Funkcionális programozás
7. előadás

MÁRTON Gyöngyvér

Sapientia Egyetem, Matematika-Informatika Tanszék
Marosvásárhely, Románia
mgyongyi@ms.sapientia.ro

2024, tavaszi félév

Miről volt szó?

- feladatok, kiíratások listákkal
- Haskell projektek
- rendezések: `sortOn`, összefésülő rendezés (merge sort)
- feladatok:
  - madárnyelv
  - maximum elem és pozíciói
  - válogatások
  - statisztikai számítások
  - Benford törvénye

Miről lesz szó?

- rendezések:
  - beszúró rendezés (insertion sort),
  - gyorsrendezés (quick sort),
- hajtogatások (fold operations)
  - maximum elem pozíciói
  - prímszámok
  - fibonacci számok
  - könyvtárfüggvények implementációja: sum, head, last, map, filter, elem, any,

Beszúró rendezés

A beszúró rendezést (insertion sort) két függvénnyel valósítjuk meg, az első az `ins` függvény, amely beszúr egy elemet egy rendezett listába, a második az `insertionS`, amely az `ins` függvényt alkalmazva meghatározza a rendezett sorrendet.

Adott elem beszúrása rendezett listába:

```
ins :: (Ord a) => a -> [a] -> [a]
ins x [] = [x]
ins x (k : ve)
    | x > k = k : ins x ve
    | otherwise = x : k : ve
```

```
> ins 'h' "abcdeklmno"
"abcdehklmno"
```

```
> ins "racionalis" ["egesz", "komplex", "valos"]
["egesz","komplex","racionalis","valos"]
```

Beszúró rendezés

Az insertionS úgy rendezi a lista elemeit, hogy az első elemet, a `k`-t beszúrja a rendezett `ve` listába, ahol a `ve` lista rendezését rekurzívan az `insertionS` függvény végzi.

```
insertionS :: (Ord a) => [a] -> [a]
insertionS [] = []
insertionS (k : ve) = ins k (insertionS ve)

> insertionS [3,12,7,8,4,5,11]
[3,4,5,7,8,11,12]

> insertionS "acbdeklhmno"
"abcdehklmno"
```

Gyorsrendezés

- a gyorsrendezést (quick sort) halmazkifejezésekkel végezzük
- kiválogatjuk a `k`-nál kisebb elemeket a `kLs`-be, illetve a `k`-nál nagyobb elemeket az `nLs`-be
- mindkét részlistát a `quickS` függvénnyel rekurzívan rendezzük
- a kapott részlistákat összefűzzük, a két rendezett lista közé helyezve a `k` elemet

```
quickS :: (Ord a) => [a] -> [a]
quickS [] = []
quickS (k : ve) = quickS kLista ++ [k] ++ quickS nLista
    where
    kLista = [x | x <- ve, x < k]
    nLista = [x | x <- ve, x >= k]

> quickS [7,5,9,10,13,2,3,11]
[2,3,5,7,9,10,11,13]
```

Hajtogatások (fold operations)

- a *hajtogatásokat*, a *folding* műveleteket végző függvények magasabb rendű függvények
- listaelemeken végeznek kiértékeléseket explicit rekurzió nélkül
- a `foldl` függvény háromparaméteres, az első egy bináris operátor, a második egy kezdőérték, a harmadik pedig egy lista, típusdeklarációja a következő:

```
foldl :: Foldable t => (a -> b -> a) -> a -> t b -> a
```

  - balra asszociatív: balról jobbra haladva dolgozza fel a listaelemeket
  - rendre alkalmazza a bináris operátort úgy, hogy a bal oldali operandusa a `foldl` függvény második paramétere, *jobb oldali operandusa az aktuális listaelem* lesz
  - a második paraméter minden listaelem feldolgozása után felülíródik a bináris művelet eredményével
  - a függvény által meghatározott érték a második argumentumban kiszámolt érték lesz
  - akkor számol, amikor *megy be* a rekurzióba, ezért a rekurzió legalsó szintjén már meg van határozva a végső eredmény.

Hajtogatások (fold operations)

A lista elemeinek összege:

```
> foldl (+) 0 [1..10]
55
```

10 faktoriális:

```
> foldl (*) 1 [1..10]
3628800
```

A következő kiértékelés esetében nem csak az eredményt tüntetjük fel, hanem a számítási sorozatot is:

```
> foldl (-) (-2) [34,6,12,65,8,11,23]
-161
```

```
((((((((-2 - 34) - 6) - 12) - 65) - 8) - 11) - 23)
```

Hajtogatások (fold operations)

A `foldl` függvény implementációja, ahol a mintaillesztés alkalmazása miatt módosítottuk a típusdeklarációt, a harmadik paraméter típusát lista típusra cseréltük:

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl op res [] = res
myFoldl op res (k : ve) = myFoldl op (op res k) ve

> ls = [9,6,12,65,8,11,23]
> myFoldl max (head ls) ls
65
```

Ha a listaelemeket k_1, k_2, k_3, ..., k_n-nel jelöljük, és a `res` a kezdőérték, akkor általánosan a `foldl` hívásai során a következőképpen kerülnek sorra a műveletek, ahol az operátort infix formába írtuk:

$$(\ldots (((res\ op\ k_1)\ op\ k_2)\ op\ k_3)\ \ldots\ op\ k_n).$$

Hajtogatások (fold operations)

Módosítjuk a `myFoldl` függvény kódsorát, kiíratjuk a részeredményeket:

```
myFoldlIr op res [] = return res
myFoldlIr op res (k : ve) = do
    putStr $ show (op res k) ++ " "
    temp <- myFoldlIr op (op res k) ve
    return temp

> ls = [9,6,12,65,8,11,23]
> myFoldlIr max (head ls) ls
9 9 12 65 65 65 65 65
```

Megjegyzések:

- a `do` blokk keretén belül az `<-` operátort használva lehetőség van arra, hogy a `myFoldlIr` függvény által meghatározott értéket lekérjük, amely *nem tiszta* függvény
- az `<-` operátort most másképp használjuk, mint a halmazkifejezések során, a Haskellnek azonban a kontextusból meg tudja állapítani, hogyan járjon el

Hajtogatások (fold operations)

A `foldr` a `foldl` függvény párja, típusdeklarációja a következő:

```
foldr :: Foldable t => (b -> a -> a) -> a -> t b -> a
```

- jobbról asszociatív: a listaelemeket jobbról balra dolgozza fel
- egy bináris operátort alkalmaz egy kezdeti érték és a megfelelő listaelemek között
- a függvény első paramétere a bináris operátor, második paramétere pedig a kezdeti érték lesz
- a megfelelő listaelemeket a harmadik paraméterként megadott listából veszi
- a bináris operátor *bal oldali operandusa az aktuális listaelem*, jobb oldali operandusa pedig a `foldr` függvény második paramétere lesz
- a kezdőérték csak a legutolsó rekurzív híváskor kerül feldolgozásra
- a részértékek akkor kerülnek meghatározásra, amikor a függvény *jön vissza* a rekurzióból
- a végső kifejezés meghatározására csak akkor kerülhet sor, amikor minden rekurzív függvényhívás kiértékelődött

Hajtogatások (fold operations)

A lista elemeinek összege:

```
> foldr (+) 0 [1..10]
55
```

Az első 5 páros szám szorzata:

```
> foldr (*) 1 [2,4..10]
3840
```

A következő kiértékelés esetében nem csak az eredményt tüntetjük fel, hanem a számítási sorozatot is:

```
> foldr (-) (-2) [34,6,12,65,8,11,23]
-3
```

```
34 - (6 - (12 - (65 - (8 - (11 - (23 - (-2)))))))
```

Hajtogatások (fold operations)

A `foldr` függvény implementációja, ahol a mintaillesztés alkalmazása miatt módosítottuk a típusdeklarációt, a harmadik paraméter típusát lista típusra cseréltük:

```
myFoldr :: (b -> a -> a) -> a -> [b] -> a
myFoldr op res [] = res
myFoldr op res (k : ve) = op k $ myFoldr op res ve

> ls = [9,6,12,65,8,11,23]
> myFoldr max (head ls) ls
65
```

Ha a listaelemeket k_1, k_2, k_3,..., k_n-nel jelöljük, és a `res` a kezdőérték, akkor általános esetben a `foldr` a következőképpen értékeli ki a műveleteket:

$$k_1 \ op \ (k_2 \ op \ (k_3 \ op \ (\ldots \ (k_n \ op \ res) \ \ldots))).$$

Hajtogatások (fold operations)

Módosítjuk a `myFoldr` függvény kódsorát, kiíratjuk a részeredményeket:

```
myFoldrIr op res [] = return res
myFoldrIr op res (k : ve) = do
    temp <- myFoldrIr op res ve
    putStr $ show temp ++ " "
    return $ op k temp
```

A következő lekérdezések eredményeit hasonlítsunk össze a `myFoldlIr` hasonló paraméterezésével kapott eredményeivel:

```
> myFoldrIr (+) 0 [1..10]
0 10 19 27 34 40 45 49 52 54 55

> myFoldrIr (*) 1 [2,4..10]
1 10 80 480 1920 3840

> ls = [9,6,12,65,8,11,23]
> myFoldrIr max (head ls) ls
9 23 23 23 65 65 65 65
```

Hajtogatások (fold operations)

A lusta kiértékelési stratégia miatt a `foldl` hatékonysága nem elfogadható, egy 1 millió elemszámú lista elemei összegének a meghatározása több másodpercig is eltarthat:

```
> :set +s
> foldl (+) 0 [1..10000000]
50000005000000
(3.15 secs, 1,612,359,432 bytes)
```

Valamivel jobb időt kapunk, ha a `foldr` változattal dolgozunk:

```
> foldr (+) 0 [1..10000000]
50000005000000
(2.32 secs, 1,615,360,800 bytes)
```

- A fenti hatékonysági problémák kiküszöbölésére a Haskell több `foldl` implementációt vezet be, ezekben szigorú (strict) kiértékelési stratégiát tesz lehetővé: `foldl'`, `foldl1`, `foldl1'`
- a `foldl'`, illetve `foldl1'` a `Data.List`-ben találhatók
- a `foldl1`, illetve `foldl1'` változatok esetében nem kell megadni kezdőértéket, ez mindig a lista első eleme lesz

Hajtogatások (fold operations)

Összehasonlításképpen időket mérhetünk:

```
> import Data.List (foldl', foldl1')

> foldl' (+) 0 [1..10000000]
50000005000000
(0.23 secs, 880,062,880 bytes)

> foldl1 (+) [1..10000000]
50000005000000
(2.75 secs, 1,612,359,336 bytes)

> foldl1' (+) [1..10000000]
50000005000000
(0.43 secs, 880,059,088 bytes)
```

Következésképpen a `foldl` helyett mindig a `foldl'`, vagy amikor lehetséges a `foldl1'` változatokat használjuk.

Hajtogatások (fold operations)

1. feladat

Írjunk egy Haskell-függvényt, amely a `foldl'`-t alkalmazva meghatározza a paraméterként megadott lista legnagyobb elemét, illetve a legnagyobb elem előfordulási pozícióit.

```
import Data.List (foldl')
myMaximum :: (Num b, Ord a) => [a] -> (a, [b])
myMaximum ls = (max, mLs)
    where
    (mLs, _, max) = foldl' op res ls
    res = ([], 0, head ls)
    op tRes k
        | k == m = (p : pLs, p + 1, m)
        | k < m = (pLs, p + 1, m)
        | k > m = ([p], p + 1, k)
            where
            (pLs, p, m) = tRes
```

```
> myMaximum [3, 5, 6, 10, 3, 10, 7, 6, 10, 4, -10, 10]
(10,[11,8,5,3])
```

Megjegyzések

- a `foldl'` függvény második paraméterének típusa egy háromelemű tuple
- a meghívásra kerülő bináris operátor egyik operandusa egy háromelemű tuple a másik az aktuális listaelem
- a háromelemű tuple elemei rendre a következő értékeket jelölik: a maximum elem pozíciói, az aktuális pozíció, illetve az aktuális maximum elem

Hajtogatások (fold operations)

A következő myMaximum_ függvény is a legnagyobb elemet, illetve a legnagyobb elem pozícióit határozza meg, csak explicit rekurziót használ:

```
myMaximum_ :: (Num b, Ord a) => [a] -> (a, [b])
myMaximum_ ls = (max, mLs)
    where
    (mLs, _, max) = myMaxAux ([], 0, head ls) ls
    myMaxAux tRes [] = tRes
    myMaxAux tRes (k : ve)
        | k == m = myMaxAux (p : pLs, p + 1, m) ve
        | k < m = myMaxAux (pLs, p + 1, m) ve
        | k > m = myMaxAux ([p], p + 1, k) ve
            where
            (pLs, p, m) = tRes
```

Hajtogatások (fold operations)

2. feladat

Írjunk egy Haskell-függvényt, amely megvizsgálja, hogy egy adott szám prímszám-e vagy sem, majd alkalmazzuk a függvényt a prímszámok listájának kigenerálására. Az implementáció során alkalmazzuk a `foldr` függvényt.

```
primTeszt_ :: Integer -> Bool
primTeszt_ n = n > 1 && foldr op True [3,5..]
    where
    op k resB = k * k > n || (mod n k /= 0 && resB)

> primTeszt_ 1789
True
```

Megjegyzések:

- a `foldr` harmadik paramétere a páratlan számok végtelen listája lesz, de ez nem eredményez végtelen számítási folyamatot, mert az `op`-ben a `||` operátort használtuk
- amikor a `k * k > n` feltétel eredménye `True` lesz, nincs már szükség a `||` jobb oldalán álló kifejezés kiértékelésére
- az oszthatóságok vizsgálata a `k-2, k-4` stb. értékekkel kezdődik (visszafelé haladva a páratlan számok listájában)

Hajtogatások (fold operations)

3. feladat

Írjunk egy Haskell-függvényt, amely kigenerálja az első n prímszám listáját. Az implementáció során alkalmazzuk a `foldr` függvényt.

A prímszámok listáját a `primLista` függvény generálja ki, amely a hatékonyabb `primTeszt`-el végzi a prímek tesztelését:

```
primTeszt :: Integer -> Bool
primTeszt n = n > 1 && foldr op True primLista
    where
    op k resB = k * k > n || (mod n k /= 0 && resB)

primLista :: [Integer]
primLista = 2 : filter primTeszt [3,5..]
```

```
> take 10 primLista
[2,3,5,7,11,13,17,19,23,29]
> last $ take 10000 primLista
104729
(1.61 secs, 253,024,968 bytes)
```

Hajtogatások (fold operations)

A következőkben a hajtógató függvények segítségével számos, korábban már megadott függvény implementációját mutatjuk be.

Egy lista elemeinek összege:

```
mySumL :: (Foldable t, Num a) => t a -> a
mySumL = foldl (\ res k -> res + k) 0

mySumR :: (Foldable t, Num a) => t a -> a
mySumR = foldr (\ k res -> res + k) 0

> mySumL [1,2,3,4,5]
15
```

Hajtogatások (fold operations)

A `head` és `last` függvények implementációi:

```
myHead1 :: [a] -> a
myHead1 = foldr (\ k res -> k) undefined

myLast1 :: [a] -> a
myLast1 = foldl (\ res k -> k) undefined
```

Megjegyzések:

- az `undefined` egyik kódsorban sem kerül kiértékelésre
- az `undefined` használata azért szükséges, mert konkrét érték megadása esetében korlátoztuk volna a bemeneti lista típusát

Hajtogatások (fold operations)

A `head` és `last` függvények következő implementációinál nem kell megadunk kezdőértéket:

```
myHead2 :: [a] -> a
myHead2 = foldr1 (\ k res -> k)

myLast2 :: [a] -> a
myLast2 = foldl1' (\ res k -> k)

> myHead2 "Lohavasi-vizeses"
'L'

> myLast2 [("lohavasi", 80), ("durrogo", 25)]
("durrogo", 25)
```

Hajtogatások (fold operations)

A `map` implementációi, ahol a `foldl'`-vel megadott kód kevésbé hatékony a `++` operátor alkalmazása miatt.

```
myMapL :: Foldable t => (a -> b) -> t a -> [b]
myMapL fg = foldl' (\ resLs k -> resLs ++ [fg k]) []

myMapR :: Foldable t => (a -> b) -> t a -> [b]
myMapR fg = foldr (\ k resLs -> fg k : resLs) []

> myMapL (\ x -> x * x) [1..10]
[1,4,9,16,25,36,49,64,81,100]
```

Hajtogatások (fold operations)

A `filter` implementációi, ahol a `foldl'`-vel megadott kód kevésbé hatékony a ++ operátor alkalmazása miatt

```
myFilterL :: Foldable t => (a -> Bool) -> t a -> [a]
myFilterL fg = foldl' op []
    where
    op resLs k = if fg k then resLs ++ [k] else resLs

myFilterR :: Foldable t => (a -> Bool) -> t a -> [a]
myFilterR fg = foldr op []
    where
    op k resLs = if fg k then k : resLs else resLs

> myFilterL ( x -> mod x 3 /= 0) [1..10]
[1,2,4,5,7,8,10]
```

Hajtogatások (fold operations)

Az `elem` implementációi:

```
myElemL :: (Foldable t, Eq a) => a -> t a -> Bool
myElemL x = foldl' op False
    where
     op resB k = (x == k) || resB

myElemR :: (Foldable t, Eq a) => a -> t a -> Bool
myElemR x = foldr op False
    where
     op k resB = (x == k) || resB

> myElemL 'o' "havasrekettyei vizeses"
False
> myElemR 'a' "havasrekettyei vizeses"
True
```

Hajtogatások (fold operations)

Az any függvény implementációi:

```
myAnyL :: Foldable t => (a -> Bool) -> t a -> Bool
myAnyL fg = foldl' op False
    where
    op resB k = fg k || resB

myAnyR :: Foldable t => (a -> Bool) -> t a -> Bool
myAnyR fg = foldr op False
    where
    op k resB = fg k || resB

> myAny even [1, 4.. 10]
True

> myAny even [1, 3.. 10]
False
```

Hajtogatások (fold operations)

4. feladat

Írjunk egy Haskell-függvényt, amely fold műveletet alkalmazva meghatározza az ***n*****-*****ik*** *Fibonacci-számot.*

```
fibonacciN1 :: Integer -> Integer
fibonacciN1 n = fibonacciAux (0, 1) n
    where
    fibonacciAux (a, b) 0 = a
    fibonacciAux (a, b)  n =  fibonacciAux (b, a + b) (n-1)

fibonacciN2 :: Integer -> Integer
fibonacciN2 n = fst $ foldl op (0,1) [1..n]
    where
        op (a, b) k = (b, a + b)
```

Hajtogatások (fold operations)

5. feladat

Írjunk egy Haskell-függvényt, amely fold műveletet alkalmazva meghatározza egy listába az első n Fibonacci-számot.

```
fibonacciLs1 :: Int -> [Integer]
fibonacciLs1 n = take n $ fibonacciAux (0, 1)
    where
    fibonacciAux (a, b) =  a : fibonacciAux (b, a + b)


fibonacciLs2 :: Int -> [Integer]
fibonacciLs2 n = myFst $ foldl op ([],0,1) [1..n]
    where
        op (ls, a, b) k = (a : ls, b, a + b)

myFst (t1, t2, t3) = t1
```