PARADIGME DE PROGRAMARE

Curs 14

Tipare. 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 permite operații pe argumente care nu au tipul corect (convertește tipul numai dacă nu se pierde informație la conversie)**

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

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

  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: **verifică tipurile la compilare**
  – atât variabilele cât și valorile au un tip asociat

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

- Tipare dinamică: **verifică tipurile 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 cu 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 interpretăm 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 interpretăm tipul `(Int -> Int) -> Int`?

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])`

- **f :: exprDeTip** = declarație de tip (asociere între numele funcției și o expresie de tip)

- Declararea signaturii este opțională în Haskell.
  - Rol:
    - Documentare: ce face funcția
    - Abstractizare: cel mai general comportament al funcției
      (funcție = operator al unui TDA sau al unei clase de TDA-uri)
    - Verificare: provoacă eroare dacă intenția (declarația) nu se potrivește cu implementarea

Exemplu - myMap

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

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

o listă de elemente de același tip oarecare a

întoarce: o listă de elemente de același tip oarecare b

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

- Implementarea provoacă eroare, întrucât nu respectă declarația
- Fără declarația de tip, Haskell ar fi dedus un alt tip pentru myMap, și nu am fi detectat eroarea semantică.

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 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 posibilitatea definirii unui TDA cu implementare completă (constructori, operatori, axiome)

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

- numele constructorilor de date (alese de programator) +
- 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)))
```

Constructori de date

Constructorii de date au o dublă utilizare:

- Funcție: compun noi valori pe baza celor existente

```
unu = Succ Zero
doi = Succ unu
```

- Pattern: descompun valori existente în scopul identificării structurii lor

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

Tipuri definite de utilizator – variante

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

Tipuri enumerate

data ConsTip = Val_1 | Val_2 | ... | Val_n

- Numite și tipuri sumă: | face suma/reuniunea valorilor tipului
- Enumeră valorile tipului: fiecare valoare este un constructor de date nular

Exemple

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

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

Tipuri înregistrare

data ConsTip = Cons {câmp$_1$ :: tip$_1$, ... câmp$_n$ :: tip$_n$}

- Variantă cu funcții selector pentru definiția **data** ConsTip = Cons tip$_1$ ... tip$_n$
- Numite și tipuri produs: o valoare reprezintă o combinație de valori de alte tipuri
- Au 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") 216
composer = name fc
```

Tipuri recursive

```
data ConsTip = .. | Cons_i .. ConsTip .. | ..
```

- Tipuri pentru care specificăm și cel puțin un constructor intern

Exemple

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

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

Tipuri parametrizate – în general

- Constructori de date: primesc valori ⟶ produc valori ex: Succ :

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

- Constructori de tip: primesc tipuri ⟶ produc tipuri ex: [] (,) ->

```
[Int]  (Int, Char)   [Int] -> Bool
```

- Variabile de tip: substituie constantele de tip când TDA-ul / funcția gestionează datele la fel, indiferent de tipul lor concret ex: a b

```
map :: (a -> b) -> [a] -> [b]
```

  - Nu contează: tipul elementelor listei, tipul rezultatului funcției
  - Contează: tipul elementelor listei ~ tipul inputului funcției (de aceea folosim aceeași variabilă a)

Tipuri parametrizate – definite de utilizator

```
data ConsTip a b … =
```

- Definire: Aplică constructorul de tip pe una sau mai multe variabile de tip (analog cu [a], (a, b), a -> b, etc.)
- Instanțiere: Variabila de tip se leagă la un tip concret atunci când folosim constructorii de date asupra unor valori concrete.

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)

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

- Constructor de tip
- Parametru de tip
- Constructor de date (indică absența rezultatului)
- Constructor de date cu parametu de tip a (împachetează un rezultat existent)

- Maybe a: tip predefinit, cu scopul de a rezolva problema funcțiilor care pot „eșua” (ex: căutări)

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

- `a` tipul cheii căutate
- `[(a, b)]` tipul dicționarului: listă de perechi cheie-valoare, unde cheia are tipul a
- `Maybe b` tipul rezultatului căutării: împachetează un rezultat de tip b (tipul valorilor în dicționar) sau spune că dicționarul nu conține o asemenea asociere (ambele opțiuni aparțin aceluiași tip Maybe b)

Instanțiere – exemplu

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
```

Întrucât sumL este declarat ca
sumL :: [Int] -> Int
findMaxEvenSum întoarce Maybe Int

Construcția `type`

- Creează sinonime de tip, cu rol de:
  - Documentare: tipul `Age` este mai clar decât tipul `Int`
  - Concizie: tipul `Name` este mai scurt (și mai clar) decât `(String, String)`

Exemple

```
type Age = Int

type Name = (String,String)

names :: [Name]

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

Construcția **newtype**

- Creează tipuri noi folosind un singur constructor cu un singur parametru
- Rol: tip distinct, cu operații dedicate, fără overhead la execuție
  - **data**: tipurile definite cu `data` pot avea oricâți constructori cu oricâți parametri
    - la execuție: constructorul de date există fizic în memorie
      datele trebuie despachetate pentru a fi accesate
  - **newtype**: la compilare: constructorul de date este șters
    valoarea devine echivalentă cu valoarea parametrului
    - la execuție: cost de despachetare zero

Exemplu

```
newtype Person2 = Person2 (Name, Age) deriving Show
fc2 :: Person2
fc2 = Person2 (("Frederic","Chopin"), 216)
```

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 = $Cons_1\ t_{11} .. t_{1i}$ | ... | $Cons_n\ t_{n1} .. t_{nk}$

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 = $Cons_1\ t_{11} .. t_{1i}$ | ... | $Cons_n\ t_{n1} .. t_{nk}$
Tipuri enumerate:	data ConsTip = Val_1 | Val_2 | ... | 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 = $Cons_1\ t_{11} .. t_{1i}$ | ... | $Cons_n\ t_{n1} .. t_{nk}$
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

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 = $Cons_1\ t_{11} .. t_{1i}$ | ... | $Cons_n\ t_{n1} .. t_{nk}$
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	
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 = $Cons_1\ t_{11} .. t_{1i}$ | ... | $Cons_n\ t_{n1} .. t_{nk}$
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”	
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”	

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 constructor cu un parametru)