

Chapter 1

Polimorfism

În Haskell, funcțiile suportă două tipuri de polimorfism:

1. polimorfism parametric;
2. polimorfism ad-hoc.

N.B. Tipurile de date suportă doar polimorfism parametric.

În polimorfismul parametric, funcția tratează uniform, la fel, toate tipurile posibile.

```
data Lista a = Vida | Cons a (Lista a) deriving (Eq, Show)
-- liste de elemente de tip a.

-- polimorfism parametric
lungime :: Lista a -> Int
lungime Vida = 0
lungime (Cons x tl) = 1 + lungime tl
```

Cu alte cuvinte, dacă avem o `Lista Int` sau o `Lista Bool` de aceeași lungime, funcția nu diferențiază între ele.

Altfel spus: dacă avem o `Lista Int` și înlocuim fiecare `Int` cu un `Bool` în mod consistent, funcția întoarce un același rezultat.

În polimorfismul ad-hoc, variabilele de tip sunt constrânse de apartenența la o clasă de tipuri:

```
-- polimorfism ad-hoc
qs :: Ord a => [a] -> [a]
--      ^^^^^
qs [] = []
qs (hd:tl) = qs (filter (<=hd) tl) ++ [hd] ++ qs (filter (>hd) tl)
```

În funcție de cum este definită instanța `Ord a`, funcția `qs` poate avea comportamente diferite (de exemplu, să ordoneze crescător o `[Int]` și “descrescător” o `[MyInt]`).

N.B. Marca polimorfismului parametric: lipsa constrângerilor de clasă asupra variabilelor de tip.

N.B. Marca polimorfismului ad-hoc: prezența constrângerilor.

O funcție poate prezenta atât polimorfism ad-hoc, cât și polimorfism parametric:

```
ex :: Ord a => a -> a -> b -> b -> b
ex x y u v = if x < y then u else v
```

Polimorfismul parametric este un instrument puternic pentru a face raționamente despre cod (doar uitându-ne la semnatura funcției, fără definiția asociată).

1.0.1 Exemplul 1

Spre exemplu, singura funcție cu semnatura

```
ex1 :: a -> b
```

este funcția care efectuează o buclă infinită pentru orice parametru:

```
ex1 x = ex1 x
```

Intuitiv, `ex1` nu știe cum altfel să producă o valoare de tip `b`, fiindcă `b` nu apare printre parametri.

1.0.2 Exemplul 2

Singurele funcții cu semnatura

```
ex2 :: a -> a
```

sunt:

1. funcția care buclează la infinit;
2. funcția identitate:

```
ex2 x = x
```

1.0.3 Exemplul 3

Pentru o funcție cu semnatura:

```
ex3 :: [a] -> a
```

sunt mai multe opțiuni:

1. buclă infinită;
2. să întoarcă primul element;
3. să întoarcă al doilea element;
4. să întoarcă ultimul element;
5. să întoarcă un al n -lea element (în funcție de lungimea listei);
6. să bucleze la infinit pentru anumite lungimi de lista, dar să returneze un al n -lea element pentru celelalte lungimi.

În orice caz, avem următoarea garanție: dacă funcția se oprește, valoarea întoarsă este una dintre cele din listă (nu știu să produc alte elemente de tip `a`, fiindcă `a` ar putea la fel de bine să fie *orice* tip).

1.0.4 Teoreme gratuite

Avantajul polimorfismului parametric este dat de prezența așa numitor teorem gratuite.

Iată un exemplu de teoremă despre `ex3`:

Teoremă. Pentru orice implementare a funcției `ex3` cu semnatura de mai sus, dacă `ex3 [1,2,3] == 2`, atunci `ex3 ['a','b','c'] == 'b'`.

Cei pasionați pot studia mai multe despre teoremele gratuite: <https://people.mpi-sws.org/~dreyer/tor/papers/wadler.pdf>.

Chapter 2

Evaluare leneșă

Cele mai multe limbaje de programare mainstream au evaluare (implicit) de tip “eager”. Evaluarea în Haskell este de tip “lazy”.

În evaluarea *eager*:

1. Pentru execuția unei instrucțiuni de atribuire $x = \text{expresie}$ se evaluează întâi rezultatul expresiei, iar apoi rezultatul se memorează în variabila x .
2. Pentru apelul unei funcții $f(\text{exp1}, \dots, \text{expn})$, se evaluează întâi fiecare expresie exp_i ($1 \leq i \leq n$), iar apoi funcția este apelată propriu-zis.

Haskell are o strategie de evaluare care este la polul diametral opus: nicio expresie nu este evaluată decât dacă rezultatul evaluării este necesar în calculul final.

Putem observa acest lucru efectuând câteva experimente:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib x = fib (x - 1) + fib (x - 2)      --- (*)

-- Experimentul 1
a :: Int
a = let x = fib 100 in 2

-- Experimentul 2
b :: Int
b = let x = fib 100 in
     let y = 13 in
     if y > 10 then 2 else x

-- Experimentul 3
f :: Int -> Int -> Int
f x y = if x > 2 then y else x      --- (**)
```

Observați că evaluarea expresiei **a** durează foarte puțin timp.

Observați că evaluarea expresiei **b** durează foarte puțin timp.

Observați că evaluarea expresiei **f 1 (fib 100)** durează foarte puțin timp.

Strategia de evaluare lazy, spre deosebire de strategia folosită în limbajele C/C++/Java, nu necesită evaluarea argumentelor înainte de apelul funcției.

Cu evaluare eager, expresia **f 1 (fib 100)** ar fi evaluată după cum urmează:

```
-- Varianta 1 (eager):
f 1 (fib 100) =
    (*)
f 1 (fib (100 - 1) + fib (100 - 2)) =
    (definitia "-")
f 1 (fib 99 + fib (100 - 2)) =
    (*)
f 1 ((fib (99 - 1) + fib (99 - 2)) + fib (100 - 2)) =
-- s.a.m.d.
```

Astfel, expresia `f 1 (fib 100)` este evaluată în Haskell după cum urmează:

```
-- reguli sistem:
-- if True then x else y = x      (3*)
-- if False then x else y = y    (4*)

-- Varianta 2 (lazy):
f 1 (fib 100) =
    (**)
if 1 > 2 then (fib 100) else 1 =
    (def. >)
if False then (fib 100) else 1 =
    (4*)
1.
```

De altfel, strategia lazy permite scrierea unei funcții care are comportament similar cu construcția `if-then-else`:

```
ite :: Bool -> a -> a -> a
ite True x y = x
ite False x y = y
```

În limbajul C, o astfel de funcție nu ar fi posibilă:

```
int ite(bool b, int x, int y)
{
    if (b) {
        return x;
    } else {
        return y;
    }
}
```

deoarece ar forța evaluarea argumentelor `x` și `y`.

În practică, fiecare variabilă în Haskell nu conține o valoare, ci este un pointer spre o bucată de cod (thunk) care, dacă este executată, produce valoarea în cauză.

De exemplu, în programul `let x = fib 100 in 2`, variabila `x` conține un pointer spre un cod care calculează `fib 100`.

Există mai multe strategii lazy, iar Haskell folosește o strategie numită call-by-need, care evită evaluarea de mai multe ori pentru același argument. Vom studia mai multe despre strategii de evaluare în secțiunea despre lambda-calcul.

Un avantaj al evaluării leneșe este că permite definirea de structuri de date aparent infinite:

```

listaAux :: Int -> [Int]
listaAux i = i : (listaAux (i + 1))  -- (E6)

listaNat :: [Int]
listaNat = listaAux 0                -- (E5)

```

Conceptual, `listaNat` este lista tuturor numerelor naturale (în practică, `listaNat` este o funcție care calculează această listă (dacă/când este necesar)).

Considerând funcțiile `(!!)` și `map` din biblioteca standard:

```

-- (!! ) :: [a] -> Int -> a
-- (!! ) (x:xs) 0 = x                (E1)
-- (!! ) (x:xs) n = (!! ) xs (n-1)  (E2)
-- map :: (a -> b) -> [a] -> [b]
-- map f [] = []                    (E3)
-- map f (x:xs) = (f x) : map f xs  (E4)

```

evaluarea expresiei `(map (+13) listaNat) !! 0` nu s-ar opri fără evaluare leneșă (deoarece apelul `(map (+13) listaNat)` ar bucla la infinit).

În schimb, cu evaluare leneșă, expresia `(map (+13) listaNat) !! 0` este evaluată după cum urmează:

```

(map (+13) listaNat) !! 0 =
                                (E5)
(map (+13) (listaAux 0)) !! 0 =
                                (E6)
(map (+13) (0 : (listaAux (0 + 1)))) !! 0 =
                                (E4)
((+13) 0 : map (+13) (listaAux (0 + 1))) !! 0 =
                                (E1)
(+13) 0 =
                                (def. +)

```

13.

Avantajul evaluării eager este că este mai ușor de raționat despre timpul de execuție al unui program.

În limbajele transparente referențial, valoarea unei expresii este aceeași indiferent de strategia de evaluare.

În limbajele care nu sunt transparente referențial, aceeași expresie poate să producă rezultate diferite, în funcție de strategia de evaluare.

Folosirea unei strategii lazy este critică într-un limbaj pur funcțional (cum este Haskell) pentru a scrie programe imperative (e.g., care trimit mesaje în rețea, citesc un fișier, interacționează cu utilizatorul etc.)