# Paradigme de Programare

S.l. dr. ing. Andrei Olaru
slides: Mihnea Muraru si Andrei Olaru

Catedra de Calculatoare

2013 – 2013, semestrul 2

# Cursul 8

# Clase în Haskell

# Cuprins

# Motivație

# Motivaţie

## Exemplu

**Exemplul 29.1.**

Să se definească operaţia `show`, capabilă să producă reprezentarea oricărui obiect ca şir de caractere. Comportamentul este specific fiecărui tip.

```
1  show 3 → "3"
2  show True → "True"
3  show 'a' → "'a'"
4  show "a" → "\"a\""
```

# Motivație

## Varianta 1 – Funcţii dedicate fiecărui tip

```
1  show4Bool True  =   "True"
2  show4Bool False =   "False"
3
4  show4Char c     =   "'" ++ [c] ++ "'"
5
6  show4String s   =   "\"" ++ s ++ "\""
```

# Motivație

## Varianta 1 – Funcţii dedicate – discuție

- Funcţia `showNewLine`, care adaugă caracterul “linie nouă” la reprezentarea ca şir:

```
1  showNewLine x = (show.?. x) ++ "\n"
```

- `showNewLine` nu poate fi polimorfică $\Rightarrow$ `showNewLine4Bool`, `showNewLine4Char` etc.
- Alternativ, trimiterea ca parametru a funcţiei `show*` corespunzătoare:

```
1  showNewLine sh x = (sh x) ++ "\n"
2  showNewLine4Bool = showNewLine show4Bool
```

- Prea general, fiind posibilă trimiterea unei funcţii cu alt comportament, în măsura în care respectă tipul.

# Motivație

## Varianta 2 – Supraîncărcarea funcţiei

- Definirea mulţimii `Show`, a tipurilor care expun `show`

```
1  class Show a where
2      show :: a -> String
3      ...
```

- Precizarea apartenenței unui tip la această mulţime (instanța *aderă* la clasă)

```
1  instance Show Bool where
2      show True  = "True"
3      show False = "False"
4
5  instance Show Char where
6      show c = "'" ++ [c] ++ "'"
```

- Funcţia `showNewLine` polimorfică!

```
1  showNewLine x = (show x) ++ "\n"
```

# Motivație

## Varianta 2 – Supraîncărcare – discuție (1)

- Ce tip au funcţiile `show`, respectiv `showNewLine`?

```
1  show        :: Show a => a -> String
2  showNewLine :: Show a => a -> String
```

  Semnificație: *Dacă tipul* `a` *este membru al clasei* `Show`, *i.e. funcţia* `show` *este definită pe valorile tipului* `a`, *atunci funcţiile au tipul* `a -> String`.

- Context: constrângeri suplimentare asupra variabilelor din tipul funcţiei – `Show a`.

- Propagarea constrângerilor din contextul lui `show` către contextul lui `showNewLine`.

# Motivație

## Varianta 2 – Supraîncărcare – discuție (2)

- Contexte utilizabile şi la instanţiere:

```
1  instance (Show a, Show b) => Show (a, b) where
2      show (x, y) =  "("  ++ (show x)
3                  ++ ",␣" ++ (show y)
4                  ++ ")"
```

- Tipul pereche reprezentabil ca şir doar dacă tipurile celor doi membri respectă aceeaşi proprietate

# Clase Haskell

# Clase şi instanţe

## Definiții

**Definiția 30.1 (Clasă).**

Mulţime de tipuri ce pot supraîncarca operaţiile specifice clasei. Reprezintă o modalitate structurată de control asupra polimorfismului ad-hoc. Exemplu: clasa `Show`, cu operaţia `show`.

**Definiția 30.2 (Instanţă a unei clase).**

Tip care supraîncarcă operaţiile clasei. Exemplu: tipul `Bool` în raport cu clasa `Show`.

# Clase predefinite

## Show, Eq

```
1  class Show a where
2      show :: a -> String
3      ...
4
5  class Eq a where
6      (==), (/=) :: a -> a -> Bool
7      x /= y     =  not (x == y)
8      x == y     =  not (x /= y)
```

- Posibilitatea scrierii de definiţii implicite (v. liniile 7–8).
- Necesitatea suprascrierii cel puţin unuia din cei 2 operatori ai clasei `Eq` pentru instanţierea corectă.

# Clase predefinite

## `Ord`

```
1  class Eq a => Ord a where
2      (<), (<=), (>=), (>) :: a -> a -> Bool
3      ...
```

- Contexte utilzabile şi la definirea unei clase.
- Moştenirea claselor, cu preluarea operaţiilor din clasa moştenită.
- Necesitatea aderării la clasa `Eq` în momentul instanţierii clasei `Ord`.

# Clase Haskell vs. Clase în POO

## Haskell

- Clasele sunt mulţimi de tipuri (superclase)
- Instanţierea claselor de către tipuri

## POO (e.g. Java)

- Clasele sunt mulţimi de obiecte (tipuri)
- Implementarea interfeţelor de către clase

# Aplicaţii clase

**Exemplul 31.1 (`invert`).**

Fie constructorii de tip:

```
1  data Pair a = P a a
2
3  data NestedList a
4      =   Atom a
5      |   Seq [NestedList a]
```

Să se definească operaţia `invert`, aplicabilă pe valori de tipuri diferite, inclusiv `Pair a` şi `NestedList a`, comportamentul fiind specific fiecărui tip.

# invert
## Implementare

```
1  class Invert a where
2      invert  ::  a -> a
3      invert  =   id
4
5  instance Invert (Pair a) where
6      invert (P x y) = P y x
7
8  instance Invert a => Invert (NestedList a) where
9      invert (Atom x) =   Atom (invert x)
10     invert (Seq x)  =   Seq $ reverse . map invert $ x
11
12 instance Invert a => Invert [a] where
13     invert lst = reverse . map invert $ lst
```

- Necesitatea contextului, în cazul tipurilor `[a]` şi `NestedList a`, pentru inversarea elementelor înselor.

## Problema

**Exemplul 31.2 (`contents`).**

Să se definească operaţia `contents`, aplicabilă pe obiecte structurate, inclusiv pe cele aparţinând tipurilor `Pair a` şi `NestedList a`, care întoarce elementele din componenţă, sub forma unei liste Haskell.

```
1  class Container a where
2        contents :: a -> [.?.]
```

- `a` este tipul unui container, e.g. `NestedList b`
- Elementele listei întoarse sunt cele din container
- Cum precizăm tipul acestora (`b`)?

# contents
## Varianta 1a

```
1  class Container a where
2      contents :: a -> [a]
3
4  instance Container [a] where
5      contents = id
```

- Conform definiţiei clasei:

```
1  contents :: Container [a] => [a] -> [[a]]
```

- Conform supraîncărcării funcţiei (`id`):

```
1  contents :: Container [a] => [a] -> [a]
```

- Ecuaţia `[a] = [[a]]` nu are soluţie $\Rightarrow$ eroare.

# contents
## Varianta 1b

```
1  class Container a where
2      contents :: a -> [b]
3
4  instance Container [a] where
5      contents = id
```

- Conform definiţiei clasei:

```
1  contents :: Container [a] => [a] -> [b]
```

- Conform supraîncărcării funcţiei (`id`):

```
1  contents :: Container [a] => [a] -> [a]
```

- Ecuaţia `[a] = [b]` are soluţie pentru `a = b`, dar tipul `[a] -> [a]` insuficient de general în raport cu `[a] -> [b]` $\Rightarrow$ eroare!

# contents
## Varianta 2

- Soluţie: clasa primeşte constructorul de tip, şi nu tipul container propriu-zis

```
1  class Container t where
2      contents :: t a -> [a]
3
4  instance Container Pair where
5      contents (P x y) = [x, y]
6
7  instance Container NestedList where
8      contents (Atom x)    =    [x]
9      contents (Seq x)     =    concatMap contents x
```

# Contexte

## Câteva exemple

```
1  fun1          ::  Eq a => a -> a -> a -> a
2  fun1 x y z    =   if x == y then x else z
3
4  fun2          ::  (Container a, Invert (a b), Eq (a b))
5                =>  (a b) -> (a b) -> [b]
6  fun2 x y      =   if (invert x) == (invert y)
7                        then contents x
8                        else contents y
9
10 fun3          ::  Invert a => [a] -> [a] -> [a]
11 fun3 x y      =   (invert x) ++ (invert y)
12
13 fun4          ::  Ord a => a -> a -> a -> a
14 fun4 x y z    =   if x == y then z else
15                       if x > y then x else y
```

# Contexte
## Observații

- Simplificarea contextului lui `fun3`, de la `Invert [a]` la `Invert a`.
- Simplificarea contextului lui `fun4`, de la `(Eq a, Ord a)` la `Ord a`, din moment ce clasa `Ord` este derivată din clasa `Eq`.

# Sfârșitul cursului 8

## Ce am învățat

· Clase Haskell, polimorfism ad-hoc, instanțiere de clase, derivare a unei clase, context.