PARADIGME DE PROGRAMARE

Curs 8

Polimorfism. Clase în Haskell.

1

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

2

2

Polimorfism

- Se referă la furnizarea unei interfețe comune (în cazul nostru, o aceeași funcție) pentru tipuri diferite
- 2 tipuri de polimorfism

Polimorfism parametric = **funcția se comportă la fel pentru argumente de tipuri diferite**
- **Ex:** (:), head, id

Polimorfism ad-hoc = **funcția este supraîncărcată, având comportament diferit în funcție de tipul argumentelor pe care le primește**
- **Ex:** +, *, ==
- `2 + 3 :: Int` întoarce `5`, `2 + 3 :: Double` întoarce `5.0`

Observație: operațiile aritmetice se comportă diferit în funcție de context (la sinteza de tip se deduce că rezultatul operației trebuie să aibă un anumit tip)

3

3

Supraîncărcare

Avantaje

- **Lizbilitate**
  - Mai clar `x == y`, `a == b`, `p == q` decât `eqInt x y`, `eqChar a b`, `eqBool p q`
- **Reutilizare**
  - Mai bine o singură funcție polimorfică myElem (reimplementarea lui elem din Haskell)

```
myElem _ [] = False
myElem a (x:xs) = a == x || myElem a xs
```

decât câte un myElem pentru fiecare tip de date care poate fi căutat într-o listă

`myElemInt` care folosește `eqInt`

`myElemChar` care folosește `eqChar`

`myElemBool` care folosește `eqBool` ...

4

4

Alternative (inferioare) la supraîncărcare

1) Funcții diferite pentru fiecare tip (myElemInt, myElemChar, myElemBool...)
2) Pasarea funcției al cărei comportament diferă ca parametru

```
--myElem2 :: (a -> b -> Bool) -> a -> [b] -> Bool --tipul dedus
--myElem2 :: (a -> a -> Bool) -> a -> [a] -> Bool --tipul dat explicit
myElem2 _ _ [] = False
myElem2 eq a (x:xs) = eq a x || myElem2 eq a xs
```

dar, chiar dacă declarăm noi tipul funcției, acesta este mai general decât ne-am dori, permițând și alte funcții decât cele care testează pentru egalitate

Observație

Haskell nu permite definiții multiple (cu signaturi diferite) pentru un același nume de funcție. O asemenea facilitate ar distruge mecanismul foarte puternic de sinteză de tip.

5

5

Supraîncărcare și sinteză de tip

- Tipul funcției trebuie să **restrângă utilizarea** ei **la tipurile care supraîncarcă o anumită operație**
  - Astfel prevenim și erori rezultate din aplicarea funcției pe tipuri care nu au operația respectivă
  - **Ex:** Este posibil să avem liste de funcții în Haskell:

```
*Main> zipWith (\f x -> f x) [(+1), (2/)] [3..]
[4.0,0.5]
```

dar nu este posibil să căutăm o funcție în ele, întrucât nu există un algoritm pentru a determina dacă 2 funcții sunt egale (au același comportament):

```
*Main> elem (+1) [(+1), (2/)]
...
    No instance for (Eq (a0 -> a0)) arising from a use of `elem'
```

Funcțiile nu pot fi comparate pentru egalitate, dar argumentele lui elem trebuie să poată

6

6

Supraîncărcare și sinteză de tip

Exemple (signaturi pentru funcții polimorfice ad-hoc)

- `:t sum`

  `sum :: (Foldable t, Num a) => t a -> a`

  Tipul a trebuie să supraîncarce operațiile specifice numerelor (+ , - , * ...)

- `:t elem`

  `elem :: (Foldable t, Eq a) => a -> t a -> Bool`

  Tipul a trebuie să fie comparabil pentru egalitate (să supraîncarce ==)

- `:t Data.List.sort`

  `Data.List.sort :: Ord a => [a] -> [a]`

  Tipul a trebuie să fie ordonabil (să supraîncarce <, <=, >, >= ...)

7

7

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

8

8

Clase în Haskell

Clasă Haskell = **mulțime de tipuri care supraîncarcă operațiile specifice clasei**
- mecanismul Haskell de a implementa polimorfismul ad-hoc
- și un mod de a documenta cum se comportă tipurile (ex: Int este Bounded, Integer nu)

Exemple

Show – clasa tipurilor afișabile (prin funcția show = un fel de toString din Java)
- Membri: toate tipurile în afară de IO și funcții

Num – clasa tipurilor numerice
- Membri: Int, Integer, Float, Double

Bounded – clasa tipurilor ale căror valori sunt limitate inferior și superior
- Membri: Int, Char, Bool, tupluri, Ordering

9

9

Definirea unei clase

```
class NumeClasă t where
    f1 :: signatura1
    ...
    fn :: signaturan
```

- t: Variabilă de tip care reprezintă un tip membru al clasei
- signaturile: Signaturile folosesc variabila de tip (pentru că prin definiție descriem o întreagă clasă de tipuri)

Exemplu

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

Semnificația: Pentru ca un tip a să aparțină clasei Eq, trebuie ca el să implementeze funcțiile (==) și (/=) (respectând signaturile date)

10

10

Implementări implicite

- În general, clasa doar definește funcțiile care trebuie supraîncărcate, nu le și implementează (firesc, întrucât implementările diferă de la tip la tip)
- Excepție: implementări implicite (definiții circulare ale funcțiilor, care permit ca un tip membru să redefinească doar o parte din funcții iar restul să se comporte corect)
- Minimal complete definition: un set minimal de funcții ale clasei care trebuie redefinite la instanțiere astfel încât toate funcțiile să se comporte apoi corect pe tipul respectiv

Exemplu

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

Dacă un tip specifică o implementare pentru (==), (/=) se deduce automat din implementarea implicită

Dacă specifică implementarea pentru (/=), atunci (==) este cel care se deduce automat

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

11

11

Implementări implicite

Avantaje

- Efort minim: Nu trebuie să redefinim toate funcțiile
  - Din rațiuni de performanță, uneori le vom redefini pe toate (Ex: poate că tipul are o metodă mai bună de a detecta inegalitatea decât să eșueze în verificarea egalității)
- Ușurință la instanțiere: Există mai multe definiții complete minimale, o putem alege pe cea mai convenabilă
  - Ex: Uneori e mai ușor să definesc (==), alteori e mai ușor să definesc (/=)

12

12

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

Derivarea unei clase

- Așa cum funcțiile depind de apartenența unui tip la o anumită clasă, la fel și clasele pot necesita ca tipul lor membru să aparțină deja altei clase

Derivarea clasei = impunerea condiției ca un tip să fie deja membru al altei clase (clasa părinte) în momentul în care el devine instanță a clasei copil

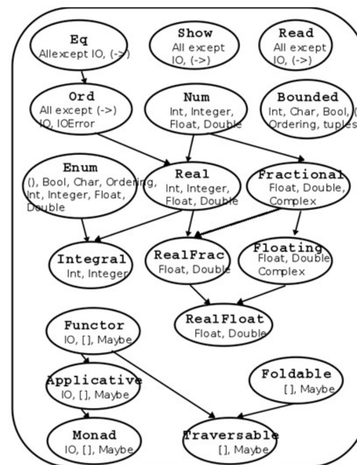
- Clasa copil nu moștenește de la clasa părinte decât promisiunea că instanțele sale vor fi implementat anumite funcții
  - Cele două instanțieri (a clasei părinte și a clasei copil) sunt separate

Exemplu

```
class Eq a => Ord a where
 ... -- alte funcții decât (==), (/=)
```

Semnificația: Pentru ca un tip a să fie membru al clasei Ord, el trebuie să fie deja membru al clasei Eq și să implementeze funcțiile specificate în clasa Ord

Ierarhia de clase în Haskell



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

Instanțierea unei clase

Instanță a unei clase = tip care supraîncarcă toate funcțiile clasei (tip membru)

```
instance NumeClasă Tip where
      f1 = ... -- implementare
      ...
      fn = ... -- implementare
```

Tip concret, nu variabilă de tip ca la definirea clasei

Implementări care respectă signaturile din definiția clasei

Exemplu

```
instance Show Dice where
      show S1 = "[·]"
      ...
      show S6 = "[:::]"
```

17

17

Instanțierea unei clase definită de utilizator

Exemplu

```
class Valuable a where
      value :: a -> Int

instance Valuable Dice where
      value S1 = 1
      …
      value S6 = 6
```

18

18

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

19

19

Context

Context = mulțimea constrângerilor (de apartenență la diverse clase) asupra variabilelor de tip din

- **signatura unei funcții**

```
rollSum :: (Valuable a, Valuable b) => (a, b) -> Int
rollSum (x, y) = value x + value y
```

- **declarația unei clase**

```
class Eq a => Ord a where
```

Se folosește un tuplu pentru constrângeri multiple

- **instanțierea unei clase**

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

20

20

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen lst
      | lst == [] = 0
      | otherwise = 1 + myLen (tail lst)

myLen2 lst
      | null lst = 0
      | otherwise = 1 + myLen2 (tail lst)
```

21

21

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen lst
      | lst == [] = 0
      | otherwise = 1 + myLen (tail lst)
```

- Rezultatul este un număr (0 :: Num a => a , 1 :: Num a => a, (+) :: Num a => a -> a -> a)
- Atât otherwise cât și == întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie comparabil pentru egalitate ((==) :: Eq a => a -> a -> Bool)
- Tipul lui lst trebuie să fie [t] ([] :: [t], (==) :: Eq a => a -> a -> Bool)
- Tipul t trebuie să fie comparabil pentru egalitate (instance Eq a => Eq [a])

22

22

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen lst
      | lst == [] = 0
      | otherwise = 1 + myLen (tail lst)
```

- Rezultatul este un număr (0 :: Num a => a , 1 :: Num a => a, (+) :: Num a => a -> a -> a)
- Atât otherwise cât și == întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie comparabil pentru egalitate ((==) :: Eq a => a -> a -> Bool)
- Tipul lui lst trebuie să fie [t] ([] :: [t], (==) :: Eq a => a -> a -> Bool)
- Tipul t trebuie să fie comparabil pentru egalitate (instance Eq a => Eq [a])

myLen :: (Num a, Eq t) => [t] -> a

23

23

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen2 lst
      | null lst = 0
      | otherwise = 1 + myLen2 (tail lst)
```

- Rezultatul este un număr (0 :: Num a => a , 1 :: Num a => a, (+) :: Num a => a -> a -> a)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)

24

24

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen2 lst
      | null lst = 0
      | otherwise = 1 + myLen2 (tail lst)
```

- Rezultatul este un număr (0 :: Num a => a , 1 :: Num a => a, (+) :: Num a => a -> a -> a)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t] (tail :: [t] -> [t])

25

25

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

```
myLen2 lst
      | null lst = 0
      | otherwise = 1 + myLen2 (tail lst)
```

- Rezultatul este un număr (0 :: Num a => a , 1 :: Num a => a, (+) :: Num a => a -> a -> a)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t] (tail :: [t] -> [t])

myLen2 :: Num a => [t] -> a

*Main> myLen2 [(+)]
1

*Main> myLen [(+)]
eroare

26

26

Simplificarea contextului

- Constrângerile din context se pot aplica **doar pe variabile de tip** (nu pe expresii de tip mai complexe decât atât)
  - `myLen :: (Num a, Eq [t]) => [t] -> a` – eroare fiindcă [t] nu e variabilă de tip
  - `myLen :: (Num a, Eq t) => [t] -> a` – corect
- Constrângerile impun apartenența la clasa copil **fără să impună explicit apartenența la toate clasele părinte** (aceasta se subînțelege)
  - `myMax :: (Eq a, Ord a) => a -> a -> a` – redundant
  - `myMax :: Ord a => a -> a -> a` – corect

27

27

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

28

28

Clasa Ord

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

  compare x y                  max x y
   | x == y =    EQ             | x >= y    = x
   | x <= y =    LT             | otherwise = y
   | otherwise = GT

  x <= y = compare x y /= GT   min x y
  x <  y = compare x y == LT    | x <= y    = x
  x >= y = compare x y /= LT    | otherwise = y
  x >  y = compare x y == GT
```

Minimal complete definition: ?

29

Clasa Ord

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

  compare x y                  max x y
   | x == y =    EQ             | x >= y    = x
   | x <= y =    LT             | otherwise = y
   | otherwise = GT

  x <= y = compare x y /= GT   min x y
  x <  y = compare x y == LT    | x <= y    = x
  x >= y = compare x y /= LT    | otherwise = y
  x >  y = compare x y == GT
```

Minimal complete definition: **(<=) SAU compare**

30

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 ŞI fromEnum**

31

Clasa Bounded

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

```
*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'
```


32

Cuvântul cheie `deriving`

- Multe din clasele predefinite sunt derivabile, adică funcțiile lor pot fi implementate automat (rudimentar) pentru un nou tip care solicită asta folosind `deriving`
- Show – afișează o valoare ca pe o aplicare succesivă de constructori
  Eq – două valori sunt egale dacă se obțin prin aplicarea acelorași constructori pe aceleași valori în aceeași ordine
  Ord și Enum – folosesc ordinea în care sunt definiți constructorii de date
  Ex: False < True pentru că data Bool = False | True

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

Exemplu (cu verificări la calculator)

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

33

33

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

34

34

Containere

- Să presupunem că vrem să definim clasa `Container` (care nu există în Haskell) pentru tipuri (existente sau definite de noi) care conțin elemente (ex: [a], Maybe a, List a, BSTree a) și că această clasă 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 a nu este accesibil în acest punct

35

35

Containere

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

t = tipul containerului, de exemplu BSTree a

Problema: trebuie să întoarcem [a] și a nu este accesibil în acest punct

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

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

```
instance Container [] where
     contents = id
```

Constructorul de tip ([ ]), nu întreg tipul parametrizat ([a])

Exercițiu la calculator: instanțierea pentru List a

36

36

Clase Haskell pentru containere

- Clasa Container nu există în Haskell, dar există 2 clase ale căror operații sunt dedicate containerelor
- **Functor** (pentru **abstractizarea operațiilor de tip map**)

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

- **Foldable** (pentru **abstractizarea operațiilor de tip fold**)

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


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

- 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

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

41

41

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

42

42

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

43

43

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

44

44

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

45

45

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

46

46

No instance for (Num <>)

- numerele în Haskell sunt polimorfice (în funcție de context, 1 este Int / Float / Double / ...)
- în orice context în care apare un întreg (ex: `1`), acesta este înlocuit cu `fromInteger 1` (care îl transformă în tipul numeric așteptat)
- fromInteger :: Num a => Integer -> a
- 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)

47

47

No instance for (Num <>)

- numerele în Haskell sunt polimorfice (în funcție de context, 1 este Int / Float / Double / ...)
- în orice context în care apare un întreg (ex: `1`), acesta este înlocuit cu `fromInteger 1` (care îl transformă în tipul numeric așteptat)
- fromInteger :: Num a => Integer -> a
- 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)

48

48

Rezumat

Polimorfism parametric
Polimorfism ad-hoc
Clasă
Definire clasă
Derivare clasă
Instanțiere clasă
Context
Clase uzuale
Clase pentru containere

49

49

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite
Polimorfism ad-hoc
Clasă
Definire clasă
Derivare clasă
Instanțiere clasă
Context
Clase uzuale
Clase pentru containere

50

50

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite
Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite
Clasă
Definire clasă
Derivare clasă
Instanțiere clasă
Context
Clase uzuale
Clase pentru containere

51

51

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite
Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite
Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei
Definire clasă
Derivare clasă
Instanțiere clasă
Context
Clase uzuale
Clase pentru containere

52

52

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

53

53

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

54

54

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context

Clase uzuale

Clase pentru containere

55

55

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context: mulțimea constrângerilor de apartenență a variabilelor de tip la diverse clase

Clase uzuale

Clase pentru containere

56

56

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context: mulțimea constrângerilor de apartenență a variabilelor de tip la diverse clase

Clase uzuale: Eq, Ord, Read, Show, Enum, Bounded

Clase pentru containere

57

57

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc: funcție cu comportament diferit pentru tipuri diferite

Clasă: mulțime de tipuri care supraîncarcă operațiile specifice clasei

Definire clasă: class <Clasă> t where <declarații de tip>

Derivare clasă: class <Clasă'> t => Clasă t where <declarații de tip>

Instanțiere clasă: instance <Clasă> <Tip> where <implementări>

Context: mulțimea constrângerilor de apartenență a variabilelor de tip la diverse clase

Clase uzuale: Eq, Ord, Read, Show, Enum, Bounded

Clase pentru containere: Functor, Foldable

58

58