**Programare Funcţională în Haskell**

Articolul din numărul trecut a realizat o scurtă introducere în limbaj prezentând mai mult isotira și beneficiile lui și mult mai putin elemente de cod propriu-zis. Astăzi, ne apropiem de acest deziderat discutând despre tipurile limbajului.

Un lucru absolut definitoriu pentru limbaj este faptul că Haskell are tipare statică: fiecare expresie are un tip cunoscut de la compilare. În plus, nu există conversii implicite între tipuri similare: programatorul va trebui să facă explicit conversiile în locurile în care acestea sunt necesare.

Dar, să începem lent, cu începutul. Primul lucru pe care-l vom spune este că, spre deosebire de C, Java și alte limbaje similare, tipurile nu sunt o pacoste: descrierea lor este opțională. Puteți scrie foarte mult cod fără a scrie o singură definiție de tip. Compilatorul și interpretorul vor apela la modulul de sinteză de tip pentru a infera tipurile cele mai generale pentru codul scris. Pe de altă parte, este bine să cunoaștem tipurile expresiilor și să programăm folosindu-ne de aceste informații, după cum se va vedea în continuare.

Cel mai simplu tip este cel al expresiilor booleene, `True` și `False`. Atenție, aceste valori se scriu cu literă mare și vom vedea în acest articol de ce. Tipul acestor două valori este `Bool`. Ca o regulă, tipurile în Haskell se scriu cu literă mare. Este o convenție cu o anumită logică pe care o vom vedea imediat.

Mergem acum la tipurile numerice. Pentru început, numerele întregi. Aici avem un lucru interesant. Valoarea `7` poate fi de tip `Int` sau de tip `Integer`. Diferența între ele este de nivel semantic/implementare: tipul `Int` este limitat de registrele și arhitectura sistemului în timp ce tipul `Integer` presupune folosirea bibliotecii pentru numere de înaltă precizie – se vor putea reprezenta numere oricât de mari. Pentru numerele reale, avem tipurile `Float` și `Double`, exact ca în C.

Poate vă întrebați acum cum se face conversia între `Int` și `Integer`. Să zicem că aveți o funcție `f` ce primește un argument de tip `Int` și aveți o expresie `x` de tip `Integer`. Nu veți putea realiza direct apelul `f x` deoarece cele 2 tipuri nu se potrivesc și compilatorul va arunca o eroare. Din fericire, există funcția `toInteger` ce va realiza conversia argumentului la un rezultat de tip `Integer`. Așadar, apelul va fi `f (toInteger x)` sau, dacă vrem să eliminăm ceva paranteze folosind operatorul `$`, vom ajunge la apelul `f $ toInteger x`.

Este foarte probabil ca acum tiparea statică să vi se pară problematică și greu de folosit. Este necesară puțină experiență cu limbajul până când vă veți obișnui cu aceste mici inconveniențe și veți observa de ce sunt ele necesare și cum vă pot scuti de multe erori.

Să mergem mai departe, către ceva tipuri mai complexe. După cum ați citit și în articolul trecut, există tipul listă de elemente de tip `a`, `[a]`. Observați că tipul este scris cu literă mică, avem de fapt o variabilă de tip ce poate fi instanțiată pentru fiecare caz particular. De exemplu, lista `[1,2,3]` este de tipul `[Integer]` (notația pentru această frază fiind `[1,2,3] :: [Integer]`). Revenind la notația tipului listă, `[a]`, observăm imediat și restricția ca toate elementele unei liste să fie de același tip, `a`. În final, tipul standard al șirurilor de caractere, `String`, nu este altceva decât o listă de caractere (`Char`): `[Char]`.

În final, ultimul tip de bază al limbajului este tipul tuplu. Un tuplu de două elemente are tipul

`(a, b)`, un tuplu de 5 elemente are tipul `(a, b, c, d, e)`, etc. Observați imediat că fiecare element al tuplului poate avea un tip diferit. Astfel, putem avea atât `("ana", "mere") :: (String, String)` cât și `("ana", 42) :: (String, Integer)`. Ca un caz particular, există și tuplul cu zero elemente, `()`, având o singură valoare `() :: ()`.

În final, cum orice expresie din Haskell are un tip, rezultă că putem vorbi și de un tip pentru o funcție. Acesta ne dă informații despre ce valori de intrare sunt acceptate și ce fel de valori sunt returnate. În unele cazuri, tipul funcției ne dă și detalii despre ce face funcția, funcționează ca o documentație. De exemplu, o funcție `f :: a -> a` ne zice ca primește un argument de orice tip și întoarce un rezultat de fix același tip. Dacă ne limităm la funcțiile care se termină (excludem cazurile de forma `f x = f x`), nu dau eroare (fără `f x = undefined` sau `f x = error ...`) și nu sunt complicate inutil (excludem și `f x = head [x, f x, f (f x)]`) obținem o singură expresie validă pentru `f`: funcția identitate. Așadar, pornind de la tipul acesteia putem deduce imediat ce semantică are, fără a ne uita în implementare. Desigur, nu putem deduce totul din tipuri, cel puțin nu la nivelul celor prezentate până acum.

Cazul funcțiilor cu mai multe argumente este interesant de studiat. De exemplu, funcția

```
addBothWith2 x y = x + y + 2
```

are tipul `Integer -> Integer -> Integer` (de fapt, tipul real este puțin mai complex dar vom reveni asupra lui spre finalul articolului). Fiecare argument este separat de următorul în semnătura de tip prin `->`. De ce această semnătură? Pentru a captura un aspect interesant al programării în Haskell: putem transmite funcțiilor un număr mai mic de argumente decât este cerut și obținem înapoi o funcție nouă. În teorie se zice că funcțiile în Haskell sunt curry (după numele lui Haskell Curry) și acest lucru este posibil doar pentru că funcțiile sunt valori de prim ordin (nu există nici o diferență între a-i trimite fucției identitate un număr sau o funcție, de exemplu).

Întorcându-ne la funcția `addBothWith2` de mai sus observăm că `addBothWith2 3` are tipul `Integer -> Integer`. Așadar, `Integer -> Integer -> Integer` și `Integer -> (Integer -> Integer)` sunt expresii similare. Pe de altă parte, `(Integer -> Integer) -> Integer` reprezintă semnătura unei funcții ce primește ca argument o functie de la întreg la întreg și întoarce un rezultat. Un exemplu ar putea fi următoarea funcție:

```
applyTo42 f = f 42
```

pe care dacă o apelăm cu `(+1)` vom obține `43` iar dacă o apelăm cu `(addBothWith2 3)` vom obține `47`.

Având toate aceste elemente putem scrie orice program dorim. Doar că dacă ne limităm doar la tipurile prezentate nu vom obține nici un beneficiu de pe urma tipării statice din Haskell, ba chiar vom avea și ceva probleme. De exemplu, există funcții predefinite doar pentru tuplurile cu 2 elemente. Pentru toate celelalte va trebui să scriem noi de mână funcții pentru accesarea și modificarea elementelor componente.

Din fericire, limbajul Haskell ne permite să ne construim tipuri proprii pentru a avea un program mai expresiv, mai declarativ. Le vom vedea pe toate în acest articol.

Pentru început, am zis mai sus că tipul String este de fapt un sinonim pentru tipul [Char]. Este mult mai usor de citit un cod care folosește [String] versus un cod care folosește [[Char]]. Idem, este mult mai comod să lucrați un program care are Vector2D, Point2D, Size fată de un program care folosește (Integer, Integer) pentru toate 3 valorile. În Haskell, putem declara orice sinonim de tip folosind type. De exemplu, tipul String este definit astfel:

```
type String = [Char]
```

Compilatorul lucrează în spate cu tipul original. Doar anumite semnături de tip vor folosi sinonimul. Și programatorul îl poate folosi oriunde în program.

Pentru a construi un tip nou folosim data. Tipurile noi în Haskell se definesc pe baza constructorilor: pur și simplu listăm fiecare constructor împreună cu argumentele necesare lui. De exemplu, următorul cod listează definiția exactă a tipului Bool.

```
data Bool = True | False
```

Tipul are 2 constructori, numiți True și False. De fapt, cei 2 constructori sunt exact cele 2 valori ale tipului. Putem enunța acum în întregime regula legată de capitalizarea atomilor din sintaxa Haskell, la nivelul valorilor (pentru nivelul tipurilor trebuie să mai introducem un concept): toți constructorii unui tip se scriu cu literă mare și numai ei.

Un tip ceva mai complex este tipul Maybe. El ne permite să avem o valoare sau posibilitatea de a semnaliza faptul că funcția a ajuns într-un caz de eroare. Astfel, suntem salvați de la a obtine un null-pointer-exception la runtime: programatorul va trebui să trateze ambele cazuri în funcțiile scrise de el.

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

Observați că tipul este generic: primește ca argument un alt tip sub forma variabilei de tip a. Unul dintre constructori folosește acest tip pentru a împacheta valoarea. Putem avea deci tipul Maybe Int sau tipul Maybe (Maybe String), fiecare cu semantica proprie.

Dezavantajul folosirii tipului Maybe este că în cazul în care funcția eșuează nu se poate salva și motivul eșecului. Din fericire, există și tipul Either, definit ca

```
data Either a b
    = Right a
    | Left b
```

Despre aceste tipuri și cum le vom folosi într-un cod real vom mai discuta în viitor.

Se poate întâmpla ca uneori numărul de câmpuri din constructor să fie foarte mare. Sau, se poate întâmpla ca să avem nevoie să accesăm anumite câmpuri din interiorul tipului. Din fericire, există o notație specială:

```
data Person = P { nume :: String, prenume :: String, varsta :: Int}
```

Ca rezultat, nu numai că se creează tipul de date `Person` și constructorul `P :: String -> String -> Int -> Person` dar avem acces și la funcțiile `nume :: Person -> String`, `prenume :: Person -> String` și `varsta :: Person -> String`.

Ca reprezentare internă, tipurile definite cu data necesită zone de memorie pentru a salva valoarea constructorului și fiecare parametru în parte. Acest lucru este ineficient pentru cazul în care tipul are un singur constructor și acesta are un singur argument. Este cazul în care am putea folosi un sinonim de tip dar am vrea să profităm în totalitate de inferența de tip (pe care o putem obține în întregime doar folosind tipuri cu constructori proprii). Din fericire, Haskell are și a treia metodă de a defini tipuri noi: folosim newtype.

```
newtype State s a = S { runState :: s -> (s, a) }
```

Putem afla tipul oricărei expresii în ghci, folosind `:t expresie`. De exemplu:

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

De unde deducem imediat că `map` va aplica funcția pe o listă și va întoarce lista rezultatelor.

Putem să mai aflăm tipul unei expresii consultând lambdabot pe #haskell (canal de IRC pe Freenode) sau via Hoogle. De fapt, Hoogle ne ajută și în căutarea inversă: putem căuta după tipul aproximativ al unei funcții, să zicem `String -> Int -> Char` și vom ajunge prin pagina de rezultate la `(!!) :: [a] -> Int -> a`, funcția care ne întoarce elementul de pe o anumită poziție din listă.

Ca exercițiu pentru astăzi, vom simula un program de manipulat baze de date. Momentan vor realiza doar o căutare în una din cele 3 tabele ce reprezintă informații despre oameni. Începem prin a defini câteva sinonime de tip, pentru a înțelege codul mai ușor:

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

Definim acum tipurile pentru cele 3 tabele de intrare:

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

Am folosit newtype și tupluri pentru eficiența reprezentării și pentru a avea inferență de tipuri. Partea deriving Show este necesară pentru a putea afișa valorile de aceste tipuri (o vom prezenta în amănunțime data viitoare).

Să construim acum câteva valori pentru cele 3 tabele pe care le folosim:

```
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)]
```

După cum observați, am avea nevoie de o funcție de căutat în listele repsective: căutăm perechea al cărei prim element este un nume dorit și ne interesează al doilea element al perechii. Desigur, am putea să scriem noi o funcție recursivă pentru asta dar este un exercițiu interesant să folosim Hoogle. Dacă am căuta o funcție `[(String, a)] -> String -> Maybe a` nu vom găsi nici un rezultat (am generalizat doar tipul celui de-al doilea element din tuplu). În schimb, dacă vom generaliza ambii parametri și vom căuta `[(a, b)] -> a -> Maybe b` primul rezultat din listă este funcția lookup (ignorați momentan partea `Eq a =>` din semnătură, o vom trata tot data viitoare).

Putem scrie acum imediat funcțiile de căutat în cele 3 tabele:

```
searchNameAge name (NAgT l) = lookup name l
searchNameAddress name (NAdT l) = lookup name l
searchNamePhone name (NPT l) = lookup name l
```

După cum observați, în partea dreaptă a egalului am folosit exact aceeași expresie. Vom vedea data viitoare cum se realizează apelul potrivit, conversia potrivită pentru tipul așteptat și cum putem reduce codul și mai mult, fiind fideli principiului DRY (*don't repeat yourself*).

Pentru astăzi, ne mai rămâne doar să testăm funcțiile scrise. Pentru început, priviți cum compilatorul ne anunță imediat ce folosim o tabelă nepotrivită:

```
*Main> searchNameAge "Ion" nameAddress

<interactive>:21:21:
    Couldn't match expected type `NameAgeTable'
                with actual type `NameAddressTable'
    In the second argument of `searchNameAge', namely `nameAddress'
    In the expression: searchNameAge "Ion" nameAddress
    In an equation for `it': it = searchNameAge "Ion" nameAddress
```

Acum, să căutăm în tabele:

```
*Main> searchNameAge "Ion" nameAge
Nothing
*Main> searchNameAge "Mihai" nameAge
Just 25
*Main> searchNameAddress "Mihai" nameAddress
Just "a random address"
*Main> searchNameAddress "Gabriela" nameAddress
Nothing
*Main> searchNamePhone "Gabriela" namePhone
Nothing
```

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

Vom continua data viitoare aplicația pentru a realiza și operații de tip join pe aceste tabele.

După cum vedeți, folosirea tipurilor ne ajută dar ne și încurcă. Depinde foarte mult de programator să aleagă varianta corectă de design. După ce tipurile au fost puse în scenă, compilatorul devine mai mult sau mai puțin (în funcție de design) un aliat și ne ajută să avem cât mai puține bug-uri la runtime.

Data viitoare ne vom ocupa tot de tipuri dar dintr-o perspectivă mai interesantă: vom vedea cum este implementat polimorfismul și cum putem captura șabloane comune la nivelul tipurilor.