

Типове и класове в Haskell

Трифон Трифонов

Функционално програмиране, 2025/26 г.

18–22 декември 2025 г.

Тази презентация е достъпна под лиценза Creative Commons Признание-Некомерсиално-Споделяне на споделеното 4.0 Международен

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)**
 - параметризиране 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] -> Integer`
 - `map :: (a -> b) -> [a] -> [b]`
 - `repeated :: Integer -> 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 последователни елементи от тип, в който имаме операции за следващ и предишън елемент

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 -> Integer
  empty :: a -> Bool
  empty x = size x == 0
```

Класови ограничения

Дефиниция

Ако C е клас, а t е типа 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 е 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 В, който допълва методите на вече съществуващ клас А.
Тогава казваме, че:

- Класът В **наследява** (разширява) класа А
- Класът В е **подклас** (произведен клас, subclass) на класа А
- Класът А е **надклас** (родителски клас, superclass) на класа В

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

```

Множествено наследяване

Даден клас може да наследява едновременно няколко родителски класа.

Примери:

```
class (Ord a, Num a) => Real a where
```

...

```
class (Ord a, Measurable a) => OrdMeasurable a where
    sortByOrder, sortBySize :: [a] -> [a]
```

Сравнение на Haskell с други обектно-ориентирани езици

- Класовете в Haskell съответстват на **абстрактни класове или интерфейси**
- Методите в Haskell съответстват на **чисти виртуални функции**
- Класовете и обектите в C++ **нямат директен еквивалент в Haskell**
 - В Haskell данните винаги са разделени от методите
 - Няма ограничения на достъпа (public, private)
 - Няма понятие за разширяване на тип данни, само на интерфейс
 - Съответно, няма **подтипов полиморфизъм**
- Екземплярите съответстват на **имплементиране (наследяване) на интерфейси**
 - В C++ и Java то може да бъде и множествено
- В Haskell претоварените функции имат **еднозначно определен тип**
 - В C++ сме свободни да пишем функции с едно и също име и различни сигнатури
 - В Haskell сме длъжни да наложим класови ограничения
- В Haskell не можем да правим насилствено преобразуване на тип към даден клас (**casting**)

Потребителски дефинирани типове

В Haskell имаме възможност да дефинираме нови типове данни.

- **data** <тип> [<параметри>] = <дефиниция>
- <тип> трябва да започва с главна буква
- <параметри> е списък от различни типови променливи
- <дефиниция> ::= <елемент> { | <елемент> }
 - <дефиниция> описва различните варианти за елементи на типа
- <елемент> ::= <конструктор> { <тип-на-параметър> }
 - всеки вариант за елемент на типа се описва с уникален <конструктор>
 - <конструктор> трябва да започва с главна буква
 - <конструктор> може да има произволен брой параметри, чиито типове се задават в дефиницията

Изброени типове

Най-простият вид потребителски дефинирани типове са **изброените типове**.

Примери:

- `data Bool = False | True`
- `data Compare = LT | EQ | GT`
- `data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun`
- `today :: Weekday`
- `today = Tue`
- `isWeekend :: Weekday -> Bool`
- `isWeekend Sat = True`
- `isWeekend Sun = True`
- `isWeekend _ = False`

Записи

Друга възможност за потребителски дефинирани типове са **записите**.

- `type Name = String`
- `type Score = Int`
- `data Player = Player Name Score`
 - Да, името на типа може да съвпада с името на (единствения) конструктор
- `katniss :: Player`
- `katniss = Player "Katniss Everdeen" 45`
- `getName :: Player -> Name`
- `getName (Player name _) = name`
- `better :: Player -> Player -> Name`

```
better (Player name1 score1) (Player name2 score2)
| score1 > score2  = name1
| otherwise          = name2
```

Записи с полета

- По същество записите са еквивалентни на кортежите...
- ...които по същество са декартово произведение на типове
- Би било удобно, ако имахме имена на всяко от полетата
- В Haskell има специален синтаксис:
- `{ <поле>{, <поле>}::<тип> {, <поле>{, <поле>}::<тип> } }`
- за всяко от полетата автоматично се дефинира функция селектор
- **Пример:**
 - `data Player = Player { name :: Name, score :: Score }`
 - `name :: Player -> Name`
 - `score :: Player -> Score`

Алтернативи

Можем да дефинираме типове, които обединяват няколко други типа.

Примери:

- `data Shape = Circle { radius :: Double }`
 | `Rectangle { width, height :: Double }`
- `circle :: Shape`
- `circle = Circle 2.3`
- `rect :: Shape`
- `rect = Rectangle 3.5 1.8`
- `area :: Shape -> Double`
- `area (Circle r) = pi * r^2`
- `area (Rectangle w h) = w * h`
- `enlarge :: Shape -> Shape`
- `enlarge (Circle r) = Circle (2*r)`
- `enlarge (Rectangle w h) = Rectangle (2*w) (2*h)`

Автоматични екземпляри на класове

При работа с алгебрични типове се сблъскваме с един недостатък:

- `Circle 2.3 == Circle 4.5` → Грешка!
- `circle` → `circle :: Shape`
- `[Mon..Fri]` → Грешка!
- `Thu < Sat` → Грешка!

За използването на тези елементарни операции се налага ръчно да пишем тривиални екземпляри за класове като `Eq`, `Ord`, `Enum`, `Show`:

```
instance Eq Shape where
```

```
Circle x      == Circle y      = x == y
Rectangle a b == Rectangle c d = (a,b) == (c,d)
-              == -            = False
```

```
instance Show Shape where
```

```
show (Circle x)      = "Circle " ++ show x
show (Rectangle a b) = "Rectangle " ++ show a ++ " " ++ show b
```

Автоматични екземпляри на класове

Haskell има възможност за автоматично извеждане на екземпляри на класовете [Eq](#), [Ord](#), [Enum](#), [Show](#), [Read](#).

- **data** <тип> [<параметри>] = <деконструкция> **deriving** <класове>
- <класове> е кортеж от стандартни класове, екземпляри за които искаме автоматично да бъдат изведени
- **data Weekday** = Mon | Tue | Wed | Thu | Fri | Sat | Sun
deriving ([Eq](#), [Ord](#), [Enum](#), [Show](#), [Read](#))
- [Eq](#): два елемента се считат равни, ако имат равни конструктори с равни параметри
- [Ord](#): елементите се сравняват лексикографски, като конструкторите се считат наредени в реда, в който са деконструирани
- [Enum](#): позволено само за изброени типове
- [Show](#), [Read](#): извежда се/въвежда се конструкторът и след това всеки един от параметрите му

Параметризиирани типове

Потребителските типове могат да бъдат **генерични**, т.е. да зависят от типови параметри.

Примери:

- `data Maybe a = Nothing | Just a
deriving (Eq, Ord, Show, Read)`
- `Just 5 :: Maybe Int`
- `Just "wow" :: Maybe String`
- `Nothing :: Maybe a`
- `Just Nothing :: Maybe (Maybe a)`
- `getAt :: Integer -> [a] -> Maybe a`
- `getAt _ [] = Nothing`
- `getAt 0 (x:_) = Just x`
- `getAt n (_:xs) = getAt (n-1) xs`

Сума на типове

- `data Either a b = Left a | Right b
deriving (Eq, Ord, Show, Read)`
- `Left 3 :: Either Int b`
- `Right 'a' :: Either a Char`

Задача. Да се напише функция, която по даден списък от играчи намира най-добрия резултат, ако е постигнат от единствен играч, или списък от имената на играчите, които са постигнали най-добър резултат.

```
searchBest :: [Player] -> Either Score [Name]
searchBest players
| length bestPlayers == 1 = Left best
| otherwise = Right $ map name bestPlayers
  where best = maximum $ map score players
        bestPlayers = filter ((==best) . score) players
```

Рекурсивни алгебрични типове

Можем да дефинираме типове, позававайки се на самите тях **рекурсивно**.

Пример: `data Nat = Zero | Succ Nat deriving (Eq, Ord, Show, Read)`

- `five = Succ $ Succ $ Succ $ Succ $ Succ Zero`
- `fromNat :: Nat -> Integer`
- `fromNat Zero = 0`
- `fromNat (Succ n) = fromNat n + 1`
- `fromNat five → 5`

Двоични числа

```
data Bin = One | BitZero Bin | BitOne Bin  
        deriving (Eq, Ord, Show, Read)
```

- `six = BitZero $ BitOne $ One`
- `fromBin :: Bin -> Integer`
- `fromBin One = 1`
- `fromBin (BitZero b) = 2 * fromBin b`
- `fromBin (BitOne b) = 2 * fromBin b + 1`
- `succBin :: Bin -> Bin`
- `succBin One = BitZero One`
- `succBin (BitZero b) = BitOne b`
- `succBin (BitOne b) = BitZero $ succBin b`
- `fromBin $ succBin $ succBin six → 8`

Списъци

```
data List a = Nil | Cons a (List a)
             deriving (Eq, Ord, Show, Read)
```

- `l = Cons 1 $ Cons 2 $ Cons 3 $ Nil`
- можем да използваме синтаксиса за записи:

```
data List a = Nil | Cons { listHead :: a, listTail :: List a }
             deriving (Eq, Ord, Show, Read)
```

- `listHead l → 1`
- `fromList :: List a -> [a]`
- `fromList Nil = []`
- `fromList (Cons x t) = x:fromList t`
- `(++) :: List a -> List a -> List a`
- `Nil +++ l = l`
- `Cons h t +++ l = Cons h (t +++ l)`

Двоични дървета

```
data BinTree a = Empty | Node { root :: a, left, right :: BinTree a }
    deriving (Eq, Ord, Show, Read)
```

Примери:

- $t = \text{Node } 3 (\text{Node } 1 \text{ Empty Empty}) (\text{Node } 5 \text{ Empty Empty})$
- $\text{depth} :: \text{BinTree } a \rightarrow \text{Integer}$
- $\text{depth Empty} = 0$
- $\text{depth } (\text{Node } x \text{ l r}) = \max(\text{depth l}, \text{depth r}) + 1$
- $\text{leaves} :: \text{BinTree } a \rightarrow [a]$
- $\text{leaves Empty} = []$
- $\text{leaves } (\text{Node } x \text{ Empty Empty}) = [x]$
- $\text{leaves } (\text{Node } x \text{ l r}) = \text{leaves l} ++ \text{leaves r}$

Функции от по-висок ред за двоични дървета

Трансформиране на двоично дърво ([map](#)):

```
mapBinTree :: (a -> b) -> BinTree a -> BinTree b
mapBinTree _ Empty      = Empty
mapBinTree f (Node x l r) = Node (f x) (mapBinTree f l) (mapBinTree f r)
```

Сиване на двоично дърво ([foldr](#)):

```
foldrBinTree :: (a -> b -> b) -> b -> BinTree a -> b
foldrBinTree _ nv Empty      = nv
foldrBinTree op nv (Node x l r) =
  foldrBinTree op (x `op` foldrBinTree op nv r) l
```

Дървета с произволен брой наследници

Можем да правим **взаимнорекурсивни** дефиниции:

```
data Tree a = Tree { rootTree :: a, subtrees :: TreeList a }
data TreeList a = None | SubTree { firstTree :: Tree a, restTrees :: TreeList a }
```

```
leaf x = Tree x None
tree = Tree 1 $ SubTree (leaf 2)
        $ SubTree (Tree 3 $ SubTree (leaf 4) $ None)
        $ SubTree (leaf 5) $ None
```

```
level :: Integer -> Tree a -> [a]
level 0 (Tree x _)      = [x]
level k (Tree _ ts)     = levelTrees (k - 1) ts
```

```
levelTrees :: Integer -> TreeList a -> [a]
levelTrees _ None         = []
levelTrees k (SubTree t ts) = level k t ++ levelTrees k ts
```

Полиморфни списъци

```
data SExpr = SBool Bool | SChar Char | SInt Int |  
            SDouble Double | SList { list :: [SExpr] }  
            deriving (Eq, Ord, Show, Read)
```

```
expr = SList [SInt 2, SChar 'a', SList [SBool True, SDouble 1.2, SList []]]
```

```
countAtoms :: SExpr -> Integer  
countAtoms (SList sls) = sum $ map countAtoms sls  
countAtoms _           = 1
```

```
flatten :: SExpr -> SExpr  
flatten (SList sls) = SList $ concatMap (list . flatten) sls  
flatten x          = SList [x]
```