

PARADIGME DE PROGRAMARE

Curs 8

Tipuri definite de utilizator. Polimorfism. Clase în Haskell.

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Construcția `type`

- Se folosește pentru a crea **sinonime de tip**, în scop de:
 - **Documentare**: este mai clar ce face o variabilă de tip `Age` decât o variabilă de tip `Int`
 - **Concizie**: este mai scurt (și mai clar) `Name` decât `(String, String)`

Exemple

```
type Age = Int
```

```
type Name = (String, String)
```

```
names :: [Name]
```

```
names = [("Frederic", "Chopin"), ("Antonio", "Vivaldi"), ("Maurice", "Ravel")]
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Construcția `newtype`

- Se folosește pentru a crea **tipuri noi** (definite de utilizator) folosind **un singur constructor cu un singur parametru**
- Mai **eficient** decât **data**:
 - Pentru valorile tipurilor definite cu **data** trebuie să se facă pattern match pe constructori și apoi să se acceseze valorile închise în aceștia
 - Pentru valorile tipurilor definite cu **newtype**, existând un singur constructor, acesta este șters încă din faza de compilare, și înlocuit cu valoarea închisă în el (care se știe ce tip are)
- Util pentru a defini apoi operații pe tipul respectiv

Exemplu

```
newtype Person = Person (Name, Age) deriving Show
fc :: Person
fc = Person (("Frederic", "Chopin"), 209)
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Tipuri Enumerate

- Numite și tipuri sumă (| face suma/reuniunea valorilor tipului)
- Enumeră toate valorile tipului, sub forma
`data ConstTip = Val1 | Val2 | ... | Valn`

Exemple

```
data Dice = S1 | S2 | S3 | S4 | S5 | S6
```

```
*Main> :i Bool
```

```
data Bool = False | True           -- Defined in `GHC.Types'
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerate
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Tipuri înregistrare

- Numite și tipuri produs: o valoare a tipului se obține prin combinarea unor valori de alte tipuri, sub forma
`data` ConstTip = Cons {câmp₁ :: tip₁, ... câmp_n :: tip_n}
care este o variantă cu funcții selector pentru definiția
`data` ConstTip = Cons tip₁ ... tip_n
- Au un corespondent în majoritatea limbajelor de programare (ex: struct în C++)

Exemplu

```
data Person2 = Person2 {name :: Name, age :: Age}  
fc2 :: Person2  
fc2 = Person2 ("Frederic", "Chopin") 209  
composer = name fc2
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerare
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Tipuri recursive

- Tipuri pentru care specificăm și cel puțin un constructor intern
`data` ConsTip = .. | Cons_i .. ConsTip .. | ..

Exemple

```
data Natural = Zero | Succ Natural deriving Show
```

```
data IntList = Nil | Cons Int IntList deriving Show
```

Tipuri definite de utilizator – Cuprins

- Construcția type
- Construcția newtype
- Tipuri enumerare
- Tipuri înregistrare
- Tipuri recursive
- Tipuri parametrizate

Tipuri parametrizate

- Constructorul de tip este aplicat pe una sau mai multe variabile de tip, permițând obținerea unor tipuri particulare la instanțiere
`data` `ConsTip a b ... =`
- Nu are rost să avem `IntList`, `CharList`, `PairList`, etc., este de preferat să avem un singur tip parametrizat `List a`, unde `a` se va lega la un tip concret în momentul în care plasăm valori de un tip concret în listă

Exemplu

```
data List a = Nil | Cons a (List a) deriving Show
```

```
lst1 = Cons 1 $ Cons 2.5 $ Cons 4 Nil      -- :t lst1 => lst1 :: List Double
```

```
lst2 = Cons "Hello " $ Cons "world!" Nil -- :t lst2 => lst2 :: List [Char]
```

Polimorfism și clase – Cuprins

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

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:** length, 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)

Supraîncărcare

Avantaje

- **Lizibilitate**

- 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 ...
```


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 semnături diferite) pentru un același nume de funcție. O asemenea facilitate ar distruge mecanismul foarte puternic de sinteză de tip.

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

Supraîncărcare și sinteză de tip

Exemple (signaturi pentru funcții polimorfice ad-hoc)

• `:t sum`

`sum :: Num a => [a] -> a`

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

• `:t elem`

`elem :: Eq a => a -> [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**
- Derivarea unei clase
- Instanțierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparație cu clasele din POO
- Mesaje de eroare

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

Definirea unei clase

```
class NumeClasă t where
```

```
f1 :: signatura1
```

```
...
```

```
fn :: signaturan
```

Variabilă de tip care reprezintă un tip membru al clasei

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)

Implementări implicite

- În general, clasa doar definește funcțiile care trebuie supraîncărcate, nu le și implementează (firească, 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 (/=)

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 (`/=`)

Polimorfism și clase – Cuprins

- Polimorfism parametric și ad-hoc
- Clase Haskell
- **Derivarea unei clase**
- Instanțierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparatie 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

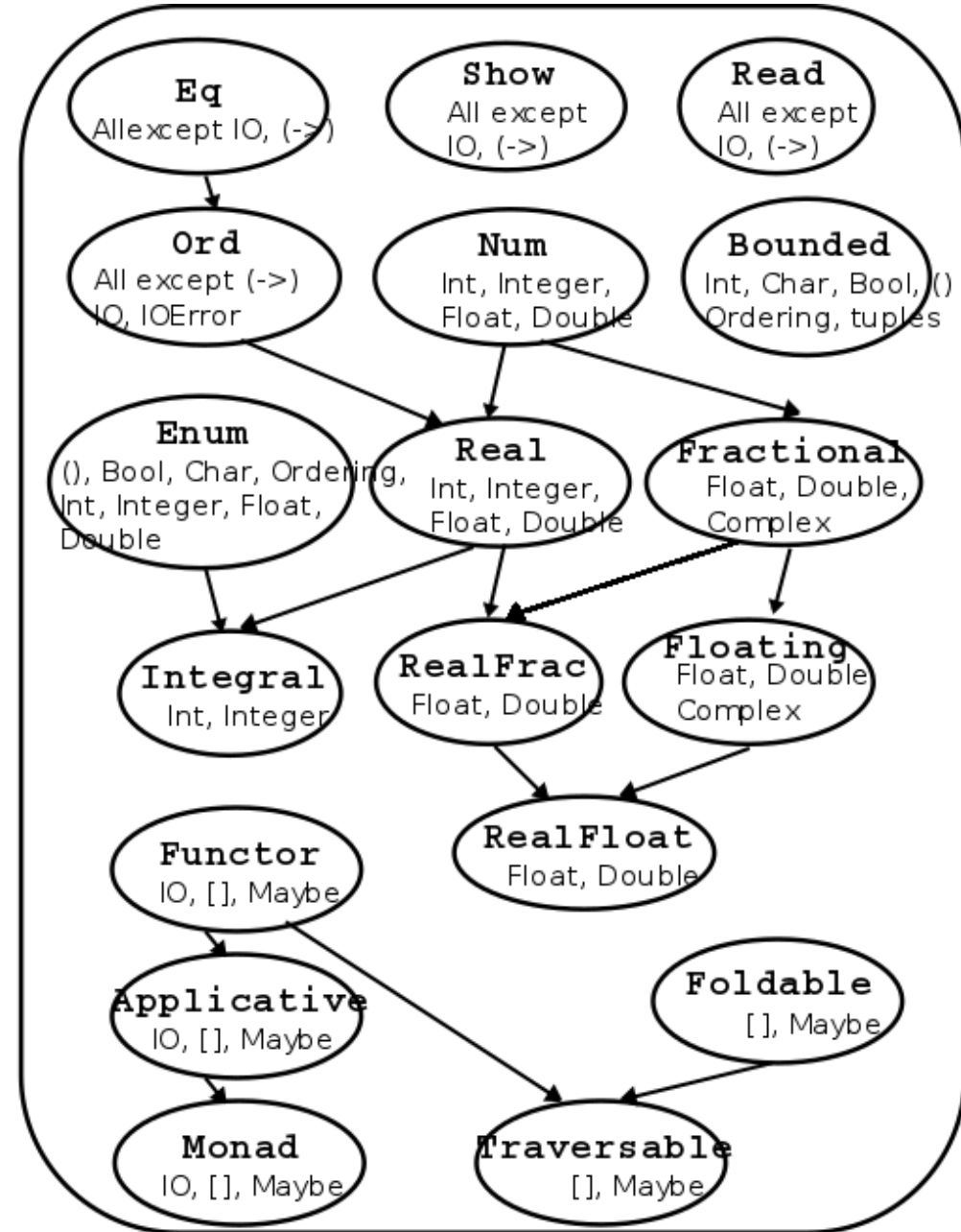
- Clasa copil nu moștenește de la clasa părinte decât promisiunea că instanțele sale vor fi implementat anumite funcții
 - Cele două instanțieri (a clasei părinte și a clasei copil) sunt separate

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
- Comparatie 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 = "::::"
```

Instanțierea unei clase definită 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
- Derivarea unei clase
- Instanțierea unei clase
- Context
- Clase uzuale
- Clase pentru containere
- Comparatie cu clasele din POO
- Mesaje de eroare

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
```

- **instanțierea unei clase**

```
instance Eq a => Eq [a] where
```

```
  [] == [] = True  
  (x:xs) == (y:ys) = x == y && xs == ys  
  _ == _ = False
```

Se folosește un tuplu pentru
constrângeri multiple

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 ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și == întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie comparabil pentru egalitate ($(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul lui lst trebuie să fie [t] ($[] :: [t]$, $(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul t trebuie să fie comparabil pentru egalitate ($\text{instance Eq } a \Rightarrow \text{Eq } [a]$)

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 ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât `otherwise` cât și `==` întorc `Bool` (necesar pentru a folosi gărzi)
- Tipul lui `lst` trebuie să fie comparabil pentru egalitate ($(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul lui `lst` trebuie să fie `[t]` ($[] :: [t]$, $(==) :: \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}$)
- Tipul `t` trebuie să fie comparabil pentru egalitate ($\text{instance Eq } a \Rightarrow \text{Eq } [a]$)

`myLen :: (Num a, Eq t) => [t] -> 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 ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)

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 ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t] ($\text{null} :: [t] \rightarrow \text{Bool}$)

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 ($0 :: \text{Num } a \Rightarrow a$, $1 :: \text{Num } a \Rightarrow a$, $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$)
- Atât otherwise cât și null întorc Bool (necesar pentru a folosi gărzi)
- Tipul lui lst trebuie să fie [t] ($\text{null} :: [t] \rightarrow \text{Bool}$)

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

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

```
1
```

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

```
eroare
```

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

Test

Determinați tipul funcției f : $f\ x\ y = fst\ x\ ++\ head\ y$

Obs: Nu este necesar să scrieți reguli de sinteză de tip dar trebuie să se vadă pașii (sinteza componentelor), nu doar rezultatul final.

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

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
```

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

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

Minimal complete definition:
?



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
```

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

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

Minimal complete definition:
(<=) SAU compare



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
```

```
*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'
```


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

Exemplu (cu verificări la calculator)

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

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



Polimorfism și clase – Cuprins

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

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

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

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 = Data.Foldable.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 = Data.Foldable.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

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 <>)

- 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)

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)

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip

Polimorfism parametric

Polimorfism ad-hoc

Clasă

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

Polimorfism parametric

Polimorfism ad-hoc

Clasă

Definire clasă

Derivare clasă

Instanțiere clasă

Context

Clase uzuale

Clase pentru containere

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: newtype, data / type

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

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

Rezumat

Mecanisme pentru definirea tipurilor / sinonimelor de tip: `newtype`, `data` / `type`

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`