Функційне програмування мовою Haskell

Базові типи і класи

Сорти (kind) типів

Щоб переконатися, що вирази типів дійсні, їх класифікують за видами (сортами, родами, kind), які приймають одну з двох можливих форм:

- Символ $*$ представляє вид усіх нульарних конструкторів типу.
- • Якщо k_1 і k_2 є видами, тоді $k_1 \to k_2$ є видом типів, які приймають тип виду k_1 і повертають тип виду k_2.

Виведення видів (сортів) типу перевіряє дійсність виразів типу подібно до того, як виведення типу перевіряє дійсність виразів значення. Однак, на відміну від типів, види повністю неявні і не є видимою частиною мови.

Сорти (kind) типів

Щоб переконатися, що вирази типів дійсні, їх класифікують за видами (сортами, родами, kind), які приймають одну з двох можливих форм:

- Символ $*$ представляє вид усіх нульарних конструкторів типу.
- • Якщо k_1 і k_2 є видами, тоді $k_1 \rightarrow k_2$ є видом типів, які приймають тип виду k_1 і повертають тип виду k_2.

ghci> :kind Int

ghci> :k Char

ghci> :k Either

Тип **Bool**

```
data Bool = False
          | True
          deriving (Read, Show, Eq, Ord, Enum, Bounded)
```

- опис
- kind
- функції
- втілення (екземпляр)
- інше (..., otherwise, …)

Тип **Maybe**

```haskell
data  Maybe a  = Nothing
               | Just a
               deriving (Eq, Ord)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

Тип **Maybe**

```
data  Maybe a  = Nothing
               | Just a
               deriving (Eq, Ord)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing = n
maybe _ f (Just x) = f x
```

Тип **Maybe**

```
data  Maybe a  = Nothing
               | Just a
               deriving (Eq, Ord)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

```
head' :: [a]-> Maybe a
head' []     = Nothing
head' (x:_) = Just x
```

Тип **Maybe**

```
data  Maybe a  = Nothing
               | Just a
               deriving (Eq, Ord)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

```
head' :: [a]-> Maybe a
head' []     = Nothing
head' (x:_) = Just x
```

```
*Main> maybe 0 (*1) (head' [3..7])
3
*Main> maybe 0 (*12) (head' [3..7])
36
*Main> maybe 0 (*12) (head' [])
0
*Main> maybe "" (show) (head' [])
""
*Main> maybe "" (show) (head' [3..7])
"3"
```

Тип **Either**

```haskell
data  Either a b = Left  a
                 | Right b
                 deriving (Eq, Ord, Read, Show)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

Тип **Either**

```haskell
data  Either a b = Left  a
                 | Right b
                 deriving (Eq, Ord, Read, Show)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

```haskell
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)    =  f x
either _ g (Right y)   =  g y
```

Тип **Either**

```haskell
data  Either a b = Left  a
                 | Right b
                 deriving (Eq, Ord, Read, Show)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

```
head'' :: [a] -> Either String a
head'' [] = Left "this is empty list"
head'' (x:_) = Right x
```

Тип **Either**

```haskell
data  Either a b = Left  a
                 | Right b
                 deriving (Eq, Ord, Read, Show)
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

```haskell
head'' :: [a] -> Either String a
head'' [] = Left "this is empty list"
head'' (x:_) = Right x
```

```
*Main> head'' [3..7]
Right 3
it :: (Enum a, Num a) => Either String a

*Main> head'' []
Left "this is empty list"
it :: Either String a
```

Тип **Ordering**

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

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

Тип ***списки (lists)***

```haskell
data [a] = []
         | a : [a]
         deriving (Eq, Ord)
         -- Not legal Haskell; for illustration only
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

Тип ***списки (lists)***

```haskell
data [a] = []
         | a : [a]
         deriving (Eq, Ord)
         -- Not legal Haskell; for illustration only
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

Тип ***списки (lists)***

```
data [a] = []
         | a : [a]
         deriving (Eq, Ord)
         -- Not legal Haskell; for illustration only
```

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

стандартні функції

map, (++), filter, concat, concatMap, head, last, tail, init, null, length, (!!), foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1, iterate, repeat, replicate, cycle, take, drop, splitAt, takeWhile, dropWhile, span, break, lines, words, unlines, unwords, reverse, and, or, any, all, elem, notElem, lookup, sum, product, maximum, minimum, zip, zip3, zipWith, zipWith3, unzip, unzip3

Тип ***кортежі (tuples)***

Кортежі - алгебраїчні типи даних із спеціальним синтаксисом. Кожен тип кортежу має єдиний конструктор.

Конструктори, наприклад
(,) 1 “fgh” ⇔ (1,”fgh”)

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

Тип ***кортежі (tuples)***

Кортежі - алгебраїчні типи даних із спеціальним синтаксисом. Кожен тип кортежу має єдиний конструктор.

Конструктори, наприклад
(,) 1 “fgh” ⇔ (1,”fgh”)

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

fst, snd, curry, uncurry для 2-кортежів

Тип ***кортежі (tuples)***

Кортежі - алгебраїчні типи даних із спеціальним синтаксисом. Кожен тип кортежу має єдиний конструктор.

Конструктори, наприклад
(,) 1 “fgh” ⇔ (1,”fgh”)

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

```
class Eq a where
…
Instances
…
```

```
⊞ (Eq a, Eq b) => Eq (a, b)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f) => Eq (a, b, c, d, e, f)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g) => Eq (a, b, c, d, e, f, g)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h) => Eq (a, b, c, d, e, f, g, h)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i) => Eq (a, b, c, d, e, f, g, h, i)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j) => Eq (a, b, c, d, e, f, g, h, i, j)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j, Eq k) => Eq (a, b, c, d, e, f, g, h, i, j, k)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j, Eq k, Eq l) => Eq (a, b, c, d, e, f, g, h, i, j, k, l)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j, Eq k, Eq l, Eq m) => Eq (a, b, c, d, e, f, g, h, i, j, k, l, m)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j, Eq k, Eq l, Eq m, Eq n) => Eq (a, b, c, d, e, f, g, h, i, j, k, l, m, n)
⊞ (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j, Eq k, Eq l, Eq m, Eq n, Eq o) => Eq (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o)
```

Тип ***Unit Datatype***

```
data () = ()
    deriving (Eq, Ord, Bounded, Enum, Read, Show)
```

The unit datatype () has one non-⊥ member, the nullary constructor ()

- опис
- kind
- функції
- втілення (екземпляр)
- (..., приклади, …)

Функційний тип

Функції - це абстрактний тип: Ніякі конструктори безпосередньо не створюють функціональні значення.
Наступні прості функції знаходяться в прелюдії: id, const, (.), Flip, ($) і до.

```
id :: a -> a  Source#
```

Identity function.

Функційний тип

Функції - це абстрактний тип: Ніякі конструктори безпосередньо не створюють функціональні значення.
Наступні прості функції знаходяться в прелюдії: id, const, (.), Flip, ($) і до.

```
id :: a -> a  Source#
```

Identity function.

```
const :: a -> b -> a  Source#
```

const x is a unary function which evaluates to x for all inputs.

Функційний тип

Функції - це абстрактний тип: Ніякі конструктори безпосередньо не створюють функціональні значення.
Наступні прості функції знаходяться в прелюдії: id, const, (.), Flip, ($) і до.

```
id :: a -> a  Source#
```

Identity function.

```
const :: a -> b -> a  Source#
```

const x is a unary function which evaluates to x for all inputs.

```
(.) :: (b -> c) -> (a -> b) -> a -> cinfixr 9  Source#
```

Function composition.

Функційний тип

Функції - це абстрактний тип: Ніякі конструктори безпосередньо не створюють функціональні значення.
Наступні прості функції знаходяться в прелюдії: id, const, (.), Flip, ($) і до.

```
flip :: (a -> b -> c) -> b -> a -> c   Source#
```

flip f takes its (first) two arguments in the reverse order of f.

Функційний тип

Функції - це абстрактний тип: Ніякі конструктори безпосередньо не створюють функціональні значення.
Наступні прості функції знаходяться в прелюдії: id, const, (.), Flip, ($) і до.

```
flip :: (a -> b -> c) -> b -> a -> c  Source#
```

flip f takes its (first) two arguments in the reverse order of f.

```
($) :: (a -> b) -> a -> b infixr 0   Source#
```

f $ x = f x

Application operator. This operator is redundant, since ordinary application (f x) means the same as (f $ x). However, $ has low, right-associative binding precedence, so it sometimes allows parentheses to be omitted; for example:

```
f $ g $ h x = f (g (h x))
```

It is also useful in higher-order situations, such as map ($ 0) xs, or zipWith ($) fs xs

```
Prelude> map ($ 1) [(1+),sin,exp]
[2.0,0.8414709848078965,2.718281828459045]
```

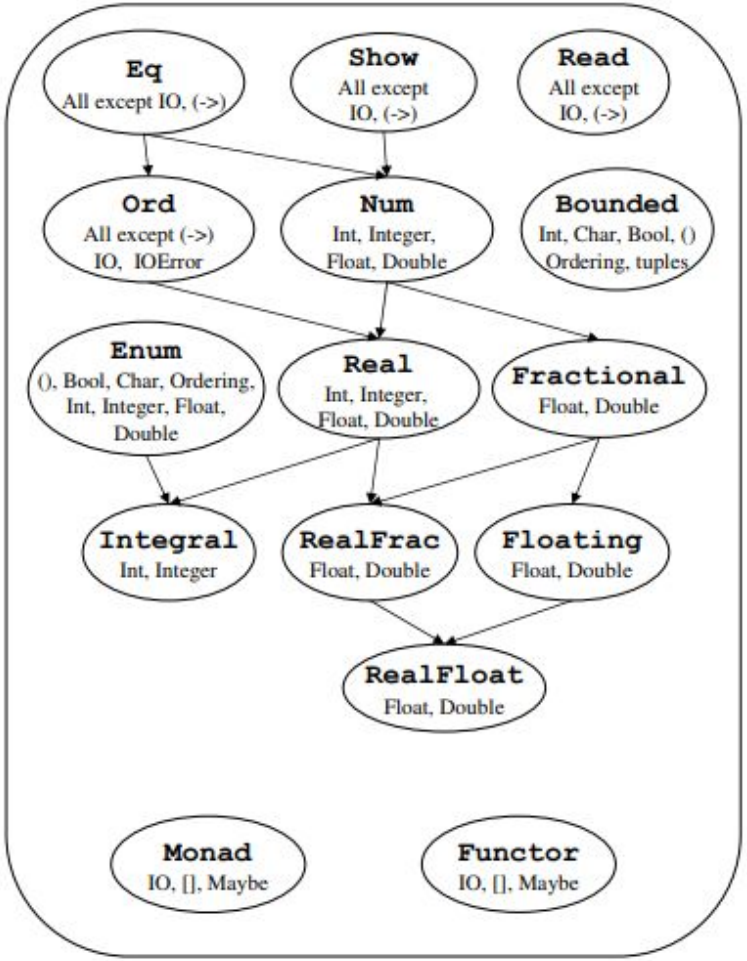

Figure 6.1: Standard Haskell Classes

Клас **Eq**

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


     x /= y  = not (x == y)
     x == y  = not (x /= y)
```

Клас **Eq**

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


    x /= y  = not (x == y)
    x == y  = not (x /= y)

instance (Eq a) => Eq [a] where
    []     == []     = True
    (x:xs) == (y:ys) = x == y && xs == ys
    _xs    == _ys    = False
```

Клас **Eq**

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


      x /= y  = not (x == y)
      x == y  = not (x /= y)
```

```
instance Eq Int where
    (==) = eqInt
    (/=) = neInt

-- See GHC.Classes#matching_overloaded_methods_in_rules
{-# INLINE [1] eqInt #-}
{-# INLINE [1] neInt #-}
eqInt, neInt :: Int -> Int -> Bool
(I# x) `eqInt` (I# y) = isTrue# (x ==# y)
(I# x) `neInt` (I# y) = isTrue# (x /=# y)
```

Клас **Ord**

```
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


  max x y | x <= y    =  y
          | otherwise =  x
  min x y | x <= y    =  x
          | otherwise =  y
```

Клас **Enum**

```
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]
```

Клас **Enum**

```
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]
```

Клас **Bounded**

```
class Bounded a where
      minBound, maxBound :: a
```

Клас **Bounded**

```
class Bounded a where
      minBound, maxBound :: a

Prelude> minBound::(Int,Char,Bool)
(-9223372036854775808,'\NUL',False)
```

Клас **Eq**