FUNKCIONÁLIS PROGRAMOZÁS ELŐADÁS JEGYZET

Szerkesztette: *Balogh Tamás*

2013. május 30.

Ha hibát találsz, kérlek jelezd a *info@baloghtamas.hu* e-mail címen!

Adatok:

Név: Horváth Zoltán

E-mail: hz@inf.elte.hu

Webolal http://www.people.inf.elte.hu/fp

Követelmények: Hetente kisbeadandó, nagyobb beadandó, géptermi zh, vizsga

1. Előadás

Funkcionális programozás

A funkcionális programozás: függvény hívások.
A programok: függvények kompoziciója.
n faktoriális $\Rightarrow$ $[1..n]$ sorozat elemeinek szorzata. `fact` függvény $\Rightarrow$ `productum` függvény egy sorozat elemeire.

Clean

```
fact n = prod [1..n]
```

Haskell

```
fact n = product [1..n]
```

Előre definitált függvényhalmaz

Clean:
StdEnv

Haskell:
Prelude

Példa

Növelés: `inc x = x + 1`
Négyzet függvény: `square x = x * x`
`squareinc x = square (inc x) = square ∘ inc x`

Mohó algoritmus

```
squareinc 7
square(inc 7)
square(7+1)
square 8
8 * 8
64
```

Lusta algoritmus

```
squareinc 7 --ez van
square(inct 7)
(inc 7) * (inc 7)
8 * (inc 7)
8 * 8
64
```

Normálforma: További helyettesítés már nem végezhető el. Pl.: fent a 64.

Függvény definíciók

`f x =f x` – függvény definíció
`g x y =y` – függvény definíció
`g 1 2` $\Rightarrow$ 2
`g (f 3)5` $\Rightarrow$ 5 Ez csak lusta kiértékeléssel kapható meg, mohóval megakadna az `f 3` kiszámolásánál.

2. Előadás

Modern funkcionális programozási nyelvek jellemzése

- Nincs előző értéket megsemmisítő értékadás
- Szigorúan típusos, típuslevezetés, polimorfizmus, absztrakt és algebrai típusok
- Hivatkozási helyfüggetlenség
- Magasabb rendű függvények(az argumentum vagy érték is lehet függvény)
- Rekurzió
- Lusta kiértékelés a mohóság vizsgálatával
- Curry féle módszer: ∀ függvénynek egy paramétere van

Zermelo-Fraenkel halmazkifejezések: Lista generálás szűrőfeltétellel(opcionális)

Clean

```
{x * x \\ x <- [1..] | odd(x)}
```

Haskell

```
[x * x | x <- [1..], odd x]
```

3. Előadás

8 királynő

Haskell

```
queens 0 = [[]]
queens n = [ q:b
| b <- b queens (n - 1), q <- [0..7], safe q b]
safe q b
= and [not (checks q b i)
| i <- [0..(length b) - 1)]
checks q b i
= q == b !! i || abs (q - b !! i) == i + 1
main = print (length (queens 8), queens 8)
```

Listák

`[], [1, 18, 29], [1..10], [1..]`
`[1,2,3]== (1:(2:(3:[])))` – : operátor hozzá fűz egy elemet a meglévő lista elejéhez, vagyis ez építi fel a listát.

Mintaillesztés

Clean

```
hd [x:xs] = x
tl [x:xs] = xs
fac 0 = 1
fac n | n > 0
= n * fac (n - 1)
sum [] = 0
sum [x:xs]
= x + sum xs
length [] = 0
length [_:xs]
= 1 + length xs
```

Haskell

```
hd (x:xs) = x
tl (x:xs) = xs
fac 0 = 1
fac n | n > 0
= n * fac (n - 1)
sum [] = 0
sum (x:xs)
= x + sum xs
length [] = 0
length (_:xs)
= 1 + length xs
```

4. Előadás

Típus

Típusellenőrzés

Clean

```
1 + True
//Type error: argment 2 of + cannot unify demanded type Int with Bool

length 3
//argument 1 of length cannot unify demadned type (a b) | length a with
   Int
```

Típusdefiníciók

Clean:
Alaptípusok: Int, Real, Bool, Char

Haskell:
Alaptípusok: Int, Integer, Float, Double, Bool, Char

```
start :: Int
start = 3 +4

x ::[Int]
x = [1,2,3]

y :: [Bool]
y = [True, True, False]
```

```
start :: Int
start = 3 +4

x ::[Int]
x = [1,2,3]

y :: [Bool]
y = [True, True, False]
```

```
z :: [[Int]]
z = [[1,2,3], [1,2]]

sum :: [Int] -> Int
sqrt :: Real -> Real
```

```
z :: [[Int]]
z = [[1,2,3], [1,2]]

sum :: Num a => [a] -> a
sqrt :: Floatin a => a -> a
```

5. Előadás

Osztályok

A lstinlinedouble származtatott függvény. Nem kell, és nem is szabad az osztály kulcsszót használni, és példányosítani. ⋮

Szinonímák

Globális konstansok

Egyszer értékelődnek ki, futási időben, és újrafelhasználhatók. Növekszik általuk a memóriaigény, viszont a futási idő csökkenthető. Pl.:

Clean

```
smallods =: [1, 3 .. 10000]
```

Haskell

```
smallods = [1, 3 .. 10000]
```

Típusszinonímák

fordítási időben cserélődnek

Makrók

Fordítási időben történik a kiértékelés.

Clean

```
black :== 1
white :== 0
```

Haskell

```
black = 1
white = 0
```

Így tehát a következő kifeejezés nem hibás: `black + 1 =2`.

Magasabbrendű listafüggvények

Magasabbrendű függvény: Olyan függvény, amelynek valamelyik paramétere függvény, vagy az eredménye függvény.

Filter

`filter` - adott tulajdonságot teljesítő elemek leválogatása.

Clean

```
filter::(a->Bool)[a]->[a] //típus definíció: polimorf fv., kétváltozós,
   ahol az 1. változó egy fv.(a-ból bool-ba képező)
filter p []=      []
filter p [x:xs]
  |p x =            [x:filterpxs]
  |otherwise =    filter p xs

even x = x mod 2 == 0
odd = not o even  // oddx = not (even x)

evens = filter even [0..]
```

Haskell

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  |px = x:filter p xs
  |otherwise = filter p xs

even x = x `mod` 2 == 0
odd = not.even

evens = filter even [0..]
```

Példa: `filter even [3,2,2,1]` → $[2, 2]$
Példa a kód alapján: `filter even [2 :1]` → `[2 : filter even [1 :[]] ]` → `[ 2 : filter even []]` → `[2 : [] ]`

Map

Elemenként alkalmazza a paraméterül kapott függvényt. Példa: `map inc [2,1,7]` → $[3, 2, 8]$

Clean

```
map :: (a -> b) [a] -> [b]
map f []        = []
map f [x:xs]  = [ f x : map f xs]

odds = map inc evens
```

Haskell

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs) = f x : map f xs
odds = map (+1) evens
```

Foldr

Elemenkénti felhasználás általános változata. Pl. a `sum` egy speciális esete. r betű jelzi, hogy jobbról értékelődik ki.
Pl.: `foldr (+)[3,2,2,1]` $\rightarrow 1 + (2 + (3 + 0))$.

Clean

```
foldr :: (.a -> .(.b -> .b)) .b ![.a] -> .b

foldr f e []         = e
foldr f e [x:xs] = f x (foldr f e xs)

sum = foldr (+) 0 //sum xs = dolr (+) 0 xs

and = foldr (&&) True
```

Haskell

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []       = e
foldr f e (x:xs)   = f x (foldr f e xs)

sum = foldr (+) 0 -- sum xs = foldr (+) 0 xs
and = foldr (&&) True
```

6. Előadás

Foldr folyt.

Miért ilyen a típusleírása? Megfigyelhető, hogy 2 féle típusváltozót tartalmaz. Lehet különböző típus is. Pl.: `Integer, Double` q A binér operátornak két féle operandust adhatunk, de az eredménye a 2. operandus típusa lesz.
Clean-ben a típusváltozók elötti .-ok azt jelzik, hogy polimorf függvények. A . típusannotációváltozó. A ! jelzi, hogy mohó kiértékelésű az adott paraméter.

takeWhile/dropWhile

`takeWhile` - Elemek megtartása, amíg p teljesül.
`dropWhile` - Elemek eldobása, amig p teljesül.
Pl.: `takeWhile even [2,4,5,6]` $\rightarrow [2, 4]$

takeWhile - Clean

```
takeWhile p [] = []
takeWhile p [x:xs]
    | p x          = [x : takeWhile p xs]
    | otherwise    = []
```

takeWhile - Haskell

```
takeWhile p [] = []
takeWhile p (x:xs)
  | p x          = x : takeWhile p xs
  | otherwise    = []
```

Iterálás

f iterálása amíg p nem teljesül.

Clean

```
until :: (a -> Bool) (a -> a) a -> a
until p f x
  | p x          = x
  |otherwise     = until p f (f x)
```

Példa: négyzetgyök számítása Newton-iterációval

```
sqrtn :: Real -> Real
sqrtn x = until goodEnough improve 1.0
  where
    improve y = (y + x / y) / 2.0
    goodEnough y = (y * y) ~=~ x
    (~x~) a b = abs (a - b) < 0.000001
```

Listák

Mivel a listában csak egy féle elemet tartalmazhat, így ha az 1. elem `int`, akkor az összes listaelem `int` típusú.
Ez csak szigorúan típusos nyelvekben van így, például Erlangban nem.
`3*x` szintén `int` kell hogy legyen, hiszem az 1. operandus `int`.

9. Előadás

Műveletek listákkal

Flatten

Listák listájából lesz egy lista. Ezt látni abból, hogy az `x`-et nem az `:` operátorral fűzzük a rekurzívan meghívott `flatten` függvényhez, hanem a `++` operátorral.

Clean

```
flatten [x:xs] =  x ++ flatten xs
flatten [] =  []
```

isMember függvény

Második paramétere egy `[a]` típus lista. Jobb oldalon szerepel: `x == e`, és `x` eleme a listának, ebből adódik, hogy az első paraméterének típusa megegyezik a második paraméterben szereplő listában szereplő elemek típusával. Az eredmény logikai típus, hiszen a második sorában az eredmény értéke `False`.

Clean

```
flatten [x:xs] =  x ++ flatten xs
flatten [] =  []
```

Típusosztályok használata

10. Előadás

Listing.

10. Gyakorlat

Magasabbrendű függvények

Map függvény

Megjegyzés: A kapott lista hossza és a visszaadott lista hossza megegyezik.

map függvény

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Filter függvény

filter függvény

```
filter p [] = []
filter p (x:xs)
  | p x == True = x: filter p xs
  | p x == False = filter p xs
```

Filter függvény

filter függvény

```
filter p [] = []
filter p (x:xs)
  | p x == True = x: filter p xs
  | p x == False = filter p xs
```

Count függvény a filter segítségével

count függvény

```
count p xs = length (filter p xs)
```

Iterate

iterate

```
iterate f x = x : iterate f (f x)
```

All

all

```
all p [] = True
all p (x:xs)
  | p x == True = True && all p xs
  | p x == False = False
```

Any

any

```
any p [] = False
any p (x:xs)
  | p x == True = True
  | p x == False = any p xs
```

zipWith függvény

zipWith függvény

```
zipWith f (x:xs) (y:ys) = f x y :  zipWith f xs ys
zipWith f _ _ = []
```

differences függvény

differences függvény

```
differences xs = zipWith (-) (drop 1 xs) xs
```

Forrás

- ELTE IK programtervezői informatikus szak 2013 tavaszi féléves Funkcionális programozás előadás alapján írt órai jegyzetem.