Haskell: Tipuri de date utilizator

- Data publicării: 18.04.2022
- Data ultimei modificări: 18.04.2022

Obiective

Scopul acestui laborator este introducerea **mecanismului de tipuri** al limbajului Haskell, precum și prezentarea unor modalități de a defini **noi tipuri** de date.

Aspectele urmărite sunt:

- particularități ale tipurilor de date în Haskell
- sinteza de tip
- definirea tipurilor de date utilizator:
    - sinonime de tip
    - tipuri de date enumerate
    - tipuri înregistrare
    - tipuri parametrizate
    - tipuri recursive
    - tipuri izomorfe

Introducere

După cum am discutat în cadrul laboratorului introductiv, limbajul Haskell implementează un **mecanism de tipuri** specific, având la bază tipurile de date algebrice [http://www.haskell.org/haskellwiki/Algebraic_data_type]. Scopul mecanismului este impunerea **corectitudinii** la nivelul programelor, mai exact, garantarea unor proprietăți de corectitudine în funcție de tipurile folosite. Aceasta implică o serie de caracteristici ale limbajului în raport cu sistemul de tipuri.

În primul rând, Haskell este un limbaj **puternic** tipat. Astfel, două tipuri A și B vor fi tratate distinct, conversia între acestea realizându-se **explicit**. De exemplu, în C următoarea secvență de cod:

```
int x = -1;
double y = x;
```

este considerată corectă de către compilator, deși variabilele `x` și `y` au tipuri diferite. În Haskell, secvența echivalentă de cod:

```
x :: Int
x = -1
y :: Double
y = x
```

va genera o eroare de tip, fiind necesară folosirea unor funcții de conversie (de exemplu `fromIntegral`) pentru realizarea „cast”-urilor de la un tip la altul.

Observăm că o consecință a tipării puternice o reprezintă imposibilitatea de a defini liste eterogene. De exemplu următoarea expresie rezultă într-un mesaj de eroare:

```
> :t [1, 'a', True]

<interactive>:1:10: error:
    • Couldn't match expected type ‘Char’ with actual type ‘Bool’
```

```
    • In the expression: True
      In the expression: [1, 'a', True]
```

De asemenea, Haskell este tipat **static**, sau **la compilare**: după cum am observat și în exemplul anterior, programul nu va compila decât dacă programul este lipsit de erori la nivel de tip. Raționamentul este acela că tipurile de date reprezintă principala metodă de **abstractizare** în limbajele de programare, astfel că, dacă semantica programelor este corectă, atunci corectitudinea implementării va decurge din aceasta. Evident, afirmația nu e general valabilă, printre altele datorită faptului că Haskell acceptă implementarea funcțiilor parțiale. De exemplu, în expresia:

```
 > head []

*** Exception: Prelude.head: empty list
```

funcția `head` poate fi aplicată în general pe liste, însă aplicarea ei pe lista vidă va genera o eroare **dinamică** (în timpul rulării programului), deoarece nu este posibilă definirea funcției pentru această valoare.

Stabilirea statică a tipurilor este făcută cu ajutorul unui mecanism de **sinteză de tip**: la compilare sunt verificate tipurile tuturor expresiilor, compilarea terminându-se cu succes doar când acestea corespund. Sinteza este efectuată pe tipuri de date oricât de complexe, astfel că, de exemplu, o expresie `expr` având tipul:

```
expr :: [(a,Int)]
```

va fi verificată în adâncime, de la „rădăcină” (tipul listă) către „frunze” (variabila de tip `a`, tipul `Int`).

În continuare, vom studia construcțiile sintactice Haskell care ne permit definirea tipurilor de date utilizator.

type

Construcția `type` ne permite definirea unui **sinonim** de tip, similar cu `typedef` din C. De exemplu:

```
type Point = (Int, Int)
```

Putem astfel să declarăm o definiție de forma:

```
p :: Point
p = (2, 3)
```

Observăm că Haskell nu face distincția între constructorul perechii `(2,3)` și constructorul `Point`, cele două tipuri fiind identice. Singura restricție este aceea că valorile perechii trebuie să fie de tip `Int`, astfel că expresia:

```
p2 :: Point
p2 = (2.0, 3.0)
```

va genera o eroare de tip, deoarece `Point` este identic cu `(Int,Int)`, iar valorile `2.0`, respectiv `3.0`, au tipuri fracționare.

data

Construcția `data` permite definirea de noi tipuri de date algebrice, având următoarea formă:

```
data NumeTip = Constructor1 | Constructor2 | .. | ConstructorN
```

Observăm distincția între *numele tipului* (denumit și *constructor de tip*), care poate fi folosit în expresii de tip (spre exemplu, `expr:: NumeTip`), și *numele constructorilor* (denumiți și *constructori de date*), acestea fiind folosite în definiții, cum ar fi `expr = Constructor1`. De exemplu:

```
data PointT = PointC Double Double deriving Show
```

definește tipul `PointT` prin constructorul `PointC`, construit pe baza unei perechi de `Double`. Cele două nume sunt **distincte** din punctul de vedere al limbajului, însă pot fi suprapuse. De exemplu, un punct în trei dimensiuni poate fi definit astfel:

```
data Point3D = Point3D Double Double Double deriving Show
```

În Haskell, constructorii de date sunt reprezentați ca funcții. Dacă inspectăm tipul constructorilor definiți anterior, vom obține:

```
> :t PointC
PointC :: Double -> Double -> PointT
> :t Point3D
Point3D :: Double -> Double -> Double -> Point3D
```

De asemenea, putem consulta tipurile constructorilor definiți implicit de către limbaj:

```
> :t (,)
(,) :: a -> b -> (a, b)
> :t []
[] :: [a]
> :t (:)
(:) :: a -> [a] -> [a]
```

Tipuri enumerate

`data` permite declararea de tipuri enumerate, similare cu construcția `enum` din C. De exemplu:

```
data Colour = Red | Green | Blue | Black deriving Show
```

Observăm faptul că această construcție permite pattern matching-ul pe constructorii tipului:

```
nonColour :: Colour -> Bool
nonColour Black = True
nonColour _ = False
```

De asemenea, e util de menționat faptul că sintaxa `|` denotă o sumă algebrică la nivel de tipuri, fiind în acest sens asemănătoare cu construcția `union` din C.

Tipuri înregistrare

Putem redefini tipul anterior `PointT` pentru a arăta după cum urmează:

```
data PointT = PointC
    { px :: Double
    , py :: Double
    } deriving Show
```

Definiția este semantic identică cu cea anterioară, singura diferență fiind asocierea unor **nume** câmpurilor structurii de date. Aceasta duce la definiția implicită a două funcții, `px` și `py`, având următoarea semnătură:

```
> :t px
px :: PointT -> Double
> :t py
py :: PointT -> Double
```

Acestea au rolul de a selecta valorile asociate fiecărui câmp în parte, având deci implementarea implicită:

```
px (PointC x _) = x
py (PointC _ y) = y
```

Numele câmpurilor pot fi folosite și pentru "modificarea" selectivă a câmpurilor unui obiect. De exemplu pentru `p` de tipul `PointT`, următorul cod va crea un nou `PointT` al cărui câmp `px` va avea valoarea `5`, restul câmpurilor având aceleași valori ca pentru `p`.

```
newP = p { px = 5 }
```

Un mecanism util în implementări este alias-ul (@), care numește un pattern, astfel încât să putem accesa prin nume atât componentele unui pattern cât și pe acesta ca întreg. Putem rescrie o porțiune de cod ca aceasta:

```
returnReversed (PointC x y) reversed
  | reversed = PointC y x
  | otherwise = PointC x y
```

cu următorul cod:

```
returnReversed point@(PointC x y) reversed
  | reversed = PointC y x
  | otherwise = point
```

Tipuri parametrizate

Haskell ne permite crearea de tipuri care primesc ca parametru un alt tip. De exemplu tipul de date ''%%Maybe%%'' [http://www.haskell.org/haskellwiki/Maybe] are următoarea definiție:

```
data Maybe a = Just a | Nothing deriving (Show, Eq, Ord)
```

unde `a` este o variabilă de tip. Acesta are doi constructori, `Just` și `Nothing`, tipurile acestora fiind:

```
> :t Just
Just :: a -> Maybe a
> :t Nothing
Nothing :: Maybe a
```

Observăm că valorile de tip `Maybe a` pot fie să încapsuleze o valoare de tipul `a`, fie să nu conțină nimic, în mod similar cu tipul `void` din C. Această structură ne este utilă atunci când lucrăm cu funcții care pot eșua în a întoarce o valoare utilă. De exemplu, putem folosi `Maybe` pentru a reimplementa funcția `head` în așa fel încât să evităm excepțiile dinamice apărute de aplicarea funcției pe lista vidă:

```
maybeHead :: [a] -> Maybe a
maybeHead (x : _) = Just x
maybeHead _ = Nothing
```

Observație: Parametrizarea la nivel de tip poate fi efectuată și în cazul construcțiilor `type` și `newtype` (prezentată mai jos), în mod similar cu `data`.

Tipuri recursive

Haskell permite **recurența** la nivel de tip, mai exact referirea tipului declarat la un moment dat în cadrul propriilor constructori. Astfel, putem defini tipul listă în următorul fel:

```
data List a = Void | Cons a (List a) deriving Show
```

Această construcție este de fapt implicit prezentă în Haskell, ca **zahăr sintactic**:

```
data [a] = [] | a : [a] deriving Show
```

Un alt exemplu este definirea mulțimii numerelor naturale în aritmetica Peano:

```
data Natural = Zero | Succ Natural deriving Show
```

newtype

Construcția `newtype` este similară cu `data`, cu diferența că ne permite crearea unui tip de date cu **un singur** constructor, pe baza altor tipuri de date existente. De exemplu:

```
newtype Celsius = MakeCelsius Float deriving Show
```

sau

```
newtype Celsius = MakeCelsius { getDegrees :: Float } deriving Show
```

folosind sintaxa de tip înregistrare.

Observăm că `newtype`, spre deosebire de `type`, creează un **nou tip**, nu un tip identic. Acest lucru ne este util când dorim să forțăm folosirea unui anumit tip cu o semantică dată. De exemplu atât `Celsius` cât și `Fahrenheit` pot fi reprezentate ca `Float`, însă acestea sunt tipuri de date diferite:

```
newtype Fahrenheit = MakeFahrenheit Float deriving Show

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit (MakeCelsius c) = MakeFahrenheit $ c * 9/5 + 32
```

Diferența principală între `data` și `newtype` este că `newtype` permite crearea de tipuri **izomorfe**: atât `Celsius` cât și `Fahrenheit` sunt tipuri identice cu `Float` din punctul de vedere al structurii, însă folosirea lor în cadrul programului diferă, `Float` având o semantică mai generală (orice număr în virgulă mobilă).

Resurse

- Cheatsheet [https://github.com/cs-pub-ro/PP-laboratoare/blob/master/haskell/tipuri/haskell-cheatsheet-2.pdf]
- Schelet [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/tipuri-schelet.zip]
- Soluții [https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/haskell/tipuri-solutii.zip]

Referințe

- *Algebraic data type [http://www.haskell.org/haskellwiki/Algebraic_data_type]*
- *Haskell Wikibook [http://en.wikibooks.org/wiki/Haskell/Type_declarations]* -Declararea tipurilor
- *Constructor [https://wiki.haskell.org/Constructor]* - Distincție între constructori de tip și constructori de date

- *Learn you a Haskell [http://learnyouahaskell.com/making-our-own-types-and-typeclasses]* - Capitolul "Making your own types"
- *Real World Haskell [http://book.realworldhaskell.org/read/defining-types-streamlining-functions.tml]* - Capitolul "Defining types"