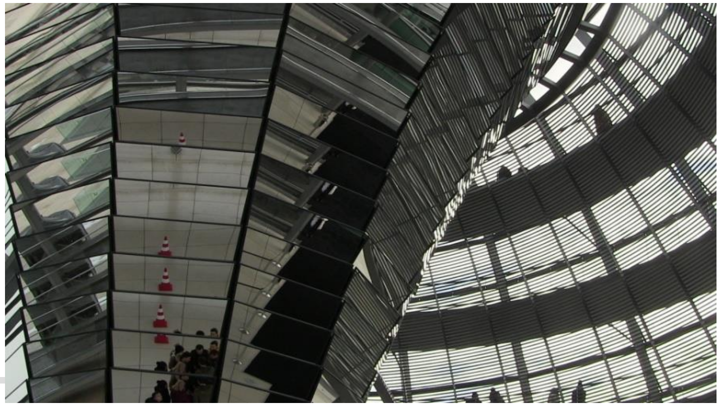

Funkcie a funkcionály

referečná transparentosť

Peter Borovanský

I-18

http://dai.fmph.uniba.sk/courses/FPRO/

Funkcia (predikát) argumentom

Haskell
A Purely Functional Language
featuring static typing, higher-order functions, polymorphism, type classes and monadic effects

- zober zo zoznamu tie prvky, ktoré spĺňajú podmienku (test)
  Booleovská podmienka príde ako argument funkcie a má typ (a -> Bool):

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

> filter even [1..10]
[2,4,6,8,10]

alternatívna definícia:

```
filter p []       = []
filter p (x:xs)   = if p x then x:(filter p xs) else filter p xs
```

vlastnosti (zväčša úplne zrejmé):

- filter True xs = xs ... [x | x <- xs, True] = [x | x <- xs] = xs
- filter False xs = [] ... [x | x <- xs, False] = [ ]
- filter p1 (filter p2 xs) = filter (p1 && p2) xs
- (filter p1 xs) ++ (filter p2 xs) = filter (p1 || p2) xs

filter p []	= []
filter p (x:xs)	= if p x then x:(filter p xs) else filter p xs

Dôkaz

filter p1 (filter p2 xs) = filter (p1 && p2) xs

Indukcia vzhľadom na parameter xs

- []
  L.S. = filter p1 (filter p2 []) = filter p1 [] = [] = filter (p1 && p2) [] = P.S.
- (x:xs)
  L.S. = filter p1 (filter p2 (x:xs)) = ... definícia
  filter p1 (if p2 x then x:(filter p2 xs) else filter p2 xs) = ... if-then-else
  if p2 x then filter p1 (x:(filter p2 xs)) else filter p1 (filter p2 xs) = ... indukcia
  if p2 x then filter p1 (x:(filter p2 xs)) else filter (p1 && p2) xs = ... definícia
  if p2 x then
  if p1 x then x:(filter p1 (filter p2 xs)) else filter p1 (filter p2 xs)
  else filter (p1 && p2) xs = ... 2 x indukcia
  if p2 x then
  if p1 x then x:(filter (p1 && p2) xs) else filter (p1 && p2) xs
  else filter (p1 && p2) xs =

filter p []		= []
filter p (x:xs)	= if p x then x:(filter p xs) else filter p xs

Dôkaz

filter p1 (filter p2 xs) = filter (p1 && p2) xs

<u>if p2 x then</u>

<u>if p1 x then</u> x:(filter (p1 && p2) xs) else filter (p1 && p2) xs

else filter (p1 && p2) xs = ... požívame vlastnosť if-then-else

```
if A then                    if A && B then C
  if B then C                else D
     else D
else D
```

if (p1 && p2) x then x:(filter (p1 && p2) xs) else filter (p1 && p2) xs = ... def.

filter (p1 && p2) (x:xs) = P.S.

Funkcia argumentom map

- funktor, ktorý aplikuje funkciu (1.argument) na všetky prvy zoznamu

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

- Príklady:

```
map (+1) [1,2,3,4,5]                      = [2,3,4,5,6]
map odd  [1,2,3,4,5]                      = [True,False,True,False,True]
and (map odd  [1,2,3,4,5])                = False

map head [ [1,0,0], [2,1,0], [3,0,1] ]    = [1, 2, 3]
map tail [ [1,0,0], [2,1,0], [3,0,1] ]    = [ [0,0], [1,0], [0,1] ]
map (0:) [[1],[2],[3]]                    = [[0,1],[0,2],[0,3]]
```

Vlastnosti map

- map id xs = xs ☑ map id = id
- map (f.g) xs = map f (map g xs) ☑ map f . map g = map (f.g)
- head (map f xs) = f (head xs) ☑ head . map f = f . head
- tail (map f xs) = map f (tail xs) ☑ tail . map f = map f . tail
- map f (xs++ys) = map f xs++map f ys ☑
- length (map f xs) = length xs ☑ length . map f = length
- map f (reverse xs) = reverse (map f xs) ☑ map f.reverse=reverse.map f
- sort (map f xs) = map f (sort xs) ? sort . map f = map f . sort
- map f (concat xss) = concat (map (map f) xss) ☑

map f . concat = concat . map (map f)

```
concat              :: [[a]] -> [a]
concat []           = []
concat (xs:xss)     = xs ++ concat xss
concat [[1], [2,3], [4,5,6],[]] = [1,2,3,4,5,6]
```

Vlastnosti map, filter

Na zamyslenie:

- filter p (map f xs) = ??? (filter (p.f) xs)
- filter p (map f xs) = map f (filter (p.f) xs)
- filter p . map f = map f . filter (p.f)

filter p (map f xs) = filter p [f x | x<-xs]
= [y | y <- [f x | x<-xs], p y]
= [f x | x<-xs, p (f x)]
= map f [x | x<-xs, p (f x)]
= map f (filter (p.f))

Quíz - prémia

nájdite pravdivé a zdôvodnite

- map f . take n = take n . map f
- map f . reverse = reverse . map f
- map f . filter p = map fst . filter snd . map (fork (f,p))
  where fork :: (a->b, a->c) -> a -> (b,c)
  fork (f,g) x = (f x, g x)
- filter (p . g)= map (inverzna_g) . filter p . map g   ak inverzna_g . g = id.
- reverse . concat= concat . reverse . map reverse
- filter p . concat = concat . map (filter p)

Haskell – foldr

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

a : b : c : [] -> f a (f b (f c z))
```

```
     :                        f
    / \                      / \
   a   :    foldr f z       a   f
      / \   -------->          / \
     b   :                    b   f
        / \                      / \
       c  []                    c   z
```

```
Main> foldr (+) 0 [1..100]
5050

Main> foldr (\x y->10*y+x) 0 [1,2,3,4]
4321
```

```
-- g je vnorená lokálna funkcia

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z = g
  where    g []     = z
           g (x:xs) = f x (g xs)
```

Haskell – foldl

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

a : b : c : [] -> f (f (f z a) b) c

```
    :                         f
   / \                       / \
  a   :    foldl f z        f   c
     / \   ------->        / \
    b   :                 f   b
       / \               / \
      c  []             z   a
```

```
Main> foldl (+) 0 [1..100]
5050

Main> foldl (\x y->10*x+y) 0 [1,2,3,4]
1234
```

Vypočítajte

- foldr max (-999) [1,2,3,4]
  foldl max (-999) [1,2,3,4]

- foldr (_ -> \y ->(y+1)) 0 [3,2,1,2,4]
  foldl (\x -> _ ->(x+1)) 0 [3,2,1,2,4]

- foldr (-) 0 [1..100] =

(1-(2-(3-(4-...-(100-0))))) = 1-2 + 3-4 + 5-6 + ... + (99-100) = -50

- foldl (-) 0 [1..100] =

  (...(((0-1)-2)-3) ... - 100) = -5050

Funkcia je hodnotou

- [a->a] je zoznam funkcií typu a->a
  napríklad: [(+1),(+2),(*3)] je [\x->x+1,\x->x+2,\x->x*3]

- čo je foldr (.) id [(+1),(+2),(*3)] ?? [a->a]
  akého je typu
  foldr (.) id [(+1),(+2),(*3)] 100 303
  foldl (.) id [(+1),(+2),(*3)] 100 ???

lebo skladanie fcií je asociatívne:

- ((f . g) . h) x = (f . g) (h x) = f (g (h x)) = f ((g . h) x) = (f . (g . h)) x

- funkcie nevieme porovnávať, napr. head [(+1),(+2),(*3)] = id

- funkcie vieme permutovať, length $ permutations [(+1),(+2),(*3),(^2)]

Maximálna permutácia funkcií

- zoznam funkcií aplikujeme na zoznam argumentov

```
apply           :: [a -> b] -> [a] -> [b]
apply fs args = [ f a | f <- fs, a <- args]
```

```
apply [(+1),(+2),(*3)] [100, 200]
[101,201,102,202,300,600]
```

- čo počíta tento výraz

```
maximum $
    apply
      (map (foldr (.) id) (permutations [(+1),(^2),(*3),(+2),(/3)]))
      [100]
```

31827

- (+1).(+2).(*3).(^2).(/3) 100

3336.333333333334

- ((/3).(^2).(*3).(+2).(+1)) 100

31827.0

Kvíz

```
foldr (:) [] xs =    xs

foldr (:) ys xs =    xs++ys


foldr ? ? xs = reverse xs
```

Vlastnosti

Fussion Law:

Ak platí

$$f\ z1 = z2\ \&\&\ f\ (g1\ a\ b) = g2\ a\ (f\ b)$$

potom platí

$$f\ .\ foldr\ g1\ z1 = foldr\ g2\ z2$$

Príklad použitia Fussion Law:

$$(n*).\ foldr\ (+)\ 0 = foldr\ (+).(n*)\ 0$$

Dôkaz (pomocou Fussion Law): overíme predpoklady

f = (n*), z1 = z2 = 0, g1 = (+), g1 = g2 = (+). (n*)

- (n*) 0 = 0
- (n*) (a+b) = (n*a + n*b) = (+).(n*) a ((n*) b)

Vlastnosti

Acid Rain (fold/build/deforestation theorem)

foldr f z . g (:) [] = g f z

Intuícia: Keď máme vytvoriť zoznam pomocou funkcie g zo zoznamových konštruktorov (:) [], na ktorý následne pustíme foldr, ktorý nahradí (:) za f a [] za z, namiesto toho môžeme konštruovať priamo výsledný zoznam pomocou g f z.

Otypujme si to:

Ak z :: u, potom f :: x->u->u, foldr f z :: [x]->u

Ľavá strana: ([x]->u).(t->[x]) výsledkom je typ t->u

Pravá strana: g :: (x -> u -> u) -> u -> (t ->u)

length . map _ = length

foldr f z . g (:) [] = g f z

```
map :: (a -> b)  -> [a] -> [b]
map h = foldr ((:) . h) []                -- (:).h a as = (:)(h a as) = h a: as
       = (\f z -> foldr (f . h) z) (:) []

length    :: [a] -> Int
length    = foldr (\ _ -> \n -> n+1) 0

length . map h = length

(foldr (\ _ -> \n -> n+1) 0) . (foldr ((:) . h) [])  =
= podľa Acid Rain theorem (f  = (\ _ -> \n -> n+1), z = 0,  ale čo je g ? ...
g x y = (foldr (x . h) y)
g f z = (foldr (f . h) z) = foldr ((\ _ ->\n -> n+1) . h) 0 =
                                   foldr ((\_ ->\n -> n+1) ) 0 = length
lebo
((\ _->\n ->n+1) . h) x y = (\ _->\n->n+1) (h x) y = (\n->n+1) y = y+1
```

foldr a foldl pre pokročilejších

definujte foldl pomocou foldr, alebo naopak:

```
   :                  f
  / \                / \
 a   :   foldr f z  a   f
    / \  -------->     / \
   b   :              b   f
      / \                / \
     c  []              c   z
```

```
myfoldl f z xs = foldr (\x -> \y -> (f y x)) z (myReverse xs)
myfoldr f z xs = foldl (\x -> \y -> (f y x)) z (myReverse xs)
```

(flip f)

```
   :                  f
  / \                / \
 a   :   foldl f z   f   c
    / \  ------>    / \
   b   :           f   b
      / \         / \
     c  []       z   a
```

- odstránime myReverse
  ```
  myReverse xs = foldr (\x -> \y -> (y ++ [x])) [] xs
  myfoldl' f z xs = foldr (\x -> \y -> (f y x)) z
                          (foldr (\x -> \y -> (y ++ [x])) [] xs)
  ```
- odstránime ++
  ```
  xs ++ ys = foldr (:) ys xs
  myfoldl'' f z xs = foldr (\x -> \y -> (f y x)) z
                           (foldr (\x -> \y -> (foldr (:) [x] y)) [] xs)
  ```

hmmm..., teoreticky (možno) zaujímavé, prakticky nepoužiteľné ...

foldr a foldl posledný krát

Zamyslime sa, ako z foldr urobíme foldl:

```
    :                   f
   / \                 / \
  a   :   foldr f z   a   f
     / \  -------->      / \
    b   :               b   f
       / \                 / \
      c  []               c   z
```

induktívne predpokladajme, že rekurzívne volanie foldr nám vráti výsledok, t.j. hodnotu y, ktorá zodpovedá foldl:

- y = myfoldl f [b,c] = \z -> f (f z b) c

```
    :                   f
   / \                 / \
  a   :   foldl f z   f   c
     / \  ------>    / \
    b   :           f   b
       / \         / \
      c  []       z   a
```

nech x je ďalší prvok zoznamu, t.j.

- x = a

ako musí vyzerať funkcia ?, ktorou fold-r-ujeme, aby sme dostali myfoldl f [a,b,c] = \z' -> f (f (f z' a) b) c = ? x y

- ? = (\x y z' -> y (f z' x))

dosaďme:

- (\z' -> (\z -> f (f z b) c) (f z' a)) =
- (\z' -> f (f (f z' a) b) c) =
- \z' -> f (f (f z' a) b) c

Pre tých, čo neveria, fakt posledný krát

? = (\x y z′ -> y (f z′ x))

- myfoldl''' f xs z = foldr (\x y z -> y (f z x)) id xs z
- myfoldl''' f [] = id
- myfoldl''' f [c] = (\x y z -> y (f z x)) c id = \z -> f z c
- myfoldl''' f [b,c] = (\x y z -> y (f z x)) b (\w -> f w c) =
  \z -> (\w -> f w c) (f z b) =
  \z -> f (f z b) c
- myfoldl''' f [a,b,c] = (\x y z -> y (f z x)) a (\w -> f (f w b) c) =
  \z -> (\w -> f (f w b) c) (f z a) =
  \z -> f (f (f z a) b) c
- myfoldl '''' f z xs = foldr (\x y z -> y (f x z)) id xs z

... doma skúste foldr pomocou foldl ...