Deklaratív programozás

A logikai és funkcionális programozás alapjai

Csató Lehel

Matematika-Informatika Tanszék
Babeş–Bolyai Tudományegyetem, Kolozsvár

2022 ősz

Az Előadások Témái

- Imperatív/deklaratív programok, Prolog alapok
- Listák, listakezelés
- Aritmetikai műveletek és operátorok
- Vágások használata, negáció
- Gyűjtő-predikátumok Prolog-ban, összegzések
- Vég-rekurzió optimalizálás
- Dinamikus predikátumkezelés; input-output prolog-ban
- Kifejezések dekompozíciója
- Rendezések, kereső algoritmusok
- Funkcionális programozás bevezető, történelem
- A Haskell programnyelv elemei, példaprogramok
- Algebrai típusok, típusosztályok
- Magasabb-rendű függvények, Map-Reduce, rendezések
- Input-output
- *(opc) Algebrai struktúrák 2: Monádok stb*
- Ismétlések és kérdések megvitatása

Adminisztra ... ... trívia

link: Canvas learning management system (LMS)

Régi honlap: `http://www.cs.ubbcluj.ro/~csatol/log_funk`

Vizsga (félév elején) – lásd SYLLABUS!

Labor (40%) + Parciális (25%) + Kollokvium (35%)

Laborgyakorlatok:

1. 2 Prolog feladatcsoport **20%**
2. 2 Haskell feladatcsoport **20%**
3. $\aleph_0$ feladat **első n beküldőnek**

Függvények, mellékhatások I

A függvények definíciója (matematika)

Egy függvény az x bemenetre egy y kimenetet „produkál”:

$$y = f\ x$$

- Egy bemenetre, mindig mindig ugyanaz a **kimenet**.
- Nem minden „informatikus” függvény ilyen; pl. a `readChar` függvénynek nincs bemenete és sok lehetséges kimenete van.
- Függvényeink mellékhatás-mentesek voltak.

Függvények, mellékhatások II

Mellékhatásokat „kezelő” függvények:

- Általában a bemenetért / kimenetért felelős függvények ilyenek.
- Ha egy karaktert be akarunk olvasni:

```
getChar :: IO Char
```

- Ha egy karaktersort meg akarunk jeleníteni:

```
putStrLn :: String -> IO ()
```

Az IO típus/osztálykonstruktor

- Jelzi, hogy az argumentum „kintről” érkezik.
- A `Maybe`-hez hasonlóan, az érték be lett csomagolva;
- A lista egy-egy elemét kivesszük – a `!!` operátorral, illetve pl a `head` függvénnyel.

Impure

Az olyan programokat, melyek „bemeneteket” kezelnek, általában nem tiszta függvénynek nevezzük.

Haskell-ben:

```
getChar ::  IO Char
```

- Az `IO` algebrai típus, mely azt jelzi, hogy a függvény „tartalmaz” mellékhatást;
- Ezt a mellékhatást fel tudjuk dolgozni;

Hasonlóan a `Maybe` típushoz...

Függvények, mellékhatások IV

- Az **IO** típusmódosító bemenet-kimenet műveleteket jelent[13]
- Az **IO** osztály műveleteit is típusosztályok segítségével adjuk meg.
- Ez a típusosztály a monád.

A függvényeket „le kell futtatni”

- Tíz karakter beolvasását felsoroljuk egy listában:

```
take 10 $ repeat getChar
```

  azonban a lista függvényeit végre kell hajtani.
- Erre van a **do** szintaxis.

[13] A Haskell-beli **IO** függvényeket definiál.

A „do” jelölés

A do függvénnyel:

- IO műveleteket helyezünk egymás után;

  `do {a1; a2; ... an}`

- Az `a1; a2; ... an` IO-függvényeket szekvenciálisan végezzük el.

- Példa:

  ```
  do { putStr "Elso resz,"; putStr " masodik resz."}
  ```

  eredménye `"Elso resz, masodik resz."`.

- Egyszerűsített jelölést is használhatunk, ahol egymás alá írjuk a függvényeket (hasonlóan a **where** kulcsszóhoz).

A „do” jelölés II

Az egyszerűsített **do** parancs:

```
do
   action1
   action2
   action3

-- alternatívák

do
   x1 <- action4
   action5
   let x2 = function7
   action7
   return x3

-- fontos az ‘‘indent’’
```

Jellemzők:

- Mindegyik függvény kimenete **IO**-t tartalmaz.
- Például a
  ```
  putChar :: Char -> IO ()
  putStrLn :: String -> IO ()
  ```
- „IO-sító” függvény pl a (az IO egy monád)
  ```
  return :: Monad m => a -> m a
  ```
  A **do** testében: `return :: a -> IO a`
- A balranyíl - „<-” - kicsomagolja az **IO**-ból az értéket;
- A `let v = ...` konstrukció egy függvény értékét teszi elérhetővé;
- A **return** a kilépés a **do** blokk-ból – az **x3** lehet a definiált **x1**... változókat tartalmazó kifejezés is.

Példa:

```haskell
nameDo :: IO ()
nameDo = do
    putStr "First name: "
    fN  <-  getLine
    putStr "Last name: "
    lN  <-  getLine
    let full = fN ++ " " ++ lN
    putStrLn $ "Hello " ++ full
```

```haskell
nameLm :: IO ()
nameLm = putStr "First name: " >>
    getLine >>= \ fN ->
    putStr "Last name: " >>
    getLine >>= \ lN ->
    let full = fN ++ " " ++ lN
    in putStrLn $ "Hello " ++ full
```

- A két függvény ekvivalens;
- A **do** jelölés egyszerűsíti a programozást.
- Az `()` az „üres típus”.

Példa visszatérített értékre:

```haskell
nameRet :: IO String
nameRet  =  do
    putStr "First name: "
    fN  <-  getLine
    putStr "Last name: "
    lN  <-  getLine
    let full = fN ++" "++ lN
    return full
```

```haskell
import System.Random
-- {{ :set -package random }}

newRand = randomIO :: IO Int

randList n
    | n > 0 = randList_ (n,[])
    | otherwise = return []

randList_ (0,xs) = return xs
randList_ (n,xs) = do
    x <- newRand
    randList_ (n-1,x:xs)
```

Típus-aláírások:

```haskell
 newRand  :: IO Int
randList  :: (Ord a, Num a) => a -> IO [Int]
randList_ :: (Eq a, Num a) => (a, [Int]) -> IO [Int]
```