LAB 9

Friday, April 29, 2022 1:30 PM

http://learnyouahaskell.com/types-and-typeclasses#believe-the-type

TYPE VARIABLES

OBS: Tipurile se scriu cu literă mare: Char, Int, Bool ...

Ce tip are funcția head? -> 'head' primește ca argument o listă

```
ghci> :t head
head :: [a] -> a
```

↳ variabilă de tip => ∀ tip

↳ funcție polimorfică

POLIMORFISM -> parametric
-> ad-hoc

① PARAMETRIC

-> tipul unei valori conține una sau mai multe variabile de tip, dar fără constrângeri

(ex): id :: a -> a
head :: [a] -> a
[] :: [a]
map :: (a->b) -> [a] -> [b]

=> același implementare pentru ∀ tip

! În diferite contexte, id poate avea tipul Char -> Char, Int -> Int, (Int -> Maybe Bool) -> (Int -> Maybe Bool)

! Dacă o variabilă de tip apare de mai multe ori, atunci ea se înlocuiește cu același tip peste tot

ex: id :: a -> a poate fi: id :: Int -> Int
dar nu: id :: Bool -> Int
a ≠ a

② AD-HOC

↳ atunci când (de ex.) o funcție are implementări diferite pentru tipuri diferite
=> apar constrângeri de tip

ex: (==) face ceva diferit în funcție de tipul pe care îl primește

=> implementări diferite pt. tipuri diferite

Felul în care verificăm egalitatea pt. numere (1==1) este diferit de felul în care verificăm egalitatea pt. string-uri ("Ana" == "Ana")
=> implementări diferite pentru un comportament similar

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

constrângere de tip

TYPECLASSES (and class instances)

Pentru a defini un anumit comportament => typeclass
↳ un fel de interfață

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

constrângere de clasă

=> funcția (==) poate primi 2 argumente cu același tip 'a', care poate fi ∀ tip, cu constrângerea că a trebuie să fie o instanță a clasei Eq, și întoarce un Bool

Eq => o interfață pentru a verifica egalitatea

=> ∀ tip pentru care are sens să verificăm egalitatea poate fi o instanță al clasei Eq

Cum arată o clasă?

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

=> are niște funcții (==), (/=) care trebuie să fie implem. de instanțele ei
=> aceste funcții pot fi implementate direct în clasă => membri impliciți

Cum facem ca un tip să fie instanță a unei clase?

- pt. clasa Eq => • toate tipurile standard din Haskell (mai puțin IO) sunt by-default membri
  • putem să îl adăugăm noi

```
instance Eq Person where
    (==) person1 person2 = firstName person1 == firstName person2 &&
                           lastName person1 == lastName person2
```

↑ tip

Exemplu BST

```haskell
data BST a = Empty | Node a (BST a) (BST a)

instance Eq a => Eq (BST a) where
    Empty == Empty = True
    Node info1 l1 r1 == Node info2 l2 r2 = info1 == info2 && l1 == l2 && r1 == r2
    _ == _ = False
```

=> tipul (BST a) este o instanță a clasei Eq, dacă 'a' este instanță a clasei Eq
=> o nouă constrângere

EXTINDERE DE CLASE

Dacă a este în Eq, atunci a poate fi în Ord dacă definește funcțiile de mai jos.

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

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

=> o clasă poate extinde altă clasă => nu putem să spunem cum să ordonăm elemente dacă nu știm să definim egalitatea între ele

CLASE PREDEFINITE

↳ mai sus

Ord, Show, Read, Num, Eq, ...

CONSTANTE POLIMORFICE

Așa cum am văzut că există tipuri de date polimorfice și funcții polimorfice
=> și constante polimorfice

(ex) 5 :: Num a => a

minBound :: bounded a => a

```haskell
class Constant a where
  one :: a
  five :: a

instance Constant Int where
  one = 1
  five = 5

instance Constant Float where
  one = 1.0
  five = 5.0

instance Constant String where
  one = "one"
  five = "five"
```

CE TREBUIE REȚINUT:

CLASE → predefinite (ex: Eq, Num, Ord...)
→ definite de noi (ex: Inversabile, Constant...)
↳ au specific un **comportament**, descris prin **funcții**
↳ implementate
- în clasă => membri impliciți
- de o instanță

INSTANȚE (ale unor clase)

Atunci când vrem ca un anumit tip să fie o instanță a unei clase =>
→ implementăm funcțiile necesare din acea clasă
→ folosim 'deriving', pentru clasele care pot fi instanțiate automat de Haskell (Ord, Enum, Bounded, Show, Read)