Haskell: Polimorfism și clase

- Data publicării: 19.04.2022
- Data ultimei modificări: 19.04.2022

Obiective

Scopul acestui laborator este de a prezenta implementarea polimorfismului în sistemul de tipuri al limbajului Haskell.

Aspectele urmărite sunt:

- polimorfism
- polimorfism parametric
- polimorfism ad-hoc
- clase

Polimorfism

Polimorfismul este un mecanism al limbajului Haskell (și nu doar al acestuia) prin care se poate defini un **set de operații (interfață comună)** pentru mai multe tipuri. Categoriile de polimorfism pe care le vom întâlni cel mai des în Haskell sunt: parametric și ad-hoc.

Polimorfism parametric

Așa cum ați observat în laboratoarele anterioare, Haskell permite definirea de funcții care operează pe structuri de date generice. Să luăm spre exemplu funcția `length`. Aceasta permite calcularea lungimii oricărei liste, indiferent de tipul elementelor din listă. Această proprietate a limbajului poartă denumirea de **polimorfism parametric**.

Fără acesta, am fi nevoiți să avem câte o versiune a funcției `length` pentru fiecare tip de listă în parte. Am avea, pentru liste de întregi `lengthInts :: [Int] -> Int`, pentru liste de string-uri `lengthStrings :: [String] -> Int`, pentru liste de numere reale `lengthDouble :: [Double] -> Int`, și așa mai departe. Și pentru toate aceste functii definiția este aceeași:

```
length_ [] = 0
length_ (_:xs) = 1 + length_ xs
```

Este evident că lungimea unei liste nu depinde în niciun fel de tipul elementelor din listă. Putem face abstracție de acesta și îl putem înlocui cu o **variabilă de tip**, obținând astfel următorul tip pentru funcția `length`

```
length :: [a] -> Int
```

Semnătura de tip spune: "Pentru orice tip `a`, funcția `length` ia o listă cu elemente din `a` și întoarce un întreg". Spunem în acest moment că funcția `length` este polimorfică.

O funcție poate avea oricâte variabile de tip în semnătura ei. De exemplu funcția `map` are tipul:

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

Cu alte cuvinte definiția lui `map` nu depinde în niciun fel de tipul funcției sau de tipul listei de elemente pe care o va traversa și asupra căreia va aplica funcția argument, atâta vreme cât tipul funcției este compatibil cu tipul elementelor din listă.

Această proprietate a limbajului ne permite să implementăm algoritmi generici, aplicabili într-un număr mare de cazuri, încurajând reutilizarea de cod.

Polimorfism ad-hoc

Să analizăm funcția `elem`. **elem** caută un element într-o listă și întoarce `True` dacă lista conține elementul căutat sau `False` altfel:

```
elem _ []       = False
elem x (y:ys)   = x == y || elem x ys
```

Tipul acestei funcții este:

```
elem :: Eq a => a -> [a] -> Bool
```

Elementul de noutate este `Eq a =>`. Acesta se numește **constrângere de tip** și apare ca urmare a folosirii funcției `(==)`. Spre deosebire de funcția `length`, care putea fi folosită indiferent de tipul listei, funcția `elem` este generică într-un sens mai restrâns. Ea funcționează doar pentru liste cu elemente care definesc egalitatea (operatorul ==).

Cu alte cuvinte, `elem` poate fi aplicată pentru o listă de `Int` sau `String`, pentru că știm să definim egalitatea pentru aceste tipuri, dar nu și pentru o listă de funcții. Deși pare neintuitiv, următoarea expresie:

```
elem (+1) [(+1), (*2), (/3)]
```

va produce următoarea eroare la compilare:

```
    No instance for (Eq (a0 -> a0))
      arising from a use of `elem'
    Possible fix: add an instance declaration for (Eq (a0 -> a0))
    In the expression: elem (+ 1) [(+ 1), (* 2), (/ 3)]
```

Cu alte cuvinte, tipul funcție nu aparține clasei `Eq` și deci nu definește `(==)`.

Revenind la definiția lui `elem`, nu numai că această funcție merge doar pe tipuri care definesc operația de egalitate, dar se va comporta diferit pentru fiecare mod în care egalitatea este implementată. Pentru o implementare diferită a egalității pentru numere întregi, de exemplu, vom obține un comportament diferit pentru funcția `elem`. Spunem că `elem` este **polimorfică ad-hoc**.

Ca să înțelegem mai bine acest concept, trebuie să înțelegem conceptul de clasă din Haskell.

Clase

Am întâlnit conceptele de clasă până acum sub forma constrângerilor de tip, așa cum este și cazul funcției `elem`. Clasele (eng. Type class) sunt un concept esențial în Haskell și unul dintre punctele forte ale limbajului.

Primul lucru de remarcat este că noțiunea de clasă din Haskell și noțiunea de clasă din Java sunt concepte foarte diferite. Nu încercați să înțelegeți una în funcție de cealaltă – asemănarea celor două denumiri este o coincidență.

Clasele din Haskell seamănă mai mult cu conceptul de interfață din Java. O clasă reprezintă un set de funcții care definesc o interfață sau un comportament unitar pentru un tip de date.

Să luăm exemplul clasei Eq, întâlnită mai sus. Ea este definită astfel:

```
class  Eq a  where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Eq definește 2 funcții, (==) și (/=). Pentru **a înrola** un tip în clasa Eq, ambele funcții trebuie implementate.

Echivalentul din Java pentru Eq ar fi următorul:

```
interface Eq<A implements Eq<A>> {
    boolean eq(another: A);
    boolean notEq(another: A);
}
```

În continuare vom analiza cum un tip poate fi înrolat în clasa Eq. În acest scop, vom defini tipul Person:

```
data Person = Person {name :: String, cnp :: Integer}
```

Includem Person în clasa Eq astfel:

```
instance Eq Person where
    Person name1 cnp1 == Person name2 cnp2 = name1 == name2 && cnp1 == cnp2
    p1 /= p2 = not (p1 == p2)
```

Observați că adăugarea Person la Eq se face independent de definiția lui Person – spre deosebire de Java, unde o interfață poate fi implementată de un tip doar la definirea acestuia din urmă.

Această abordare are câteva avantaje:

- decuplarea implementării clasei de definirea tipului – modularitate mai bună
- putem înrola tipuri create anterior în alte biblioteci în clase proaspăt create de noi
- putem defini multiple implementări ale unei clase pentru un acelasi tip de date în module diferite și să importăm doar implementarea care ne interesează într-un anumit caz (însă acest lucru nu este recomandat)

Putem adăuga și tipuri de date generice într-o clasă. Să luăm exemplul tipului BST definit în laboratorul anterior:

```
data BST a = Empty | Node a (BST a) (BST a)
```

Adăugarea acestui tip la clasa Eq se face astfel:

```
instance Eq a => Eq (BST a) where
    Empty == Empty = True
    Node info1 l1 r1 == Node info2 l2 r2 = info1 == info2 && l1 == l2 && r1 == r2
    _ == _ = False

    t1 /= t2 = not (t1 == t2)
```

Observăm că sintaxa Eq a => își face din nou apariția. Forma Eq a => Eq (BST a) înseamnă: "dacă a aparține clasei Eq, atunci și tipul (BST a) aparține clasei Eq". Această constrângere este necesară deoarece pentru a verifica că un nod este egal cu un altul (Node info1 l1 r1 == Node info2 l2 r2, trebuie să verificăm că informațiile corespunzătoare din noduri sunt egale (info1 == info2) – ceea ce înseamnă că elementele stocate în arbore trebuie să aparțină la rândul lor clasei Eq.

Extindere de clase

Haskell permite ca o clasă să extindă o altă clasă. Acest lucru este necesar când dorim ca un tip inclus într-o clasă să fie inclus doar dacă face deja parte dintr-o altă clasă.

De exemplu `Ord`, care conține tipuri cu elemente care pot fi ordonate (`<`, `>`, `>=`, etc.) este definită astfel:

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>), (>=) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

Important de reținut din definiția de mai sus este linia `class (Eq a) => Ord a`. Semnificația aici este: "Dacă `a` este în `Eq`, atunci `a` poate fi în `Ord` dacă definește funcțiile de mai jos."

Aceasta este o definiție naturală pentru clasa `Ord`. Nu are sens să discutăm despre ordonarea elementelor dintr-un tip dacă acesta nu definește ce înseamnă egalitatea dintre elemente.

Membri impliciți

Observăm că de fiecare dată când am definit funcția `(/=)`, am definit-o de fapt ca fiind opusul funcției `(==)`. Acesta poate fi un caz foarte des întâlnit. Ar fi util dacă Haskell ne-ar permite să oferim implementări implicite pentru funcțiile din clase.

Vestea bună e că Haskell pune la dispoziție această facilitate. Iată cum putem defini `Eq`:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    a /= b = not (a == b)
```

Acum, când înrolăm un tip în `Eq` e suficient să implementăm doar `(==)`.

```
instance Eq Person where
    Person name1 cnp1 == Person name2 cnp2 =
        name1 == name2 && cnp1 == cnp2
```

Desigur, dacă dorim, putem suprascrie implementarea implicită pentru `(/=)` cu o implementare proprie.

La drept vorbind, în biblioteca standard Haskell (modulul `Prelude`) `Eq` este definit astfel:

```
class  Eq a  where
    (==), (/=)           :: a -> a -> Bool
    x /= y               = not (x == y)
    x == y               = not (x /= y)
```

Observăm că ambele funcții au implementări implicite. Astfel, atunci când instanțiem Eq putem să implementăm fie (/=), fie (==), fie pe ambele.

Clase predefinite

Biblioteca standard Prelude oferă un set de clase predefinite, care sunt introduse implicit în programele Haskell. Enumerăm câteva:

- Ord – pentru tipuri care pot fi ordonate - definește funcții precum <, >, <=, etc.
- Show – pentru tipuri care pot fi reprezentate ca String-uri - principala funcție este show. Această funcție este folosită și de consola GHCi atunci când afișează rezultatele.
- Read – inversa lui Show - pentru tipuri care pot fi citite din String
- Enum – pentru tipuri care pot fi enumerate - folosită implicit de construcții de forma [a..b] care generează toate elementele între două limite (sau plecând de la un punct de start).
- Num – clasă pentru toate tipurile numerice - definește operațiile aritmetice de bază: (+), (-), (*), etc.
- Integral – clasă pentru tipurile întregi. (Int și Integer sunt incluse aici). Definește funcții ca mod sau div
- Fractional – clasă pentru numere reprezentabile ca fracții - definește funcția (/)
- Floating – clasă pentru numere reale - definește funcții ca sqrt, exp, sin, cos, etc.
- Monad – definește tipuri care pot reprezenta acțiuni monadice. Mai multe despre monade aici: Monade [http://book.realworldhaskell.org/read/monads.html]

Deriving

Am observat că implementările noastre pentru clasa Eq de mai sus au fost relativ simple. În general implementarea lui Eq pentru un tip nou de date presupune verificarea câmp cu câmp a fiecărei componente a noului tip de date. Pentru că este simplu, compilatorul de Haskell poate face asta automat pentru noi, folosind cuvântul cheie deriving.

```
data BST a = Empty | Node a (BST a) (BST a) deriving (Eq)
```

va genera o implementare a clasei Eq similare cu cea făcută de noi mai sus.

Există mai multe clase care pot fi instanțiate în acest fel: Ord, Enum, Bounded, Show, Read.

Num

Poate v-ați pus la un moment dat întrebarea ce tip au expresiile numerice simple, cum ar fi expresia 5. Cu siguranță trebuie să aibă un tip numeric – însă despre ce tip e vorba? 5 este în același timp reprezentarea pentru numărul întreg 5, pentru numărul real 5.0 sau pentru numărul complex 5+0i. Deci, despre care 5 discutăm ?

GHCi ne poate raspunde la aceasta intrebare:

```
:t 5
=> 5 :: Num a => a
```

5 este deci o reprezentare pentru toate tipurile numerice (adică tipurile incluse în clasa Num). 5 este o constantă polimorfică. La fel este de exemplu și expresia 1+5 sau [] (lista vidă):

```
:t 1+5
=> 1+5 :: Num a => a

:t []
=> [] :: [t]
```

Haskell nu constrânge constanta 5 la niciun tip numeric concret până când acest lucru nu este evident din context.

Tipuri de ordin superior

Să considerăm urmatoarea problemă: Dat fiind un arbore binar de tipul `BST a`, să definim o funcție `f` cu tipul `a -> b`, care aplicată pe arbore va genera un arbore de tipul `BST b`. Va menține structura arborelui, dar pentru fiecare element dintr-un nod, va aplica funcția `f` asupra acelui element.

Pentru că semantic seamănă foarte mult cu `map`, o să-i spunem `treeMap`.

```
treeMap :: (a -> b) -> BST a -> BST b

treeMap f Empty = Empty
treeMap f (BST info l r) = BST (f info) (treeMap f l) (treeMap f r)
```

Să presupunem că dorim să facem ceva similar și pentru tipul `Maybe`.

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

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f Nothing = Nothing
maybeMap f (Just x) = Just (f x)
```

Observăm un șablon comun între definițiile lui `map`, `treeMap`, `maybeMap`. Semnăturile lor sunt foarte similare. De fapt ele expun același principiu – definesc structuri de date, sau așa-zise "containere", care pot fi "mapate" – adică putem aplica o funcție pe fiecare din elementele conținute și obținem un nou container cu aceeași forma, dar cu alt conținut.

Întrebarea firească este: putem captura acest concept de containere "mapabile" într-o clasă? Răspunsul este da. Să-i spunem acelei clase `Functor` – această clasă va expune o singură metodă, numită `fmap`, care reprezintă generalizarea funcțiilor `map` de mai sus.

`Functor` poate fi definită în felul următor:

```
class Functor container where
    fmap :: (a -> b) -> container a -> container b
```

Observăm un lucru foarte interesant și anume că putem să definim clase pe tipuri care nu sunt încă complete. Observați că variabila de tip `container` trebuie să fie aplicată pe un tip de date ca să producă un tip de date valid – este deci un constructor de tip oarecare.

Să instanțiem clasa `Functor` pentru arbori binari:

```
instance Functor BST where
   fmap f Empty = Empty
   fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

Similar putem defini `fmap` pentru liste, `Maybe` și alte structuri de date.

Cele spuse mai sus dezvăluie o caracteristică interesantă a sistemului de tipuri Haskell. De exemplu, o funcție în Haskell este ceva de tipul `a -> b`, i.e. ia o valoare de tip `a` si produce o valoare de tip `b`.

BST se comportă ca o funcție, dar la nivel de tip. Constructorii de tip BST iau ca argument un tip de date și produce alt tip de date. De exemplu, `BST Int` ia tipul Int ca argument și produce un tip de date care reprezintă un arbore de întregi. Spunem că BST este un **tip de ordin superior** (higher-order type) - în cazul de față este un **constructor de tip unar**.

Pentru a afla informații despre clase în `ghci` se poate utiliza comanda `:info <typeclass` unde typeclass este clasa despre care dorim să aflăm informații.

Mai multe detalii aici [http://www.haskell.org/tutorial/classes.html]

Resurse

- Cheatsheet [https://github.com/cs-pub-ro/PP-laboratoare/blob/master/haskell/clase/haskell-cheatsheet-3.pdf]
- Schelet [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/clase-schelet.zip]
- Soluții [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/clase-solutii.zip]