

# Типове и класове в Haskell

Трифон Трифонов

Функционално програмиране, 2018/19 г.

12 декември 2018 г.

# Видове полиморфизъм в Haskell

В Haskell има два основни вида полиморфизъм.

- **параметричен полиморфизъм** — възможност да създаваме конструкции, които обработват елементи от различни типове по **универсален** начин
  - такива конструкции наричаме **генерични (generic)**
  - параметризират се с **типови променливи**, които могат да приемат произволен тип за стойност
- **ad hoc полиморфизъм** — възможност да създаваме конструкции, които обработват елементи от различни типове по **специфичен** начин
  - такива конструкции наричаме **претоварени (overloaded)**
  - налагат механизъм за **разпределение (dispatch)**, който определя коя специфична реализация на конструкцията трябва да се използва в конкретен случай

# Параметричен полиморфизъм

Генеричните конструкции в Haskell са два вида:

- **генерични типове**, конструирани чрез използване на типови променливи
  - функциите, кортежите и списъците могат да генерични
  - `type UnaryFunction a = a -> a`
  - `type Matrix a = [[a]]`
  - `type Dictionary k v = [(k, v)]`
- **генерични функции**, при които една и съща имплементация работи за различни типове
  - `length :: [a] -> Int`
  - `map :: (a -> b) -> [a] -> [b]`
  - `repeated :: Int -> UnaryFunction a -> UnaryFunction a`
  - `transpose :: Matrix a -> Matrix a`
  - `keys :: Dictionary k v -> [k]`
  - `[] :: [a]`
    - константите са частен случай на функции (функции без параметри)

# Ad hoc полиморфизъм

В Haskell имаме претоварени константи, операции и функции:

- претоварени константи

- 5 може да означава цяло, дробно или комплексно число, в зависимост от контекста
- 5.0 може да означава рационално число, число с плаваща запетая или комплексно число
- `maxBound` е максималната стойност на ограничени типове

- претоварени операции

- `+` може да събира цели, дробни, или комплексни числа
- `/` може да дели рационални, дробни или комплексни числа
- `==` може да сравнява числа, символи, кортежи или списъци

- претоварени функции

- `elem` може да търси елемент в списък от сравними елементи
- `show` може да извежда елемент, който има низово представяне
- `[from..to]` може да генерира списък от елементи от тип, в който имаме линейна наредба

# Класове от типове (typeclasses)

## Дефиниция

**Клас от типове** наричаме множество от типове, които поддържат определен тип поведение, зададено чрез множество от имена на функции и техните типове. Функциите на даден клас наричаме **методи**.

Класовете от типове дават структуриран подход към ad hoc полиморфизма.

## Примери:

- **Eq** е класът от типове, които поддържат сравнение
- **Ord** е класът от типове, които поддържат линейна наредба
- **Show** е класът от типове, чиито елементи могат да бъдат извеждани в низ
- **Num** е класът на всички числови типове

## Дефиниране на класове от типове

```
class <клас> <типова-променлива> where
  {<метод>{,<метод>} :: <тип>}
  {<метод> = <реализация-по-подразбиране>}
```

### Примери:

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

```
class Measurable a where
  size :: a -> Int
  empty :: a -> Bool
  empty x = size x == 0
```

# Класови ограничения

## Дефиниция

Ако  $C$  е клас, а  $t$  е типова променлива, то  $C\ t$  наричаме **класово ограничение**. Множество от класови ограничения наричаме **контекст**.

Класовите ограничения ни дават възможност да дефинираме претоварени функции.

## Примери:

- `elem :: (Eq a) => a -> [a] -> Bool`
- `maximum :: (Ord a) => [a] -> a`
- `(^) :: (Integral b, Num a) => a -> b -> a`
- `larger :: (Measurable a) => a -> a -> Bool`
- `larger x y = size x > size y`

# Дефиниране на екземпляри на клас

## Дефиниция

**Екземпляр** (инстанция) на клас наричаме тип, за който са дефинирани методите на класа.

```
instance <клас> <тип> where
  {<дефиниция-на-метод>}
```

## Примери:

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

```
instance Measurable Integer where
  size 0 = 0
  size n = 1 + size (n `div` 10)
```



## Екземпляри с контекст

Можем да добавяме контекст в дефиницията за екземпляри:

```
instance [<контекст> =>] <клас> <тип> where
  {<дефиниция-на-метод>}
```

Примери:

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (u,t) = x == u && y == t
```

```
instance (Measurable a, Measurable b) => Measurable (a,b) where
  size (x,y) = size x + size y
```

```
instance Measurable a => Measurable [a] where
  size = sum . map size
```

# Наследяване

Можем да дефинираме клас B, който допълва методите на вече съществуващ клас A.

Тогава казваме, че:

- Класът B **наследява** (разширява) класа A
- Класът B е **подклас** (производен клас, subclass) на класа A
- Класът A е **надклас** (родителски клас, superclass) на класа B

## Пример: Стандартен клас Ord

```

class (Eq a) => Ord a where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
  compare              :: a -> a -> Ordering
  compare x y
    | x == y    = EQ
    | x < y     = LT
    | otherwise = GT
  x < y  == compare x y == LT
  x > y  == compare x y == GT
  x == y == compare x y == EQ
  x <= y == compare x y /= GT
  x >= y == compare x y /= LT
  max x y == if x > y then x else y
  min x y == if x < y then x else y

```