# *Wykład 11*

## *Typy kwalifikowane i klasy typów w Haskellu*

Motywacja

Klasy Eq, Show, Ord, Enum, Bounded, Num

Typy wyższego rzędu

# *Motywacja*

- Jaki powinien być najogólniejszy typ (typ główny, ang. proncipal type) funkcji (+) ?
  - Int -> Int -> Int -- zbyt specyficzny
  - a -> a -> a -- zbyt ogólny
- Dodawanie powinno być zdefiniowane dla wszystkich typów numerycznych Num = {Int, Integer, Float, Double, itd.}.
- Zauważmy, że typ a -> a -> a
  jest skrótem dla ∀ a. a -> a -> a
- *Typy kwalifikowane (ang. qualified types)* pozwalają określić dziedzinę kwantyfikacji:
  ∀ a ∈ Num. a -> a -> a lub ∀ a. Num a => a -> a -> a,
  co w Haskellu jest zapisywane tak: Num a => a -> a -> a

# *Klasy typów*

- “Num” w poprzednim przykładzie jest *klasą typów* (ang. type class). Nie należy jej mylić z konstruktorem typów ani z konstruktorem wartości!

- Klasy typów nie są klasami w sensie programowania obiektowego!

- “Num” jest predykatem opisującym pewną własność typów.

- Klasy typów umożliwiają kontrolowane wprowadzenie mechanizmu *przeciążenia* (ang. overloading) do języka Haskell.

- W standardowej bibliotece Haskella zdefiniowano wiele klas typów; użytkownik może też definiować swoje klasy.

# *Klasy typów*

- Klasa  typów jest zdefiniowana względem pewnej zmiennej, przebiegającej typy.
- Zawiera ona kolekcję funkcji (metod) z podanymi sygnaturami (typami).
- Niektóre z metod mogą mieć zdefiniowane domyślne implementacje.
- Można zdefiniować dowolnie wiele instancji danej klasy typów, implementując wymagane metody.
- Pisząc funkcje dla danej klasy typów, zamiast dla konkretnego typu (jeśli to możliwe), osiągamy wyższy poziom abstrakcji. Nasz kod będzie można wykorzystać dla typów danych, które jeszcze nie są zdefiniowane.

# *Przykład: porównanie*

- Równość nie jest zdefiniowana dla wszystkich typów. Nie można porównywać funkcji.

- Operator (==) w Haskellu ma typ Eq a => a -> a -> Bool. Na przykład:

```
42 == 42                ➔ True
`a` == `a`              ➔ True
`a` == 42               ➔ << błąd typu! >>
                          (różne typy)
(+1) == (\x->x+1)       ➔ << błąd typu! >>
                        (No instance for (Eq (Integer -> Integer)))
```

- Błędy typów są wykrywane w czasie kompilacji (statycznie).

# *Klasa Eq, cd.*

Klasa typów Eq jest zdefiniowana następująco:

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

Ostatnie dwa wiersze  opisują implementacje domyślne dla operatorów klasy. Dzięki temu użytkownik musi zdefiniować tylko jedną z metod.

Typ staje się egzemplarzem (instancją) klasy w wyniku deklaracji.

# *Informacje o typach klas*

Wszelkie informacje można oczywiście znaleźć w dokumentacji, ale w pracy interakcyjnej warto używać polecenia :info
( lub :i, wykład 8, str. 14)

```
Prelude> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
...
```

Pragma MINIMAL informuje, że wystarczy zdefiniować (==) lub (/=). Pionowa kreska oznacza alternatywę, a przecinek – koniunkcję.

# *Klasa Eq, cd.*

Typy zdefiniowane przez użytkownika również mogą być instancjami klasy Eq.

```
data Kolor = Trefl | Karo | Kier | Pik

instance Eq Kolor where
  Trefl == Trefl = True
  Karo  == Karo  = True
  Kier  == Kier  = True
  Pik   == Pik   = True
  _     == _     = False
```

Dzięki implementacjom domyślnym dostępna jest też funkcja /=.

```
*Main> Trefl /= Karo
True
```

# *Klasa Eq, cd.*

Oto bardziej skomplikowany przykład.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
    Leaf a1      == Leaf a2      =  a1 == a2
    Branch l1 r1 == Branch l2 r2 =  l1==l2 && r1==r2
    _            ==       _      =  False
```

Typ `a` wartości przechowywanych w liściach drzewa również musi należeć do klasy Eq.

```
x `mem` []     = False

x `mem` (y:ys) = x==y || x `mem` ys
```

Z powodu porównania x==y Haskell sam wydedukuje ograniczenie na typ.

```
*Main> :t mem

mem :: Eq a => a -> [a] -> Bool
```

# *Automatyczne tworzenie instancji klas*

Haskell umożliwia automatyczne utworzenie instancji klasy `Eq` (i niektórych innych klas: Ord, Enum, Read, Show, Bounded). Należy w definicji typu danych użyć klauzuli deriving.

```
data Kolor = Trefl | Karo | Kier | Pik
                deriving Eq
data Tree a = Leaf a | Branch (Tree a) (Tree a)
                 deriving Eq

*Main> Leaf 1 == Leaf 1
True
*Main> Leaf 1 == Leaf 2
False
```

# *Klasa Show*

Haskell nie potrafi pokazać wartości zdefiniowanych typów.

```
*Main> Trefl


<interactive>:1:0: error:
    * No instance for (Show Kolor) arising from a use of `print'
    * In a stmt of an interactive GHCi command: print it
```

Typ Kolor musi być instancją klasy Show. To też można zrobić automatycznie.

```
data Kolor = Trefl | Karo | Kier | Pik
                deriving (Eq, Show)
data Tree a = Leaf a | Branch (Tree a) (Tree a)
                 deriving (Eq, Show)
*Main> Leaf 5
Leaf 5
```

# *Klasa Ord*

Chcielibyśmy zdefiniować porządek na kolorach i drzewach.

```
*Main> Leaf 4 < Leaf 6


<interactive>:1:0: error:
    * No instance for (Ord (Tree Integer)) arising from a use of `<'
    * In the expression: Leaf 4 < Leaf 6
      In an equation for `it': it = Leaf 4 < Leaf 6
```

Typ Tree a musi być instancją klasy Ord.

```
instance Ord a => Ord(Tree a) where
  Leaf a1      <  Leaf a2     = a1<a2
  Leaf _       <  Branch _ _  = True
  Branch _ _   < Leaf _       = False
  Branch l1 r1 < Branch l2 r2 = l1<l2 || (l1==l2 && r1<r2)
  t1           <= t2          = t1<t2 || t1==t2
*Main> Leaf 4 < Leaf 6
True
```

# *Klasa Ord, cd.*

```
Prelude> :i Ord
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
 {-# MINIMAL compare | (<=) #-}
...
```

Pragma MINIMAL informuje, że wystarczy zdefiniować compare lub (<=).

# *Klasa Ord, cd.*

```
data  Ordering  =  LT | EQ | GT
              deriving  (Eq, Ord, Bounded, Enum, Read, Show)

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

   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

   -- Note that (min x y, max x y) = (x,y) or (y,x)
   max x y | x <= y    =  y
           | otherwise =  x
   min x y | x <= y     =  x
           | otherwise =  y
```

Większość metod ma implementacje domyślne. Wystarczy zdefiniować metodę compare lub <= .

# *Klasa Ord, cd.*

W języku Haskell domyślna równość i porządek (jak w OCamlu) są generowane automatycznie, jeśli zostanie użyta klauzula deriving. **Istotna jest kolejność konstruktorów wartości i ich argumentów.**

```
data Kolor = Trefl | Karo | Kier | Pik
                deriving (Eq, Ord, Show)
data Tree a = Leaf a | Branch (Tree a) (Tree a)
                 deriving (Eq, Ord, Show)

*Main> Trefl < Kier
True
*Main> min Pik Karo
Karo
*Main> Leaf 4 < Branch (Leaf 0) (Leaf 1)
True
*Main> Branch (Leaf 0)(Leaf 2) <=  Branch (Leaf 0)(Leaf 1)
False
```

# *Równość i porządek dla definiowanych typów* - OCaml

W języku OCaml operatory porównania (równości i porządku) są przeciążane dla nowo definiowanych algebraicznych typów danych (patrz wykład 4). **Istotna jest kolejność konstruktorów wartości i ich argumentów.** Porównanie wartości funkcyjnych powoduje zgłoszenie wyjątku Invalid_argument. Porównanie struktur cyklicznych może spowodować zapętlenie.

```
type kolor = Trefl | Karo | Kier | Pik;;
type 'a tree = Leaf of 'a | Branch of 'a tree * 'a tree;;

# Trefl < Kier;;
- : bool = true
# min Pik Karo;;
- : kolor = Karo
# Leaf 4 < Branch (Leaf 0, Leaf 1);;
- : bool = true
# Branch (Leaf 0, Leaf 2) <=  Branch (Leaf 0, Leaf 1);;
- : bool = false
```

# *Klasa Enum*

Na wykładzie 8, str. 27 pokazano lukier syntaktyczny Haskella dla ciągów arytmetycznych. Takie ciągi można tworzyć dla każdego typu danych, który jest instancją klasy `Enum`. Np. zapis [1,3..] to skrót dla wywołania funkcji enumFromThen 1 3.

```
class  Enum a  where
   succ, pred     :: a -> a
   toEnum         :: Int -> a
   fromEnum       :: a -> Int
   enumFrom       :: a -> [a]           -- [n..]
   enumFromThen   :: a -> a -> [a]      -- [n,n'..]
   enumFromTo     :: a -> a -> [a]      -- [n..m]
   enumFromThenTo :: a -> a -> a -> [a]  -- [n,n'..m]
   {-# MINIMAL toEnum, fromEnum #-}
```

Instancją klasy Enum jest typ Char i większość typów numerycznych.

```
*Main> ['a'..'d']

"abcd "                        --   type String = [Char]
```

# *Klasa Enum, cd.*

Typy wyliczeniowe, zdefiniowane przez użytkownika, też mogą być instancją klasy Enum.

```
data Kolor = Trefl | Karo | Kier | Pik
                deriving (Eq, Ord, Show, Enum)


*Main> [Trefl ..]
[Trefl,Karo,Kier,Pik]
*Main> fromEnum Kier
2
*Main> toEnum 3::Kolor          -- trzeba sprecyzować typ wyniku
Pik
*Main> succ Karo
Kier
```

# *Klasa Bounded*

Użyteczną klasą jest też klasa Bounded (dla typów z dolną i górną granicą wartości).

```
Prelude> :i Bounded
class Bounded a where
  minBound :: a
  maxBound :: a
  {-# MINIMAL minBound, maxBound #-}
…
Prelude> (minBound, maxBound) :: (Int, Int)
(-9223372036854775808,9223372036854775807)

Prelude> data Kolor = Trefl | Karo | Kier | Pik  deriving (Eq, Ord, Show, Enum, Bounded)
Prelude> maxBound :: Kolor
Pik
```

# *Klasy numeryczne*

Typy numeryczne w Haskellu są instancjami różnych klas, tworzących hierarchię, pokazaną dalej. Na szczycie tej hierarchii znajduje się klasa `Num`.

```
class  (Eq a, Show a) => Num a  where
          (+), (-), (*)  :: a -> a -> a
          negate         :: a -> a
        abs, signum      :: a -> a
          fromInteger    :: Integer -> a
```

negate jest funkcją negacji, jako synonimu można użyć operatora -. Nie jest on częścią literału numerycznego, dlatego często trzeba używać nawiasów w celu wymuszenia odpowiedniej kolejności aplikacji funkcji. Np.

abs -8 ≡ abs negate 8 ≡ (abs negate) 8  <= błąd typu

Poprawnie:

abs(-8) lub abs(negate 8) lub abs $ -8  itp.

# *Klasy numeryczne cd.*

```
class  (Eq a, Show a) => Num a  where
          (+), (-), (⋆)  :: a -> a -> a
           negate        :: a -> a
        abs, signum      :: a -> a
         fromInteger     :: Integer -> a
```

signum sprawdza znak argumentu i zwraca -1 dla argumentów ujemnych, 0 dla 0 i 1 dla argumentów dodatnich.

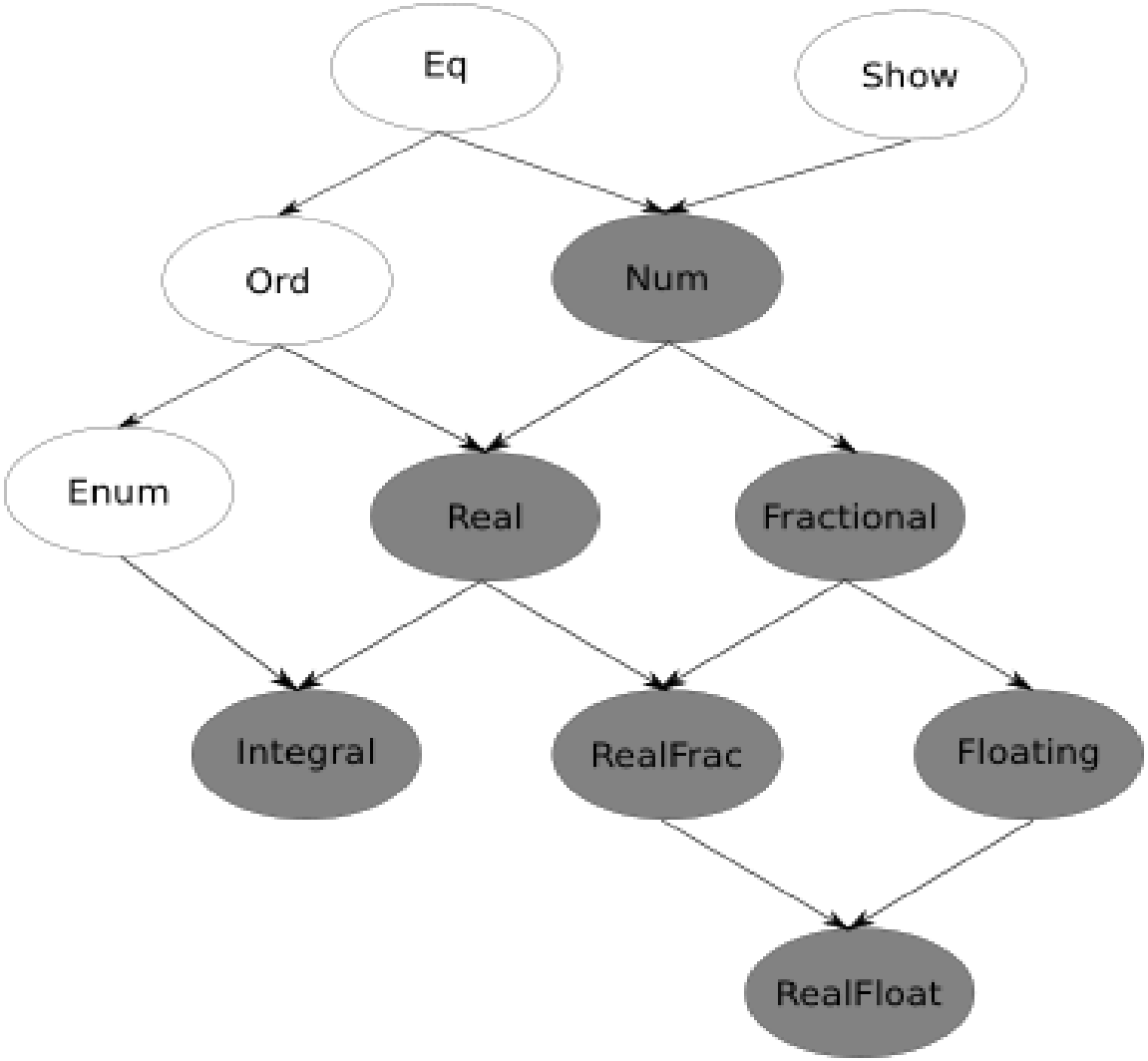
fromInteger jest funkcją koercji.

```
*Main> :t fromInteger

fromInteger :: Num a => Integer -> a
```

Każdy literał całkowitoliczbowy, np. „55” jest w rzeczywistości skrótem dla „ fromInteger 55”.

```
*Main> :t 55

55 :: Num a => a
```

# *Hierarchia klas numerycznych Haskella*



# *Wybrane typy numeryczne*

Double – liczba zmiennoprzecinkowa podwójnej precyzji

Float – liczba zmiennoprzecinkowa pojedynczej precyzji

Int – liczba całkowita ze znakiem o ustalonym rozmiarze

Integer – liczba całkowita ze znakiem dowolnej wielkości

Word – liczba całkowita bez znaku o tym samym rozmiarze, co Int

Po zaimportowaniu modułu Data.Ratio dostępny jest też typ:

Rational – liczba ułamkowa dowolnej wielkości (pamiętana jako dwie wartości typu Integer, np. 2%3)

```
Prelude> import Data.Ratio
Prelude Data.Ratio> 4%6
2 % 3
Prelude Data.Ratio> 1%2 + 1%3
5 % 6
```

# *Typy wyższego rzędu*

W Haskellu na wszystkich poziomach (aplikacja funkcji, konstruktory wartości, konstruktory typów) konsekwentnie stosowana jest notacja prefiksowa. Na wszystkich poziomach można też używać postaci rozwiniętej. Występująca również notacja infiksowa i miksfiksowa to tylko lukier syntaktyczny.

Funkcje:

```
*Main> :t (+)
(+) :: Num a => a -> a -> a
*Main> :t (+) 5
(+) 5 :: Num a => a -> a
```

Konstruktory wartości:

```
*Main> :t Branch
Branch :: Tree a -> Tree a -> Tree a
*Main> :t Branch (Leaf 5)
Branch (Leaf 5) :: Num a => Tree a -> Tree a
```

# *Typy wyższego rzędu cd.*

Konstruktory wartości, notacja miksfiksowa:

```
*Main> (2,Trefl)
(2,Trefl)
*Main> :t (,)
(,) :: a -> b -> (a, b)
*Main> :t (,) 2
(,) 2 :: Num a => b -> (a, b)
*Main> :t (,,)
(,,) :: a -> b -> c -> (a, b, c)
```

itd. dla wszystkich krotek.

Konstruktory wartości, notacja infiksowa:

```
*Main> 0:[1,2]
[0,1,2]
*Main> :t (:)
(:) :: a -> [a] -> [a]
*Main> :t (:[1,2])  -- por. wykład 3, str.8
(:[1,2]) :: Num a => a -> [a]
```

# *Typy wyższego rzędu cd.*

Konstruktory typów, notacja miksfiksowa:

```
-- toPair :: a -> b -> (a,b)
toPair :: a -> b -> (,) a b
toPair a b = (a,b)   -- toPair a b = (,) a b


-- toTriple :: a -> b -> c -> (a,b,c)
toTriple :: a -> b -> c -> (,,) a b c
toTriple a b c = (a,b,c)     -- toTriple a b c = (,,) a b c


-- toList :: a -> [a]
toList :: a -> [] a
toList a = [a]   -- toList a = (:) a []
```

# *Typy wyższego rzędu cd.*

Konstruktory typów, notacja infiksowa:

```
-- apply :: ( a -> b) -> a -> b
-- apply :: (->) a b -> a -> b
-- apply :: (->) ((->) a b) (a -> b)
apply :: (->) ((->) a b) ((->) a b)
apply f a = f a
```

W komentarzach powyżej pokazano – krok po kroku – zamianę notacji infiksowej na prefiksową.

# *Typy wyższego rzędu cd.*

Można zadać sobie pytanie: jaki jest „typ” typu `Int`,
czy konstruktorów typów `[]`, `(->)`, `(,)`, `(,,)` itd.?

„Typ” typu jest nazywany *gatunkiem* (ang. kind) i jest oznaczany symbolem *.
Reprezentuje on rodzaj dla typu, który może być przypisany konkretnej wartości. Np. gatunkiem typu `Int`, `Kolor` czy `Tree Int` jest *. A jaki jest gatunek konstruktora typu `Tree`, czy wyżej wymienionych konstruktorów typów? Potrzebna jest definicja indukcyjna.

1. Symbol * jest gatunkiem.
2. Jeśli $\kappa_1$ i $\kappa_2$ są gatunkami, to $\kappa_1 \rightarrow \kappa_2$ jest gatunkiem (konstruktora) typu, który bierze jako argument typ gatunku $\kappa_1$ i zwraca typ gatunku $\kappa_2$.

Konstruktory typów `Tree`, `[]` są gatunku * → *, `(,)` i `(->)` są gatunku * → * → *,
`(,,)` jest gatunku * → * → * → *, itd. Typ `(,)Int` jest gatunku * → *.
Typ `(,)Int Kolor` jest gatunku *. Aplikacja typów ma łączność lewostronną, tak jak aplikacja funkcji.

Zwykle gatunki nie występują bezpośrednio w programach Haskella, są jednak wykorzystywane przez system typów Haskella. Ułatwiają też zrozumienie niektórych klas typów i komunikatów o błędach gatunku.

# *Typy wyższego rzędu cd.*

Jak widzieliśmy już w materiałach do wykładu 1, a także do wykładu bieżącego, w środowisku interakcyjnym GHCi Haskella istnieje komenda :type (w skrócie :t), pokazująca typ dowolnego wyrażenia. Analogicznie za pomocą komendy :kind (w skrócie :k), można sprawdzić gatunek dowolnego typu. W Haskellu dla wielu typów (np. dla list) dla konstruktora typu i konstruktora wartości jest używana ta sama notacja ( w przypadku list []). Są one rozróżniane na podstawie kontekstu użycia.

```
Prelude> :kind Int
Int :: *
Prelude> :k []
[] :: * -> *
Prelude> :t []
[] :: [a]
Prelude> :k (,)
(,) :: * -> * -> *
Prelude> :k (,) Char
(,) Char :: * -> *
Prelude> :t (,)
(,) :: a -> b -> (a, b)
Prelude> :t (,) 'x'
(,) 'x' :: b -> (Char, b)
Prelude> :k (,,)
(,,) :: * -> * -> * -> *
Prelude> :k Tree
Tree :: * -> *
```