Programare Funcţională în Haskell

La finalul articolului trecut reuşisem să obţinem o aplicaţie simplă ce permitea căutarea unor informaţii despre persoane în 3 tabele (reprezentate ca liste de perechi). Pentru completitudine, vom prezenta în continuare codul cu care am terminat articolul trecut.

Începem cu un set de extensii ale compilatorului ce ne vor permite să fim mai expresivi.

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleInstances #-}
```

Continuăm cu definiţiile tipurilor de date:

```
type Name = String
type Age = Int
type Address = String
type PhoneNumber = Integer

newtype NameAgeTable = NAgT [(Name, Age)] deriving Show
newtype NameAddressTable = NAdT [(Name, Address)] deriving Show
newtype NamePhoneTable = NPT [(Name, PhoneNumber)] deriving Show
```

Ne amintim că `deriving Show` îi spune compilatorului să definească automat câte o metodă `show` pentru a putea converti tipul de date la `String` pentru fiecare tip de date.

Populăm cele 3 tabele cu valori de test:

```
nameAge = NAgT [("Ana", 24), ("Gabriela", 21), ("Mihai", 25), ("Radu", 24)]
nameAddress = NAdT [("Mihai", "a random address"), ("Ion", "another address")]
namePhone = NPT [("Ana", 2472788), ("Mihai", 24828542)]
```

Definim o clasă pentru căutarea după nume în aceste tabele şi înrolăm cele 3 tipuri la aceasta. Spre deosebire de `deriving`, aici va trebui să definim noi metoda. Vom folosi funcţia predefinită `lookup` pentru a căuta într-o listă de perechi.

```
class SearchableByName t a | t -> a where
   search :: Name -> t -> Maybe a

instance SearchableByName NameAgeTable Age where
  search name (NAgT l) = lookup name l

instance SearchableByName NameAddressTable Address where
  search name (NAdT l) = lookup name l

instance SearchableByName NamePhoneTable PhoneNumber where
  search name (NPT l) = lookup name l
```

Cu acest cod putem căuta în fiecare tabelă informații folosind un API comun:

```
*Main> search "Ion" nameAge
Nothing

*Main> search "Mihai" nameAge
Just 25

*Main> search "Mihai" nameAddress
Just "a random address"

*Main> search "Gabriela" nameAddress
Nothing

*Main> search "Ionela" namePhone
Nothing

*Main> search "Mihai" namePhone
Just 24828542
```

Aici ne-am oprit data trecută. Astăzi ne vom ocupa de modul în care putem obține informații din toate tabelele (vom simula o operație de join). Vom scrie o funcție `getInfo` care ne va întoarce vârsta, adresa și numărul de telefon pentru persoanele care au toate valorile trecute în baza de date (sau `Nothing` altfel). Implementarea la care ne gândim ar fi

```
getInfo1 name =
  case search name nameAge of
    Just age -> case search name nameAddress of
      Just address -> case search name namePhone of
        Just phone -> Just (age, address, phone)
        Nothing -> Nothing
      Nothing -> Nothing
    Nothing -> Nothing
```

Observați efectul de cascadă al testelor: pentru fiecare căutare nouă trebuie să ne deplasăm mai spre dreapta. Din fericire, codul de mai sus poate fi scris și ca

```
getInfo2 name = do
  age <- search name nameAge
  address <- search name nameAddress
  phone <- search name namePhone
  return (age, address, phone)
```

Pare un stil imperativ și la prima vedere testele de `Nothing` lipsesc. De fapt, codul este în continuare pur funcțional doar că aspectul declarativ este mult mai evident: se pune accentul doar pe partea esențială a codului, partea de boilerplate (codul pe care ar trebui să-l scrii în mod repetat înainte de a putea scrie cod util – în cazul nostru codul de testat dacă o valoare este `Nothing` și întors

Nothing înapoi) este ascunsă.

De fapt, mai sus avem mult zahăr sintactic. Codul din getInfo2 este rescris de compilator ca

```
getInfo3 name =
  search name nameAge >>= \age ->
  search name nameAddress >>= \address ->
  search name namePhone >>= \phone ->
  Just (age, address, phone)
```

Observați că de fapt avem de-a face cu o compoziție de funcții similară unei benzi de asamblare. Operatorul >>= ia rezultatul unei funcții și-l trimite funcției următoare. Modul în care am scris codul în TODO este demonstrativ pentru denumirea de *programmable semicolon* oferită operatorului >>=: funcționează ca ; din limbajele imperative doar că are o semantică asociată. Dacă în limbajele imperative ; era doar pentru a separa instrucțiuni, >>= poate încorpora diverse logici în spate. În cazul nostru testează de Nothing și întoarce Nothing dacă este cazul.

De asemenea, observați că return nu are semnificația lui return din C. De fapt, în getInfo2 sau getInfo2 puteți înlocui return cu Just sau viceversa și veți obține exact același comportament.

Testăm întâi codul scris în toate variantele lui

```
*Main> getInfo1 "Mihai"
Just (25,"a random address",24828542)
*Main> getInfo2 "Mihai"
Just (25,"a random address",24828542)
*Main> getInfo3 "Mihai"
Just (25,"a random address",24828542)
*Main> getInfo3 "Ioana"
Nothing
```

La final de articol vom prezenta și partea magică din spate, partea din limbajul de programare care ne permite ca >>= să fie *programmable semicolon*. De fapt, totul se bazează pe o anumită clasă de tipuri, una din setul celor care reprezintă șabloane de programare funcțională.

Vom începe prin a reaminti clasa Functor pe care am prezentat-o în articolul trecut.

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

Dacă mai țineți minte, clasa a fost introdusă pentru a putea folosi fmap – operație similară map – pentru elemente ale altor tipuri. Pentru liste fmap este exact map.

```
Instance Functor [a] where
  fmap = map
```

Este evident că vom putea folosi fmap pentru arbori, stive, grafuri, etc. Practic, putem folosi analogia unui container: fmap aplică o funcție pentru toate elementele dintr-un container și le întoarce

împachetate într-un container de aceeași formă. Dar, îl putem folosi și pentru funcții:

```
*Main Control.Applicative> fmap (+1) (const 3) $ 5
4

*Main Control.Applicative> :t fmap (+1) fst
fmap (+1) fst :: Num b => (b, b1) -> b
*Main Control.Applicative> fmap (+1) fst $ (2, 5)
3

*Main Control.Applicative> :t fmap show fst
fmap show fst :: Show a => (a, b) -> String
*Main Control.Applicative> :t fmap show fst (2,3)
fmap show fst (2,3) :: String
*Main Control.Applicative> fmap show fst (2,3)
"2"
```

Analogia eșuează. Putem privi f din clasa Functor ca pe un context computațional. Operația fmap va modifica acest context. De fapt, dacă ne amintim că funcțiile sunt în forma *curry*, tipul ne spune că fmap ridică o funcție normală la nivelul unui context computațional/container.

Nu orice tip de date suportă o instanță pentru Functor, există 2 legi din teoria categoriilor ce trebuiesc respectate. Nu voi insista asupra lor întrucât este destul de dificil de întâlnit in practică un tip care să nu le respecte.

Mergem la clasa care ne interesează, numită Monad. Tipurile de date listă, Maybe, Either sunt deja înrolate în această clasă.

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

Observați că return este o funcție, nu este un cuvânt cheie al limbajului. Singurul lui scop este să ridice o valoare normală la contextul computațional necesar. Pentru Maybe, return nu este altceva decât constructorul Just.

Cealaltă funcție este mai importantă. Operatorul >>= pronunțat *bind* realizează un lucru pe care nu l-am putea face cu fmap: imaginați-vă că avem o funcție care primește o valoare și întoarce un rezultat într-un context (de exemplu o funcție care primește o valoare și întoarce o listă de valori). Dacă am face fmap cu această funcție vom obține un context de contexte (listă de liste de valori) deci avem nevoie de o operație suplimentară (în cazul listelor, concat). Operatorul >> este un caz particular al lui >>= (deci definit în termenii lui) care folosește doar pentru înlănțuirea efectelor, nu transmite rezultatul unei expresii mai departe.

Și pentru monade există un set de 4 legi ce trebuiesc satisfăcute dar nu vom insista asupra lor. De fapt, cunoscând doar definițiile pentru Functor și Monad și tipurile esențiale din Haskell putem scrie destul de mult cod fără a avea nevoie de mai multe noțiuni din teoria categoriilor. Dar, dacă

sunteti curioși, vă recomand să citiți *Typeclassopedia* al lui Brent Yorgey pentru a vedea ce alte șabloane din programarea funcțională mai pot fi capturate prin intermediul claselor de tipuri.