

Paradigme de Programare

Conf. dr. ing. Andrei Olaru

andrei.olaru@cs.pub.ro | cs@andreiolaru.ro
Departamentul de Calculatoare

2018

Cursul 8

Clase în Haskell



- 1 Motiva[Pleaseinsertintopreamble]ie
- 2 Clase Haskell
- 3 Aplicații ale claselor

Motivație



Ex | Exemplu

Să se definească operația `show`, capabilă să producă reprezentarea oricărui obiect ca șir de caractere. Comportamentul este **specific** fiecărui tip (polimorfism **ad-hoc**).

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



```
1 showBool True    = "True"
2 showBool False  = "False"
3
4 showChar c       = "'" ++ [c] ++ "'"
5
6 showString s     = "\" ++ s ++ "\""
```



- Dorim să implementăm funcția `showNewLine`, care adaugă caracterul “linie nouă” la reprezentarea ca șir:

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

- `showNewLine` **nu** poate fi polimorfică \Rightarrow avem nevoie de `showNewLineBool`, `showNewLineChar` etc.

- Dorim să implementăm funcția `showNewLine`, care adaugă caracterul “linie nouă” la reprezentarea ca șir:

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

- `showNewLine` **nu** poate fi polimorfică \Rightarrow avem nevoie de `showNewLineBool`, `showNewLineChar` etc.
- Alternativ, trimiterea ca **parametru** a funcției `show*` corespunzătoare:

```
1 showNewLine sh x = (sh x) ++ "\n"  
2 showNewLineBool = showNewLine showBool
```

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



- Definierea **mulțimii** `Show`, a **tipurilor** care expun `show`

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

- Definirea **mulțimii** `Show`, a **tipurilor** care expun `show`

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

- Precizarea **apartenenței** unui tip la această mulțime (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] ++ "'"
```

- Definirea **mulțimii** Show, a **tipurilor** care expun show

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

- Precizarea **apartenenței** unui tip la această mulțime (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] ++ "'"
```

⇒ Funcția showNewLine **polimorfică!**

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



- Ce **tip** au funcțiile `show`, respectiv `showNewLine`?

- Ce **tip** au funcțiile `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. funcția `show` este definită pe valorile tipului `a`), atunci funcțiile au tipul `a -> String`.*

- **Context**: constrângeri suplimentare asupra variabilelor din tipul funcției: $\underbrace{\text{Show } a \Rightarrow}_{\text{context}}$
- **Propagarea** constrângerilor din contextul lui `show` către contextul lui `showNewLine`.

- Contexte utilizabile și la **instanțiere**:

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

- Tipul *pereche* reprezentabil ca șir doar dacă tipurile celor doi membri respectă **aceeași** proprietate (dată de contextul `Show`).

Clase Haskell



Haskell

- **Tipurile** sunt mulțimi de **valori**;
- **Clasele** sunt mulțimi de **tipuri**; tipurile *aderă* la clase;
- **Instanțierea** claselor de către tipuri pentru ca funcțiile definite în clasă să fie disponibile pentru valorile tipului;
- Operațiile specifice clasei sunt implementate în cadrul declarației de instanțiere.

POO (e.g. Java)

- **Clasele** sunt mulțimi de **obiecte (instanțe)**;
- **Interfețele** sunt mulțimi de **clase**; clasele *implementează* interfețe;
- **Implementarea** interfețelor de către clase pentru ca funcțiile definite în interfață să fie disponibile pentru instanțele clasei;
- Operațiile specifice interfeței sunt implementate în cadrul definiției clasei.



+ **Clasa** – Mulțime de tipuri ce pot supraîncarca operațiile specifice clasei. Reprezintă o modalitate structurată de control asupra polimorfismului **ad-hoc**. Exemplu: clasa `Show`, cu operația `show`.

+ **Instanță a unei clase** – Tip care supraîncarcă operațiile clasei. Exemplu: tipul `Bool` în raport cu clasa `Show`.

- *clasa* definește funcțiile **suportate**;
- *clasa* se definește peste o variabilă care stă pentru **constructorul unui tip**;
- *instanța* definește **implementarea** funcțiilor.



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

- Posibilitatea scrierii de definiții **implicite** (v. liniile 6–7).
- Necesitatea suprascrierii **cel puțin unuia** din cei 2 operatori ai clasei `Eq` pentru instanțierea corectă.



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

- contextele – utilizabile și la **definirea** unei clase.
- clasa `Ord` **moștenește** clasa `Eq`, cu preluarea operațiilor din clasa moștenită.
- este **necesară** aderarea la clasa `Eq` în momentul instanțierii clasei `Ord`.
- este **suficientă** supradefinirea lui `(<=)` la instanțiere.



- **Anumite** tipuri de date (definite folosind `Data`) pot beneficia de implementarea **automată** a anumitor funcționalități, oferite de tipurile predefinite în `Prelude`:
 - `Eq`, `Read`, `Show`, `Ord`, `Enum`, `Ix`, `Bounded`.

```
1 data Alarm = Soft | Loud | Deafening
2     deriving (Eq, Ord, Show)
```

- variabilele de tipul `Alarm` pot fi comparate, testate la egalitate, și afișate.

Aplicații ale claselor



Ex | invert

Fie constructorii de tip:

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

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



invert

Implementare

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

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



Ex | contents

Să se definească operația `contents`, aplicabilă pe obiecte **structurate**, inclusiv pe cele aparținând tipurilor `Pair a` și `NestedList a`, care întoarce elementele din componentă, 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`)?



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

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

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

- Ecuația `[a] = [[a]]` nu are soluție \Rightarrow eroare.



contents

Varianta 1b

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

Testăm pentru `contents [1,2,3]`:

- Conform definiției clasei:

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

- Conform supraîncărcării funcției (`id`):

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

- Ecuația $[a] = [b]$ **are** soluție pentru $a = b$, dar tipul $[a] \rightarrow [a]$ **insuficient** de general (prea specific) în raport cu $[a] \rightarrow [b] \Rightarrow$ **eroare!**



Soluție clasa primește **constructorul** de tip, și nu tipul container propriu-zis (rezultat după aplicarea constructorului) \Rightarrow includem tipul conținut de container în expresia de tip a funcției `contents`:

```
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
10
11 instance Container [] where contents = id
```



```
1
2 fun1 x y z = if x == y then x else z
3
4
5
6 fun2 x y = if (invert x) == (invert y)
7           then contents x
8           else contents y
9
10
11 fun3 x y = (invert x) ++ (invert y)
12
13
14 fun4 x y z = if x == y then z else
15             if x > y then x else y
```

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
5
6 fun2 x y = if (invert x) == (invert y)
7           then contents x
8           else contents y
9
10
11 fun3 x y = (invert x) ++ (invert y)
12
13
14 fun4 x y z = if x == y then z else
15             if x > y then x else y
```



```
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, Invertible (a b),
5 Eq (a b)) => (a b) -> (a b) -> [b]
6 fun2 x y = if (invert x) == (invert y)
7           then contents x
8           else contents y
9
10
11 fun3 x y = (invert x) ++ (invert y)
12
13
14 fun4 x y z = if x == y then z else
15             if x > y then x else y
```



```
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, Invertible (a b),
5 Eq (a b)) => (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 :: Invertible 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
```



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



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