Programare Funcţională în Haskell

Articolul din numărul trecut a realizat o trecere în revistă a tipurilor din Haskell, atât cele preexistente cât şi modalităţile prin care un programator îşi poate crea propriile lui tipuri. Continuăm excursia în tărâmul acestui limbaj prin o scurtă prezentare a unei alte caracteristici a sistemului de tipuri, prin evidenţierea modului prin care se realizează polimorfismul în acest limbaj: clasele de tipuri.

Deşi noţiunile de clasă şi polimorfism duc imedia cu gândul la programarea orientată pe obiecte, ţineţi minte că suntem în alt tărâm. Veţi vedea până la sfârşitul articolului că aproprierea de OOP este destul de forţată, aici avem de-a face cu concepte total diferite.

Vom începe întâi cu funcţia `(+)`. Data trecută i-am forţat tipul la `Integer -> Integer -> Integer` dar este evident că astfel nu vom putea aduna 2 numere reale. Dacă am generaliza tipul la `a -> b -> c` avem 2 probleme: nu putem construi tipul `c` pornind de la `a` şi `b` (dacă era un tip constant puteam alege un constructor nular al acelui tip şi aveam o funcţie constantă – desigur, nu e o implementare corectă pentru `(+)`) şi putem avea şi expresii de forma `True + "ana are mere"`. Restricţionăm semnătura la `a -> a -> a` pentru a rezolva cele 2 probleme mai sus menţionate dar tot vom putea scrie expresia fără sens semantic dar corectă din punct de vedere al compilatorului `True + False`. Ar trebui cumva să restricţionăm tipul valorilor din variabila de tip `a` la tipurile numerice.

Să considerăm un alt exemplu: conversia unei valori la un şir de caractere. În Java avem funcţia `toString` din clasa `Object` pe care o putem suprascrie în orice altă clasă. Numai că, spre deosebire de majoritatea limbajelor, în Haskell funcţiile sunt valori de prim ordin (sunt obiecte ca oricare altele). Dacă luăm în calcul şi faptul că nu se poate scrie o funcţie de conversie a unei funcţii la un şir de caractere (excluzând cazul în care returnăm un şir constant sau un şir ce ţine cont doar de elemente lexicale – `"function at line 42"`) obţinem imediat următoarea problemă: ne trebuie o funcţie `a -> String` dar cu valorile tipurilor din variabila de tip `a` restricţionate la acele tipuri pentru care conversia are sens.

Desigur, avem şi cazul invers, al unei funcţii `String -> a` de conversie de la un şir de caractere la tipul potrivit.

Putem considera şi funcţiile `(==) :: a -> a -> Bool`, `(<) :: a -> a -> Bool`, etc. În fiecare caz, avem nevoie de o restricţie suplimentară asupra tipurilor valorilor variabilelor de tip din semnătură. Restricţii la nivelul tipurilor, nu la nivelul expresiilor tipate din program.

În final, să considerăm un exemplu mai interesant: are sens să definim funcţii similare `map :: (a -> b) -> [a] -> [b]` şi pentru alte tipuri de date de tip container (arbori binari, stive, grafuri, etc.). Am prefera să generalizăm semnătura funcţiei în loc să definim funcţii cu nume diferite pentru fiecare container. Astfel, în loc să avem `mapTree :: (a -> b) -> Tree a -> Tree b` şi `mapStack :: (a -> b) -> Stack a -> Stack b` am prefera să avem `fmap :: (a -> b) -> f a -> f b` unde `f` reprezintă un constructor de tip unar (primeşte un singur tip *propriu*). Desigur, restricţionând `f` la acei constructori de tip ce vor reprezenta un container de valori.

Observaţi că `f` în semnătura `fmap` anterioară nu poate fi un tip propriu (de exemplu, nu poate fi `Bool`). Practic, ajungem să avem nevoie de un sistem de tipuri pentru tipuri. În Haskell, acesta se cheamă sistemul de kinds, fiecare tip are asociat un kind. Tipurile proprii (`Bool`, `Integer`, tipurile funcţii de la tipuri proprii la tipuri proprii – fără variabile de tip) au toate kind-ul `*`, în timp ce tipurile

constructorilor de tip sunt de forma * -> *, * -> * -> *, etc. În GHCi, puteți afla kind-ul unei expresii de tip folosind :k. Vom reveni asupra kind-urilor peste un număr de articole, momentan descrierea de aici ajută la a înțelege eventualele erori de kind ce vor apărea ca urmare a experimentării lucrurilor prezentate în acest articol și pentru a ilustra faptul că inferența de tipuri nu este un subsistem simplu al compilatorului. Dimpotrivă, sunt cazuri în care și sistemul de kinds nu este de ajuns – dar vom reveni asupra acestui aspect.

Întorcându-ne la restricțiile impuse variabilelor de tip, facem un mic detour prin lumea limbajelor orientate obiect: pentru a face ca anumite clase să implementeze un anumit set de operații foloseam moștenire și – în special – interfețe. În lumea OOP, o interfață era un contract suplimentar: dacă o clasă implementează o interfață suntem siguri că toate metodele din interfață au o implementare și pot fi folosite.

Exact aceeași este și ideea din spatele claselor de tipuri din Haskell. O clasă de tip specifică un set de funcții asociate tipului și obligativitatea ca în momentul în care un tip este înrolat în această clasă programatorul/compilatorul să declare definiții pentru fiecare funcție din clasa de tip.

După cum vedeți, este posibil ca definițiile unor funcții să fie date de compilator, nu de programator. Este cazul clauzelor deriving (introduse în articolul anterior) când se generează o implementare implicită dar este și cazul în care o clasă de tip are definiții implicite pentru anumite funcții din aceasta.

Așadar, echivalentul unei clase de tip intr-un limbaj orientat obiecte este mai aproape de o interfață, cât timp se permite definirea de metode implicite și definirea automată a metodelor în cazul unei implementări a interfeței.

Să vedem acum cum clasele de tipuri rezolvă problemele cu care am început acest articol. Pentru tipurile numerice, există clasa `Num` (și o ierarhie de clase construită peste aceasta: `Real`, `Rational`, `Integral`, etc). Definiția clasei este următoarea:

```
class Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
    fromInteger :: Integer -> a

    x – y = x + negate y
    negate x = 0 - x
```

Se observă imediat că metodele `(-)` și `negate` sunt definite una în funcție de cealaltă. Pentru înrolarea unui tip în clasa `Num` vor trebui definite toate metodele mai puțin una din cele 2.

Pentru conversia la șiruri de caractere avem clasa `Show` ce conține funcția `show :: a -> String`. Similar, pentru conversia inversă avem clasa `Read` cu funcția `read :: String -> a`. Pentru testele de egalitate/inegalitate avem clasa `Eq` din care va trebui să implementăm una din `(==)` sau `(/=)`. În fine, pentru a ordona două elemente ale unui tip avem clasa `Ord`

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

Din această clasă ajunge să implementăm metoda compare sau cele 4 teste de inegalitate. Observați că există o restricție: un tip nu poate fi înrolat în clasa Ord fără a fi înrolat și în clasa Eq. Altfel nu ar avea sens operațiile (<=) și (>=).

În final, funcția fmap poate fi aplicată tuturor tipurilor înrolate în clasa Functor. Numele provine dintr-un concept din teoria categoriilor și clasa este mult mai utilă decât pare acum. Vom reveni asupra acestui aspect într-unul din articolele următoare.

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

Cunoscând clasele, să vedem cum înrolăm un tip nou la acestea. Putem folosi clauza deriving pentru a lăsa compilatorul să deducă automat definițiile rezonabile sau vom folosi instance. De exemplu, pentru liste, avem următoarea instanțiere a tipului Functor:

```
instance Functor [] where
   fmap = map
```

Continuăm acum exemplul de data trecută cu cele 3 tabele în care căutăm informații despre persoane. Dacă mai țineți minte, am fost nevoiți să definim 3 funcții diferite de căutare. Respectând șablonul oferit de Functor / fmap am prefera să facem același lucru și aici. Vom folosi câteva extensii ale compilatorului GHC. Le vom activa folosind directiva LANGUAGE (le-am scris câte una pe rând pentru lizibilitate, de regulă se scriu într-un singur comentariu).

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

Ne amintim definițiile de tipuri din articolul trecut:

```
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
```

Observați cum am folosit deriving Show pentru a putea afișa tabela în cazul în care vrem să vedem ce conține aceasta.

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)]
```

Este momentul să definim o clasă pentru căutarea în aceste tabele și apoi să înrolăm cele 3 tipuri la această clasă.

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

Observați că a trebuit să folosim 2 variabile de tip în definiția clasei deoarece avem nevoie de 2 variabile de tip în semnătura funcției search: `t` pentru tabela în care căutăm și `a` pentru rezultatul întors. Deoarece tabela determină tipul rezultatului întors am folosit o dependență funcțională pe care o vedeți între `|` și `where`. Din cauza acestei dependențe a trebuit să activăm extensiile mai sus menționate (cea cu sinonimele deoarece vom folosi sinonime în momentul înrolării tipurilor tabelelor).

Instanțierea este relativ evidentă:

```
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
```

Aparent, am scris mai mult cod decât data trecută și pare a fi mai mult cod repetat. E drept, este un exemplu simplu și nu avem multe metode în clasa noastră. Dar, șablonul de a căuta într-o tabelă după primul câmp este complet capturat de aceasta.

Acum, să căutăm în tabele:

```
*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 "Gabriela" namePhone

Nothing

*Main> search "Mihai" namePhone

Just 24828542
```

Tipul tabelei în care căutăm specifică tipul rezultatului dar interfața de căutare este unică.

Vom continua data viitoare aplicația pentru a realiza și operații de tip join pe aceste tabele, folosind faptul că avem o interfață unificată pentru a căuta în acestea. Vom vedea atunci cum vom putea evita problema unei înlănțuiri de teste pentru `Nothing` / `Just _` ce apare în acest moment dacă încercați să scrieți codul pentru join. Vom porni de la `Functor` și vom extinde ierarhia de clase ce capturează șabloane inspirate din teoria categoriilor.