PARADIGME DE PROGRAMARE

Curs 16

Polimorfism. Clase în Haskell.

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Extinderea unei clase
- Instanțierea unei clase
- Context

Polimorfism

- Permite utilizarea unei interfețe comune (aceeași funcție) pentru tipuri de date diferite

Tipuri de polimorfism

- Parametric = funcție generică – se comportă la fel, indiferent de tipul argumentelor

  (:), head, id

- Ad-hoc = funcție supraîncărcată – comportament diferit, în funcție de tipul argumentelor

  +, *, ==

  ex: `2 + 3 :: Int` întoarce `5`, `2 + 3 :: Double` întoarce `5.0`

  (operațiile aritmetice se comportă diferit în funcție de context – tipul rezultatului rezultă din sinteza de tip)

Supraîncărcare

Avantaje

- Lizbilitate
  - Supraîncărcare: `x == y,` `a == b,` `p == q`
  - Alternativă: `eqInt x y,` `eqChar a b,` `eqBool p q`
- Reutilizare (Funcții polimorfice care utilizează operații supraîncărcate)
  - Ex: myElem :: Eq t => t -> [t] -> Bool

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

  - Alternativă:

`myElemInt` care folosește `eqInt`
`myElemChar` care folosește `eqChar`
`myElemBool` care folosește `eqBool` ...

Alte alternative

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 declarat
myElem2 _ _ [] = False
myElem2 eq a (x:xs) = eq a x || myElem2 eq a xs
```

- tipul (declarat sau sintetizat) este prea general, permițând și alte funcții decât cele care testează pentru egalitate

Observație: Haskell nu permite definiții multiple (cu signaturi diferite) pentru același nume de funcție. Această facilitate este incompatibilă cu sinteza de tip.

Supraîncărcare și sinteză de tip

- Tipul funcției restrânge utilizarea ei la tipurile care supraîncarcă o anumită operație

- Exemplu
  - Putem opera cu liste de funcții:

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

  - Nu putem testa dacă două funcții sunt egale:

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

Funcțiile nu admit operația ==

dar `elem` folosește această operație pe argumentele sale

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

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 fie numeric (să supraîncarce +, -, * ...)

- `: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 <, <=, >, >= ...)

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Extinderea unei clase
- Instanțierea unei clase
- Context

Clase în Haskell

Clasă Haskell = **mulțime de tipuri care supraîncarcă operațiile specifice clasei**

- mecanismul Haskell de a implementa polimorfismul ad-hoc
- 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

Definirea unei clase

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

→ t = variabilă de tip care reprezintă un tip membru al clasei

→ Signaturile folosesc variabila de tip
(întrucât, prin definiție, descriem o întreagă clasă de tipuri)

Exemplu

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

→ Tipul a poate fi membru al clasei Eq dacă:

- implementează funcțiile (==) și (/=)
(respectând signaturile date)

Implementări implicite

- Clasa — definește funcțiile care trebuie supraîncărcate
  Tipurile (instanțele clasei) — implementează funcțiile – cu comportament specific tipului

- Implementări implicite: — definiții circulare, nefuncționale în această formă
  - optimizează instanțierea, permițând utilizatorului să redefinească un set minim de funcții

Exemplu

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

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

- Dacă instanța definește (==), (/=) se deduce automat
- Dacă instanța definește (/=), (==) se deduce automat

Implementări implicite

Avantaje

- Efort minim: La instanțiere, nu este necesară definirea tuturor funcțiilor clasei
  - Utilizatorul poate alege redefinirea integrală, din motive de eficiență
- Flexibilitate: Se poate redefini cel mai convenabil set minimal
  - Ex: Uneori e mai ușor să definesc (==), alteori e mai ușor să definesc (/=)

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Extinderea unei clase
- Instanțierea unei clase
- Context

Extinderea unei clase

Extinderea clasei = definirea clasei copil cu impunerea condiției ca tipul membru să fie deja în clasa părinte în momentul în care devine instanță a clasei copil

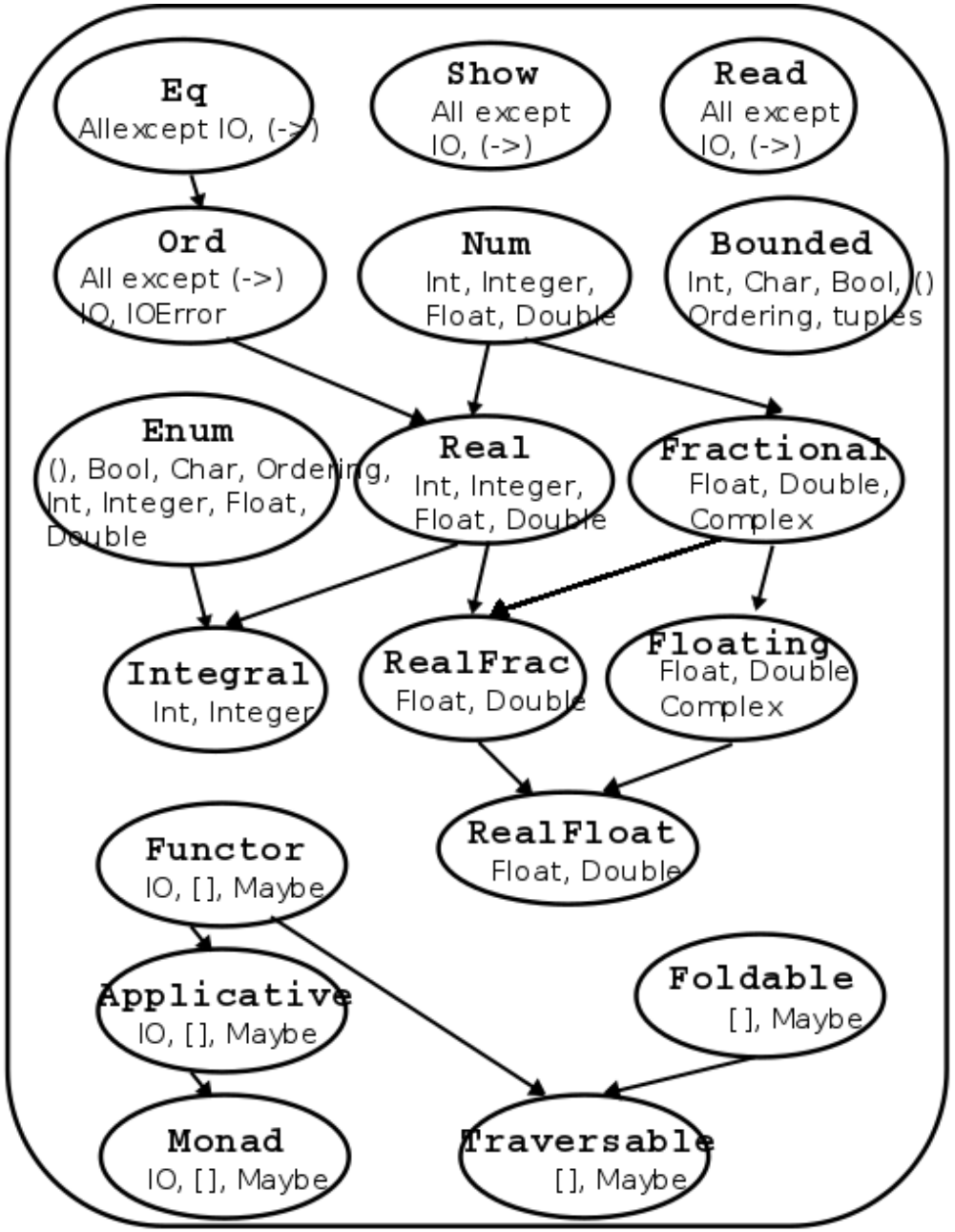
- Constrângere, nu moștenire
  - Nu se prea nimic de la clasa părinte
  - Instanțele clasei copil trebuie sa demonstreze că implementează funcțiile clasei părinte
- Independență
  - Tipurile membre instanțiază separat clasa părinte și clasa copil

Exemplu

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

→ Tipul a poate fi membru al clasei Ord dacă:
- este deja membru al clasei Eq
- implementează funcțiile specifice clasei Ord

Ierarhia de clase în Haskell



Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Extinderea unei clase
- Instanțierea unei clase
- Context

Instanțierea unei clase

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

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

→ Tip = 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 = "[:::]"
```

Instanțierea unei clase definite de utilizator

Exemplu

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

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

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- Extinderea unei clase
- Instanțierea unei clase
- Context

Context

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

- signatura unei funcții

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

Se folosește un tuplu pentru constrângeri multiple

- declarația unei clase

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

- instanțierea unei clase

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

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)
```

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 [t]: [] :: [t], `(==)` :: Eq a => a -> a -> Bool, tail :: [a] -> [a]
- Tipul lui lst trebuie să fie comparabil pentru egalitate: `(==)` :: Eq a => a -> a -> Bool
- Tipul t trebuie să fie comparabil pentru egalitate: instance Eq a => Eq [a]

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

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

```
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 [t]: [] :: [t], `(==)` :: Eq a => a -> a -> Bool, tail :: [t] -> [t]
- Tipul lui lst trebuie să fie comparabil pentru egalitate: `(==)` :: Eq a => a -> a -> Bool
- Tipul t trebuie să fie comparabil pentru egalitate: instance Eq a => Eq [a]

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 == întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t]: `tail :: [t] -> [t]`

Propagarea contextului la sinteza de tip

Să se determine tipul următoarelor 2 funcții:

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

```
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 == întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t]: `tail :: [t] -> [t]`

```
*Main> myLen2 [(+)]
1
```

```
*Main> myLen [(+)]
eroare
```

Simplificarea contextului

Reguli de scriere a contextului

- Constrângerile se aplică pe variabile de tip (nu expresii de tip mai complexe)
  - `myLen :: (Num a, Eq [t]) => [t] -> a` – eroare: [t] nu e variabilă de tip
  - `myLen :: (Num a, Eq t) => [t] -> a` – corect
- Apartenența la o clasă copil implică automat apartenența la toate clasele părinte
  - `myMax :: (Eq a, Ord a) => a -> a -> a` – redundant
  - `myMax :: Ord a => a -> a -> a` – optim

Rezumat

Polimorfism parametric

Polimorfism ad-hoc

Clasă

Definire clasă

Extindere clasă

Instanțiere clasă

Context

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

Polimorfism ad-hoc

Clasă

Definire clasă

Extindere clasă

Instanțiere clasă

Context

Rezumat

Polimorfism parametric: funcție cu același comportament pentru tipuri diferite

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

Clasă

Definire clasă

Extindere clasă

Instanțiere clasă

Context

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

Extindere clasă

Instanțiere clasă

Context

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>

Extindere clasă

Instanțiere clasă

Context

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>

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

Instanțiere clasă

Context

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>

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

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

Context

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>

Extindere 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