
        (+) = primPlusFloat
        (*) = primMulFloat
```

Tipos principais

O tipo principal de uma expressão é o tipo mais geral que lhe é possível associar, de forma a que todas as possíveis
instâncias desse tipo constituam ainda tipos válidos para a expressão.

- Toda a expressão válida tem um tipo principal único
- O Haskell infere sempre o tipo principal de uma expressão

Exemplo: Podemos definir uma classe `FigFechada`:

```
class FigFechada a where
        area :: a -> Float
        perimetro :: a -> Float

areaTotal l = sum (map area l)
{-> :type areaTotal-}
areaTotal :: (FigFechada a) => [a] -> Float
```

A classe Eq

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

A classe Ord

```
class (Eq a) => Ord a where
    compare               :: a -> a -> Ordering
    (<), (<=), (>=), (>)  :: a -> a -> Bool
    max, min              :: a -> a -> a
{- Minimal complete definition: (<=) or compare using compare can be more
efficient for complex types -}

    compare x y | x == y    = EQ
                | x <= y    = LT
                | otherwise = GT
    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT
    max x y   | x <= y      = y
```

```
                | otherwise   = x
    min x y     | x <= y      = x
                | otherwise   = y
```

A classe Show

```
class Show a where
    show      :: a -> String
    showsPrec :: Int -> a -> ShowS
    showList  :: [a] -> ShowS
    -- Minimal complete definition: show or showsPrec
    show x          = showsPrec 0 x ""
    showsPrec _ x s = show x ++ s
    showList []     = showString "[]"
    showList (x:xs) = showChar '[' . shows x . showl xs
        where showl []     = showChar ']'
              showl (x:xs) = showChar ',' . shows x . showl xs
```

Classes de construtores de tipos

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

instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

#haskell
#SoftwareEngineering