PARADIGME DE PROGRAMARE

Curs 17

Clase Haskell standard.

Clase Haskell standard – Cuprins

- Clase pentru tipuri de bază
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Clasa Eq

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

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


Minimal complete definition:
?

Clasa Eq

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

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

Minimal complete definition:
(==) SAU (/=)

Clasa 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
```

Minimal complete definition:
?

```
max x y
  | x >= y    = x
  | otherwise = y

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

Clasa 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
```

Minimal complete definition:
(<=) SAU compare

```
max x y
  | x >= y    = x
  | otherwise = y

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

Clasele Show și Read

```
class Show a where
  show :: a -> String
  ...

class Read a where
  ...
```

Minimal complete definition:
show SAU showsPrec

- Funcția utilitară read, definită în afara clasei:

```
read :: Read a => String -> a
```

Exemplu utilizare:

```
read "[1,2,3]" :: [Int] -- întoarce [1,2,3]
```

Clasa 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]

  succ               =  toEnum . (+1) . fromEnum
  pred               =  toEnum . (subtract 1) . fromEnum
  enumFrom x         =  map toEnum [fromEnum x ..]
  enumFromTo x y     =  map toEnum [fromEnum x .. fromEnum y]
  enumFromThen x y =  map toEnum [fromEnum x, fromEnum y ..]
  enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Minimal complete definition:
?

Clasa 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]

  succ               =  toEnum . (+1) . fromEnum
  pred               =  toEnum . (subtract 1) . fromEnum
  enumFrom x         =  map toEnum [fromEnum x ..]
  enumFromTo x y     =  map toEnum [fromEnum x .. fromEnum y]
  enumFromThen x y =  map toEnum [fromEnum x, fromEnum y ..]
  enumFromThenTo x y z = map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

Minimal complete definition:
toEnum ȘI fromEnum

Clasa Bounded

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

Minimal complete definition:
minBound ȘI maxBound

```
*Main> minBound :: Int
-9223372036854775808
*Main> maxBound :: Int
9223372036854775807
*Main> maxBound :: Integer

<interactive>:149:1:
    No instance for (Bounded Integer) arising from a use of `maxBound'
```

Clasa Num

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

Minimal complete definition:
(+) ȘI (*) ȘI
abs ȘI signum ȘI
fromInteger ȘI
(-) sau negate

```
  x - y = x + negate y
  negate x = 0 - x
```

Cuvântul cheie **deriving**

- Multe clase predefinite sunt derivabile: funcțiile lor pot fi implementate automat (rudimentar) pentru un nou tip care solicită acest lucru folosind **deriving**
  - Eq – a == b dacă aplică aceiași constructori pe aceleași valori în aceeași ordine
  - Ord, Enum – folosesc ordinea în care sunt definiți constructorii de date
    Ex: False < True pentru că data Bool = False | True
  - Show – afișează o valoare ca pe o aplicare succesivă de constructori
  - Read – reconstruiește o valoare dintr-un String care reprezintă o aplicare de constructori
  - Bounded – definește limitele folosind primul și ultimul constructor de date din definiție
  - Num – nu este derivabilă automat, deoarece semnificația operațiilor aritmetice depinde de logica matematică, nu de structura constructorilor

Pentru a instanția Ord trebuie să instanțiem și Eq, nu e automat

```
data Dice = S1 | S2 | S3 | S4 | S5 | S6 deriving (Eq, Ord, Enum)
```

Polimorfism și clase – Cuprins

- Clase pentru tipuri de bază
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Containere

Definim clasa `Container` (care nu există în Haskell) pentru tipuri care conțin elemente

- ex: [a], Maybe a, List a, BSTree a
- clasa oferă funcția `contents` care întoarce o listă cu toate elementele din structură

```
class Container t where
  contents :: t -> ??
```

t = tipul containerului, de exemplu BSTree a

Problema: trebuie să întoarcem [a] și
nu avem un mod de a extrage a din t

Containere

```
class Container t where
  contents :: t -> ??
```

t = tipul containerului, de exemplu BSTree a

Problema: trebuie să întoarcem [a] și
nu avem un mod de a extrage a din t

```
class Container t where
      contents :: t a -> [a]

instance Container [] where
      contents = id
```

Soluția: variabila t să reprezinte constructorul de tip
(ex: BSTree), nu întreg tipul parametrizat

→ Constructorul de tip [ ], nu întreg tipul parametrizat [a]

Clase Haskell pentru containere

Clase Haskell predefinite ale căror operații sunt dedicate containerelor:

- Functor (pentru containere care suportă transformări de tip map)

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

→ Minimal complete definition: fmap

- Foldable (pentru containere reductibile la o valoare prin operații de tip fold)

```
class Foldable t where
  ...
  foldr   :: (a -> b -> b) -> b -> t a -> b
  ...
  foldl   :: (a -> b -> a) -> a -> t b -> a
  ...
```

→ Minimal complete definition: foldr

Exerciții

La ce se evaluează ex1, ex2, ex3, ex4, cunoscând:

instance Functor Maybe -- Defined in `Data.Maybe'

instance Functor [] -- Defined in `GHC.Base'

instance Functor ((->) r) -- Defined in `GHC.Base'

instance Functor ((,) a) -- Defined in `GHC.Base'

instance Foldable ((,) a) -- Defined in `Data.Foldable'

```
ex1 = fmap (+1) (Just 5)
ex2 = fmap (+1) (+1) 2
ex3 = fmap (+1) (1,2)
ex4 = foldl (+) 10 (1,2)
```

Exerciții

La ce se evaluează ex1, ex2, ex3, ex4, cunoscând:

instance Functor Maybe -- Defined in `Data.Maybe'

instance Functor [] -- Defined in `GHC.Base'

instance Functor ((->) r) -- Defined in `GHC.Base'

instance Functor ((,) a) -- Defined in `GHC.Base'

instance Foldable ((,) a) -- Defined in `Data.Foldable'

```
ex1 = fmap (+1) (Just 5)          -- Just 6
ex2 = fmap (+1) (+1) 2            -- 4
ex3 = fmap (+1) (1,2)             -- (1,3)
ex4 = foldl (+) 10 (1,2)          -- 12
```

Polimorfism și clase – Cuprins

- Clase pentru tipuri de bază
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

Clase Haskell versus clase și interfețe POO

Clase Haskell ≠ Clase POO

- O clasă Haskell este o mulțime de tipuri
- O clasă POO este un singur tip (mulțimea valorilor de acel tip)

Clase Haskell ~ Interfețe POO

- Clasa Haskell este instanțiată de diverse tipuri
- Interfața POO este implementată de diverse clase (care sunt ca niște tipuri)
- Ambele doar precizează operațiile pe care tipul trebuie să le aibă, nu le și implementează

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Derivarea unei clase
- Instanțierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

No instance for (<> a)

- signatura furnizată este incompletă: trebuie să-i adăugăm contextul

```
eq :: a -> a -> Bool
eq x y = x == y
```

No instance for (Eq a) arising from a use of `=='

Possible fix:

add (Eq a) to the context of

the type signature for eq :: a -> a -> Bool

In the expression: x == y

No instance for (<> a)

- signatura furnizată este incompletă: trebuie să-i adăugăm contextul

```
eq :: Eq a => a -> a -> Bool
eq x y = x == y
```

No instance for (Eq a) arising from a use of `=='

Possible fix:

add (Eq a) to the context of

the type signature for eq :: a -> a -> Bool

In the expression: x == y

Could not deduce (b ~ a)

- din sinteza de tip rezultă că a = b, dar signaturile furnizate de programator nu garantează acest lucru
- rigid type variable înseamnă că signatura a fost fixată de programator și Haskell nu e liber să unifice a și b

```
eq :: (Eq a, Eq b) => a -> b -> Bool
eq x y = x == y
```

Could not deduce (b ~ a)
from the context (Eq a, Eq b)
...
`b' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool
`a' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool

Could not deduce (b ~ a)

- din sinteza de tip rezultă că a = b, dar signaturile furnizate de programator nu garantează acest lucru
- rigid type variable înseamnă că signatura a fost fixată de programator și Haskell nu e liber să unifice a și b

```
eq :: Eq a => a -> a -> Bool
eq x y = x == y
```

```
Could not deduce (b ~ a)
    from the context (Eq a, Eq b)
...
`b' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool
`a' is a rigid type variable bound by
the type signature for eq :: (Eq a, Eq b) => a -> b -> Bool
```

No instance for (Num <>)

- În Haskell, numerele sunt polimorfice
  - în funcție de context, 1 este Int / Float / Double / ...
- Întregii n sunt înlocuiți automat cu `fromInteger n`
  - fromInteger :: Num a => Integer -> a (transformă în tipul numeric așteptat în context)
- eroarea spune că am folosit un număr pe poziția pe care se aștepta un tip nenumeric

```
f = False || 1
```

No instance for (Num Bool) arising from the literal `1'

Explicație: aștept Bool, înseamnă că fromInteger 1 :: Bool = a,
înseamnă că Num a adică Num Bool (dar asta nu se întâmplă)

No instance for (Num <>)

- În Haskell, numerele sunt polimorfice
  - în funcție de context, 1 este Int / Float / Double / ...
- Întregii n sunt înlocuiți automat cu `fromInteger n`
  - fromInteger :: Num a => Integer -> a (transformă în tipul numeric așteptat în context)
- eroarea spune că am folosit un număr pe poziția pe care se aștepta un tip nenumeric

```
f = False || True
```

No instance for (Num Bool) arising from the literal `1'

Explicație: aștept Bool, înseamnă că fromInteger 1 :: Bool = a,
înseamnă că Num a adică Num Bool (dar asta nu se întâmplă)

Rezumat

Clasa Eq

Clasa Ord

Clasa Show

Clasa Read

Clasa Enum

Clasa Bounded

Clasa Functor

Clasa Foldable

Rezumat

Clasa Eq: pentru tipuri comparabile pentru egalitate (==), (/=)

Clasa Ord

Clasa Show

Clasa Read

Clasa Enum

Clasa Bounded

Clasa Functor

Clasa Foldable

Rezumat

Clasa Eq:	pentru tipuri comparabile pentru egalitate	(==), (/=)
Clasa Ord:	pentru tipuri ordonabile	(<=), compare

Clasa Show

Clasa Read

Clasa Enum

Clasa Bounded

Clasa Functor

Clasa Foldable

Rezumat

Clasa Eq:	pentru tipuri comparabile pentru egalitate	(==), (/=)
Clasa Ord:	pentru tipuri ordonabile	(<=), compare
Clasa Show:	pentru tipuri afișabile	show
Clasa Read		
Clasa Enum		
Clasa Bounded		
Clasa Functor		
Clasa Foldable		

Rezumat

Clasa Eq:	pentru tipuri comparabile pentru egalitate	(==), (/=)
Clasa Ord:	pentru tipuri ordonabile	(<=), compare
Clasa Show:	pentru tipuri afișabile	show
Clasa Read:	pentru tipuri care pot fi citite din String	readsPrec

Clasa Enum

Clasa Bounded

Clasa Functor

Clasa Foldable

Rezumat

Clasa Eq:	pentru tipuri comparabile pentru egalitate	(==), (/=)
Clasa Ord:	pentru tipuri ordonabile	(<=), compare
Clasa Show:	pentru tipuri afișabile	show
Clasa Read:	pentru tipuri care pot fi citite din String	readsPrec
Clasa Enum:	pentru tipuri enumerabile	fromEnum, toEnum
Clasa Bounded		
Clasa Functor		
Clasa Foldable		

Rezumat

Clasa Eq:	pentru tipuri comparabile pentru egalitate	(==), (/=)
Clasa Ord:	pentru tipuri ordonabile	(<=), compare
Clasa Show:	pentru tipuri afișabile	show
Clasa Read:	pentru tipuri care pot fi citite din String	readsPrec
Clasa Enum:	pentru tipuri enumerabile	fromEnum, toEnum
Clasa Bounded:	pentru tipuri limitate inferior/superior	minBound, maxBound

Clasa Functor

Clasa Foldable

Rezumat

Clasa Eq:	pentru tipuri comparabile pentru egalitate	(==), (/=)
Clasa Ord:	pentru tipuri ordonabile	(<=), compare
Clasa Show:	pentru tipuri afișabile	show
Clasa Read:	pentru tipuri care pot fi citite din String	readsPrec
Clasa Enum:	pentru tipuri enumerabile	fromEnum, toEnum
Clasa Bounded:	pentru tipuri limitate inferior/superior	minBound, maxBound
Clasa Functor:	pentru containere transformabile (mapabile)	fmap
Clasa Foldable		

Rezumat

Clasa Eq:	pentru tipuri comparabile pentru egalitate	(==), (/=)
Clasa Ord:	pentru tipuri ordonabile	(<=), compare
Clasa Show:	pentru tipuri afișabile	show
Clasa Read:	pentru tipuri care pot fi citite din String	readsPrec
Clasa Enum:	pentru tipuri enumerabile	fromEnum, toEnum
Clasa Bounded:	pentru tipuri limitate inferior/superior	minBound, maxBound
Clasa Functor:	pentru containere transformabile (mapabile)	fmap
Clasa Foldable:	pentru containere care pot fi reduse	foldr