Chapter 1

Tipuri Algebrice de Date

Haskell este un limbaj static tipizat (fiecare expresie are asociat un tip bine definit încă de la momentul compilării).

Spre deosebire de alte limbaje care încurajează programarea funcțională (de exemplu, Lisp/Scheme), Haskell și celelalte limbaje din familia ML (*meta-Language* – a nu se confunda cu Machine Learning), precum OCaml, Standard ML ș.a. încurajează definirea funcțiilor prin potrivirea de șabloane.

Tipurile algebrice de date sunt tipuri de date care se obțin prin aplicarea a două operații peste tipuri:

1. produs și
2. sumă (reuniune) disjunctă.

1.1 Tipuri enumerative

Cele mai simple tipuri algebrice de date sunt tipurile enumerative:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Tipurile de date, constructorii și clasele de tipuri încep cu literă mare. Funcțiile încep cu literă mică.

Funcțiile peste astfel de tipuri se pot scrie folosind potrivirea de șabloane (pattern-matching):

```
nextDay : Day -> Day
nextDay Mon = Tue
nextDay Tue = Wed
...
nextDay Sun = Mon
```

Detaliu tehnic Dacă veți încerca funcția de mai sus, veți observa o eroare.

```
ghci> nextDay Mon
[eroare]
```

Eroarea apare nu în cursul evaluării expresiei, ci după evaluarea rezultatului, în momentul în care sistemul trebuie să transcrie rezultatul pe ecran. Pentru a afișa o valoare de tip `a`, sistemul folosește mai întâi funcția `show :: a -> String`, care transformă o valoare de tip `a` într-un șir de caractere. Funcția `show` este definită pentru orice tip care e instanță a clasei de tipuri `Show`. Putem forța instanțierea clasei folosind clauza `deriving Show`, ca mai jos:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving Show
```

Tipul `Bool` din biblioteca standard:

```
data MyBool = MyFalse | MyTrue deriving (Show, Eq)
```

Sintaxa de mai sus este folosită pentru a face `MyBool` instanță a claselor `Show` (clasa tipurilor ale căror valori pot fi convertite în `String`) și `Eq` (clasa tipurile ale căror valori pot fi comparate).

1.2 Constructori cu parametri

Fiecare constructor poate primi *parametri*.

Vom ilustra acest aspect folosind funcția `head` pe care am discutat-o ora trecută:

```
cap :: [Int] -> Int
cap [] = ???
cap (x:xs) = x
```

Am dori un tip de date care să conțină toate valorile de tip `Int`, plus o valoare *specială* care să marcheze cazul excepțional.

```
data Rezultat = Nimic | Un Int
```

Constructorul `Nimic` are 0 parametri.

Constructorul `Un` are 1 paramatru (întreg mașină).

```
cap :: [Int] -> Rezultat
cap [] = Nimic
cap (x:xs) = Un x
```

Procesarea valorilor de tip `Rezultat` se poate face prin pattern-matching:

```
contineInt :: Rezultat -> Bool
contineInt Nimic = False
contineInt (Un _) = True
```

Diferența dintre o funcție și un constructor este că funcția procesează argumentele pentru a obține un rezultat. Constructorul lasă argumentele neprocesate, doar "le ține minte" sau "le protejează". Mai târziu, prin potrivirea de șabloane, se pot recupera constructorul și argumentele sale.

Analogie: un constructor este un plic pe care este scris numele constructorului, iar în plic se găsesc argumentele.

```
data Pereche = P Int Int deriving (Show, Eq)

prima : Pereche -> Int
prima (P x y) = x

adoua : Pereche -> Int
adoua (P x y) = y
```

1.3 Tipuri de date recursive

```
data Lista = Vida | Cons Int Lista deriving (Show, Eq)
```

Constructorul `Vida` are 0 argumente.
Constructorul `Cons` are 2 argumente: o valoare de tip `Int` și o valoare de tip `Lista`.

```
lungime :: Lista -> Int
lungime Vida = 0
lungime (Cons _ tail) = 1 + lungime tail

suma :: Lista -> Int
suma Vida = 0
suma (Cons x tail) = x + lungime tail

produs :: Lista -> Int
...

suma :: Lista -> Int
suma Vida = 0
suma (Cons x tail) = x + lungime tail
```

1.4 Tipuri de date parametrice

Nu e DRY să scriem câte un tip de date pentru diferite liste de care am putea avea nevoie:

```
data ListaInteger = VidaInteger | ConsInteger Integer ListaInteger deriving (Show, Eq)
data ListaBool = VidaBool | ConsBool Bool ListaBool deriving (Show, Eq)
[...]
```

Soluție: tipurile parametrizate.

```
data List a = Emp | Con a (List a) deriving (Show, Eq)

ghci> Con 1 (Con 2 (Con 3 Emp)) :: List Int
ghci> Con True (Con False (Con False Emp)) :: List Bool

convert :: [a] -> List a
convert [] = Emp
convert (x:xs) = Con x (convert xs)

convert' :: List a -> [a]
...
```

Definiți tipul *Pereche* parametrizat.
Scrieți funcțiile `head`, `tail`, `length`, `append`, `adaugaLaSfarsit`, `reverse`.
Scrieți funcțiile:

```
appendAll :: List (List a) -> List a

myzip :: List (Pereche a b) -> Pereche (List a) (List b)

myunzip :: (List a) -> (List b) -> List (Pereche a b)
-- ma opresc la cea mai scurta lista
```

1.5 Aplicație: expresii aritmetice

```
data Exp = Const Float | Var String | Suma Exp Exp | Prod Exp Exp deriving (Show, Eq)

deriv :: Exp -> String -> Exp
deriv (Const _) _ = Const 0
deriv z@(Var x) y = if x == y then Const 1 else z
deriv (Suma e1 e2) x = Suma (deriv e1 x) (deriv e2 x)
deriv (Prod e1 e2) x = Suma (Produs e1 (deriv e2 x)) (Produs (deriv e1 x) e2)

simpl :: Exp -> Exp
simpl z@(Const _) = z
simpl z@(Var _) = z
simpl (Suma e1 e2) = let e1' = simpl e1 in
                     let e2' = simpl e2 in
                     case (e1', e2') of
                     | (Const a, Const b) -> Const (a + b)
                     | (Const 0, _) -> e2'
                     | (_, Const 0) -> e1'
                     | _ -> Suma e1' e2'
...
```

Tipurile algebrice de date sunt foarte utile când manipulăm expresii logice, aritmetice, arbori de sintaxă abstractă (limbaje de programare).

Pentru o comparație cu clasele din limbajele orientate obiect, a se vedea "expression problem".

1.6 Tipuri abstracte de date

A nu se confunda "algebraic datatype" cu "abstract data type" (deși au aceeași abreviere: ADT), deoarece sunt concepte diferite.