PARADIGME DE PROGRAMARE

Curs 7a

Tipare tare / slabă / statică / dinamică. Tipuri și expresii de tip. Tipuri definite de utilizator.

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipare tare / slabă

- **Tipare tare:** **nu se permit operații pe argumente care nu au tipul corect (se convertește tipul numai în cazul în care nu se pierde informație la conversie)**

  Exemplu: 1+"23" → eroare (Racket, Haskell)

- **Tipare slabă:** **nu se verifică corectitudinea tipurilor, se face cast după reguli specifice limbajului**

  Exemplu: 1+"23" = 24 (Visual Basic)
  1+"23" = "123" (JavaScript)

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipare statică / dinamică

- **Tipare statică:** **verificarea tipurilor se face la compilare**
  – atât variabilele cât și valorile au un tip asociat

  Exemple: C++, Java, Haskell, ML, Scala, etc.

- **Tipare dinamică:** **verificarea tipurilor se face la execuție**
  – numai valorile au un tip asociat

  Exemple: Python, Racket, Prolog, Javascript, etc.

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipuri primitive în Haskell

Tip

Bool = [True, False]

Char = [.. 'a', 'b', ..]

Int = [.. -1, 0, 1, ..]

Altele: **Integer**, **Float**, **Double**, etc.

Tipare expresie (:t expr)

True :: Bool

'a' :: Char

(fib 0) :: Int

Constructori de tip

Constructor de tip = „funcție” care creează un tip compus pe baza unor tipuri mai simple

- **(, , ...)** : $MT^n \longrightarrow MT$ (MT = mulțimea tipurilor)
  - $(t_1, t_2, \ldots, t_n)$ = **tuplu** cu elemente de tipurile $t_1, t_2, \ldots, t_n$
  - **Ex:** (Bool, Char) echivalent cu (,) Bool Char
- **[ ]** : $MT \longrightarrow MT$
  - [t] = **listă** cu elemente de tip t
  - **Ex:** [Int] echivalent cu [ ] Int
- **->** : $MT^2 \longrightarrow MT$
  - t_1 -> t_2 = **funcție** de un parametru de tip t_1 care calculează valori de tip t_2
  - **Ex:** Int -> Int echivalent cu (->) Int Int

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

```
add :: Int -> (Int -> Int)
```

echivalent cu

```
add :: Int -> Int -> Int
```

întrucât **-> este asociativ la dreapta**

- Cum am interpreta tipul `(Int -> Int) -> Int`?

Tipul funcțiilor n-are

Exemplu: `add x y = x + y` (pentru simplitate, presupunem că + merge doar pe Int)

- Având în vedere că toate funcțiile sunt curry, care este tipul lui `(add 2)`?

```
(add 2) :: Int -> Int
```

- În aceste condiții, care este tipul lui `add`?

```
add :: Int -> (Int -> Int)
```

echivalent cu

```
add :: Int -> Int -> Int
```

întrucât **-> este asociativ la dreapta**

- Cum am interpreta tipul `(Int -> Int) -> Int`?

o funcție care – primește o funcție de la Int la Int
– întoarce un Int

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Expresii de tip

- Expresiile reprezintă valori / expresiile de tip reprezintă tipuri

  Exemple: `Char, Int -> Int -> Int, (Char, [Int])`

- **Declararea signaturii** unei funcții (opțională în Haskell) **f :: exprDeTip**
  = asociere între numele funcției și o expresie de tip, cu rol de:
  - **Documentare** (ce ar trebui sa facă funcția)
  - **Abstractizare** (surprinde cel mai general comportament al funcției, funcția percepută ca operator al unui anumit TDA sau al unei clase de TDA-uri)
  - **Verificare** (Haskell generează o eroare dacă intenția (declarația) nu se potrivește cu implementarea)

Exemplu - myMap

`myMap :: (a -> b) -> [a] -> [b]` trebuie să:

primească: o funcție de la un tip oarecare a la un tip oarecare b

o listă de elemente de același tip oarecare a

întoarcă: o listă de elemente de același tip oarecare b

- Dacă implementăm myMap astfel:

```
1. myMap :: (a -> b) -> [a] -> [b]
2. myMap f [] = []
3. myMap f (x:xs) = f (f x) : myMap f xs
```

compilatorul va da eroare, arătând că funcția nu se comportă conform declarației.

- Fără declarația de tip, Haskell ar fi dedus singur tipul lui myMap și ne-ar fi lăsat să continuăm cu o funcție care nu face ceea ce dorim.

Observații

Verificarea strictă a tipurilor înseamnă:

- Mai **multă siguranță** („dacă trece de compilare atunci merge”)
- Mai **puțină libertate**
  - Listele sunt neapărat omogene: **[a]**
    - Contrast cu liste ca `'(1 'a #t)` din Racket
  - Funcțiile întorc mereu valori de un același tip: **f :: ... -> b**
    - Contrast cu funcții ca **`member`** din Racket (care întoarce o listă sau #f)

Tipare – Cuprins

- Tipare tare / slabă
- Tipare statică / dinamică
- Tipuri primitive și constructori de tip
- Expresii de tip
- Tipuri definite de utilizator

Tipuri definite de utilizator

- Cuvântul cheie **data** dă utilizatorului posibilitatea definirii unui TDA cu implementare completă (constructori, operatori, axiome)

```
data ConsTip = Cons1 t11 .. t1i |
               Cons2 t21 .. t2j | ... |
               Consn tn1 .. tnk
```

Numele constructorilor valorilor tipului și tipurile parametrilor acestora (dacă au)

Exemple

```
data RH = Pos | Neg                    -- doar constructori nulari
data ABO = O | A | B | AB              -- doar constructori nulari
data BloodType = BloodType ABO RH      -- constructor extern
```

Exemplu – Tipul Natural

```
1.  data Natural = Zero | Succ Natural -- constructori nular și intern
2.                 deriving Show -- face posibilă afișarea valorilor tipului
3.  unu = Succ Zero
4.  doi = Succ unu
5.  trei = Succ doi
6.
7.  addN :: Natural -> Natural -> Natural -- arată exact ca axiomele
8.  addN Zero n = n
9.  addN (Succ m) n = Succ (addN m n)

addN unu trei                       -- Succ (Succ (Succ (Succ Zero)))
```

Constructorii valorilor unui TDA

Dublă utilizare a constructorilor valorilor unui TDA

- Compun noi valori pe baza celor existente (comportament de **funcție**)

  Exemple:
  ```
  unu = Succ Zero
  doi = Succ unu
  ```

- Descompun valori existente în scopul identificării structurii lor (comportament de **pattern**)

  Exemple:
  ```
  addN Zero n = n
  addN (Succ m) n = Succ (addN m n)
  ```

Variante de tipuri definite de utilizator

- Tipuri enumerate (tipuri sumă)
- Tipuri înregistrare (tipuri produs)
- Tipuri recursive
- Tipuri parametrizate

Tipuri enumerate

- Numite și tipuri sumă (| face suma/reuniunea valorilor tipului)
- Enumeră toate valorile tipului, sub forma
  data ConsTip = Val_1 | Val_2 | ... | Val_n

Exemple

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

```
*Main> :i Bool
data Bool = False | True        -- Defined in `GHC.Types'
```

Tipuri înregistrare

- Numite și tipuri produs: o valoare a tipului se obține prin combinarea unor valori de alte tipuri, sub forma
  data ConsTip = Cons {câmp$_1$ :: tip$_1$, ... câmp$_n$ :: tip$_n$}
  care este o variantă cu funcții selector pentru definiția
  data ConsTip = Cons tip$_1$ ... tip$_n$
- Au un corespondent în majoritatea limbajelor de programare (ex: struct în C++)

Exemplu

```
data Person = Person {name :: (String, String), age :: Int}
fc :: Person
fc = Person ("Frederic","Chopin") 211
composer = name fc
```

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 parametrizate – în general

- **Constructorii de valori** ale tipului (ex: **:**, **Succ**)
  - Pot primi valori ca argumente pentru a produce noi valori

    Exemple:
    ```
    unu = Succ Zero
    lista_unu = 1 : []
    ```

- **Constructorii de tip** (ex: **[ ]**, **(,)**, **->**)
  - Pot primi tipuri ca argumente pentru a produce noi tipuri

    Exemple:
    ```
    [Int], [Char], [[Char]]
    (Int, Char), ([Char], Int, Int)
    ```
  - Când TDA-ul sau funcțiile noastre se comportă la fel indiferent de tipul valorilor pe care le manipulează, folosim variabile (parametri) de tip

    Exemple:
    ```
    [a]     - o listă cu elemente de un tip oarecare a
    (a,b)   - o pereche de un element de un tip oarecare a
                        și un altul de un tip oarecare b
    ```

Tipuri parametrizate – definite de utilizator

- 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]
```

Exemplu – Tipul (Maybe a)

Tipul **(Maybe a)** există în Haskell și este definit astfel:

```
data Maybe a = Nothing | Just a
```

- Se folosește pentru situații când funcția întoarce sau nu un rezultat (de exemplu pentru funcții de căutare care ar putea să găsească sau nu ceea ce caută)
- În funcție de ce tip de valoare va stoca acest tip de date atunci când are ce stoca, constructorul de tip va produce un `Maybe Int` sau un `Maybe Char`, etc.

Exemplu de instanțiere (Maybe a)

Să se găsească suma pară maximă dintre sumele elementelor listelor unei liste de liste, dacă există (ex: `findMaxEvenSum [[1,2,3,4,5],[2,2],[2,4]] = Just 6`).

```
1. --findMaxEvenSum :: [[Int]] -> Maybe Int
2. findMaxEvenSum [] = Nothing
3. findMaxEvenSum (l:ls)
4.     | even lsum = case findMaxEvenSum ls of
5.                     Just s -> Just (max lsum s)
6.                     _ -> Just lsum
7.     | otherwise = findMaxEvenSum ls
8.       where lsum = sumL l
```

Din cauză că sumL este declarat ca
sumL :: [Int] -> Int
funcția va întoarce un (Maybe Int)

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")]
```

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 Person2 = Person2 (Name, Age) deriving Show
fc2 :: Person2
fc2 = Person2 (("Frederic","Chopin"), 211)
```

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional		
Funcții		
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții		
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching		
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare		
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare		
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare	Aplicativă	Lenesă
Tipare		

Comparație Racket - Haskell

	Racket	Haskell
Pur funcțional	Nu	Da
Funcții	Automat uncurry	Automat curry
Pattern matching	Nu	Da
Legare	Locală – statică, top-level - dinamică	Statică
Evaluare	Aplicativă	Lenesă
Tipare	Tare, dinamică	Tare, statică

Rezumat

Tipare tare/slabă

Tipare statică/dinamică

Constructori de tip

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică

Constructori de tip

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri enumerate

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri enumerate: data ConsTip = $Val_1 \mid Val_2 \mid ... \mid Val_n$

Tipuri înregistrare

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri enumerate: data ConsTip = $Val_1 \mid Val_2 \mid ... \mid Val_n$

Tipuri înregistrare: data ConsTip = Cons {$câmp_1$ :: tip_1, ... $câmp_n$ :: tip_n}

Tipuri recursive

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri enumerate: data ConsTip = $Val_1 \mid Val_2 \mid ... \mid Val_n$

Tipuri înregistrare: data ConsTip = Cons {$câmp_1$ :: tip_1, ... $câmp_n$ :: tip_n}

Tipuri recursive: data ConsTip = ... | $Cons_i$ .. ConsTip .. | ...

Tipuri parametrizate

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri enumerate: data ConsTip = $Val_1 \mid Val_2 \mid ... \mid Val_n$

Tipuri înregistrare: data ConsTip = Cons {$câmp_1$ :: tip_1, ... $câmp_n$ :: tip_n}

Tipuri recursive: data ConsTip = ... | $Cons_i$ .. ConsTip .. | ...

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Construcția „type”

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = $Cons_1\ t_{11} .. t_{1i} \mid ... \mid Cons_n\ t_{n1} .. t_{nk}$

Tipuri enumerate: data ConsTip = $Val_1 \mid Val_2 \mid ... \mid Val_n$

Tipuri înregistrare: data ConsTip = Cons {$câmp_1$ :: tip_1, ... $câmp_n$ :: tip_n}

Tipuri recursive: data ConsTip = ... | $Cons_i$ .. ConsTip .. | ...

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Construcția „type”: creează sinonime de tip (nu noi tipuri)

Construcția „newtype”

Rezumat

Tipare tare/slabă: absența / prezența conversiilor implicite de tip

Tipare statică/dinamică: verificarea tipurilor se face la compilare / execuție

Constructori de tip: (,,..), [], ->, tipurile definite cu „data”

Declararea signaturii: f :: exprTip

Construcția „data”: data ConsTip = $Cons_1\ t_{11} .. t_{1i} | ... | Cons_n\ t_{n1} .. t_{nk}$

Tipuri enumerate: data ConsTip = $Val_1 | Val_2 | ... | Val_n$

Tipuri înregistrare: data ConsTip = Cons {$câmp_1$ :: tip_1, ... $câmp_n$ :: tip_n}

Tipuri recursive: data ConsTip = ... | $Cons_i$ .. ConsTip .. | ...

Tipuri parametrizate: (a,b), [a], a -> b, data ConsTip a b ...

Construcția „type”: creează sinonime de tip (nu noi tipuri)

Construcția „newtype”: data ConsTip = $Cons_1\ t_1$ (un singur constructor cu un singur parametru)