Documentatie laborator 8 – Mai mult Haskell

Mai mult despre functii

Functii anonime

\x -> x+y -- (1)

\x y -> x+y -- (2)

\x -> \y -> x+y -- (3)

Semnul "\" este ales pentru ca arata cel mai asemanator cu lambda.

(1) functia cu argumentul x si corpul x+y.

(2) functia cu argumentele x, y si corpul x+y.

(3) functia cu argumentul x care intoarce o functie cu argumentul y si corpul x+y.

In fapt, definitiile 2 si 3 sunt echivalente, intrucat functiile din Haskell sunt by default curry.

Aplicare partiala

Multumita faptului ca functiile sunt curry, o functie de mai multe argumente poate fi aplicata numai pe o parte dintre ele. Rezultatul este o functie care "asteapta" restul de argumente pentru a calcula ceea ce calcula functia initiala.

*multiply x y = x*y*

multiply 3 va avea tipul Int -> Int si va intoarce o functie care asteapta un argument ca sa il inmulteasca cu 3.

map (multiply 3) lista va inmulti toate elementele listei cu 3.

Mai mult despre liste

Intrucat Haskell este un limbaj pur functional care foloseste evaluare lenesa, listele in Haskell sunt by default fluxuri. Acest lucru permite definitii in genul urmator:

[0,0..] -- flux de zerouri

zeros = 0:zeros -- acelasi flux de zerouri

[0..] -- fluxul numerelor naturale

take 5 ['a'..] -- va intoarce "abcde"

[1,3..20]-- va intoarce [1,3,5,7,9,11,13,15,17,19]

Functia if

Haskell pune la dispozitie o functie if, insa este de preferat sa nu o folositi in loc de pattern match. Sintaxa este: if <conditie> then <valoare-true> else <valoare-false>.

sign x = if x<0 then -1 else 1

Definitii locale

where

Fiecare ecuatie (eventual conditionala) poate fi urmata de o serie de definitii locale.

Aceste definitii sunt precedate de cuvantul "where".

unZip [] = ([], [])

unZip ((a,b):rest) = (a:l, b:r)

where

(l,r) = unZip rest

SAU

unZip [] = ([],[])

unZip ((a,b):rest) = (a:l,b:r)

where

uzRest = unZip rest

l = fst uzRest

r = snd uzRest

let

let x = 5 in x^2 + x + 1 inseamna „fie x=5 in expresia x^2 + x + 1”.

Mai mult despre inferenta de tip

Pentru a vedea ce tipuri sintetizeaza Haskell pentru functiile definite de voi, folositi :t.

```
Main> :t unZip
unZip :: [(a,b)] -> ([a],[b])
```

In tipul de mai sus, a si b sunt variabile de tip, stand pentru tipuri oarecare. In momentul in care aplic efectiv unZip pe o lista, de exemplu unZip [(15,True),(30,False)] variabila a se leaga la tipul Integer iar variabila b la tipul Bool.

Tipuri de date utilizator

In stransa legatura cu verificarea stricta a tipurilor de date in Haskell, limbajul nu permite lucrul cu liste eterogene (pe care il aveam la dispozitie in Scheme). Pentru a putea manipula structuri complexe si in Haskell, avem la dispozitie tipuri de date utilizator. Sintaxa definirii acestora este:

data NumeTip = ConstructorDeTip1 | ConstructorDeTip2 | … | ConstructorDeTipn

Numele tipului si numele constructorilor trebuie sa inceapa cu majuscula (la fel cum numele variabilelor in Haskell nu pot incepe cu majuscula, pentru a nu isca o confuzie intre ce este variabila si ce este tip / constructor de tip).

Tipuri enumerate

data Zi = Luni | Marti | Miercuri | Joi | Vineri | Sambata | Duminica

Pentru a defini functii pe astfel de tipuri se foloseste pattern match.

nrCeasuriRele zi :: Zi -> Int

nrCeasuriRele Marti = 3

nrCeasuriRele _ = 0

Tipuri produs

data StudentPP = Student Nume Nota

unde Nume si Nota sunt aliasuri pentru String si Int, definite folosind type:

type Nume = String

type Nota = Int

Am fi putut folosi un tuplu pentru a retine un nume si o nota (de exemplu: type StudentPP = (Nume, Nota)), insa dezavantajele ar fi:

- am putea confunda o alta pereche de un String si un Int cu un student la PP
- codul si mesajele de eroare ar fi mai putin documentate (tipul definit de utilizator apare ca atare in mesajele de eroare, iar orice student la PP va trebui sa apara in cod ca fiind construit cu cuvantul Student – care documenteaza exact scopul valorii respective)

Functiile definite pe astfel de tipuri folosesc tot pattern match.

adaugaBonus (Student _ nota) bonus = nota+bonus

```
Main> adaugaBonus (Student "Foarte Bun" 10) 1
11
```

Tipuri recursive

data Natural =

Zero |

Succ Natural

deriving Show

Expresia deriving Show ne permite afisarea valorilor de tip Natural.

Multumita mecanismului de pattern match, functiile definite pe aceste tipuri seamana foarte bine cu axiomele TDA-ului:

add Zero n = n

add (Succ m) n = Succ (add m n)

```
Main> add (Succ Zero) (Succ (Succ Zero))
Succ (Succ (Succ Zero))
```

Similar se pot defini tipuri mutual recursive.