Haskell CheatSheet

Laborator 9

Polimorfism

1. **Parametric**: manifestarea aceluiaşi comportament pentru parametri de tipuri diferite.

   Exemplu:

   ```
   id :: a -> a
   ```

2. **Ad-hoc**: manifestarea unor comportamente diferite pentru parametri de tipuri diferite.

   Exemplu:

   ```
   elem :: (Eq a) => a -> [a] -> Bool
   ```

Observaţie

O funcţie poate conţine ambele tipuri de polimorfism.
Exemplu:

```
lookup :: (Eq a) => a -> [(a,b)] ->
                                Maybe
```

parametric pentru `b` şi ad-hoc pentru `a`

Clase

Clasele din Haskell seamănă mai mult cu conceptul de interfaţă din Java. O clasă reprezintă un set de funcţii care definesc o interfaţă sau un comportament unitar pentru un tip de date.

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

Observaţie

Eq defineşte 2 funcţii, `(==)` şi `(/=)`. Pentru a înrola un tip în clasa **Eq**, ambele funcţii trebuie implementate.

Instanţiere

Considerăm următorul tip:

```
data Point = Point { x :: Integer
                   , y :: Integer
                   }
```

Includem Point în clasa Eq astfel:

```
instance Eq Point where
    Point x1 y1 == Point x2 y2 =
                    x1 == x2 && y1 == y2
    p1 /= p2 = not (p1 == p2)
```

Putem adăuga şi tipuri de date generice într-o clasă.

```
data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

Deriving

Ord, Enum, Bounded, Show, Read

Având în vedere că, uneori, implementările pentru unele clase sunt relativ simple, compilatorul de Haskell poate face automat aceste implementări, dacă este folosit cuvântul cheie deriving.

```
data Point = Point Float Float
    deriving (Show)
data Person = Person
              { firstName :: String
              , lastName :: String
              , age :: Int
              , height :: Float
              , phoneNumber :: String
              , flavor :: String
              } deriving (Show)
data MyList a = Empty | a :-: (MyList a)
    deriving (Show, Read, Ord, Eq)
```

Functor

Putem captura conceptul de containere ,,mapabile"într-o clasă folosind clasa **Functor**. Această clasă are o singură metodă, numită `fmap`, care este generalizarea funcţionalei `map`.

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

instance Functor [] where
    fmap = map

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

Extindere de clase

Haskell permite ca o clasă să extindă o altă clasă. Acest lucru este necesar când dorim ca un tip inclus într-o clasă să fie inclus doar dacă face deja parte dintr-o altă clasă.

```
class Located a where
    getLocation :: a -> (Int, Int)
class (Located a) => Movable a where
    setLocation :: (Int, Int) -> a -> a

data NamedPoint = NamedPoint
    { pointName :: String
    , pointX    :: Int
    , pointY    :: Int
    } deriving (Show)

instance Located NamedPoint where
    getLocation p = (pointX p, pointY p)

instance Movable NamedPoint where
    setLocation (x, y) p = p{ pointX = x
                            , pointY = y}
```