Laiskus

Meil on kaks standardset reduktsioonijärjekorda:

- normaaljärjekord ja
- aplikatiivjärjekord.

Mis on eelised ja puudused? Kumba eelistada?

Normaaljärjekord on parem!

- Normaaljärjekord leiab alati normaalkuju, kui see eksisteerib! Normaalkuju ei ole näiteks: $(\lambda\, x.\ x\ x)\ (\lambda\, x.\ x\ x)$
- Ei väärtusta argumente mida ei kasutata. Näiteks $(\lambda\, x.\ 0)$ (fact 100)
- Funktsiooni argumendiks võib anda avaldise, millel pole normaalkuju.
  - Praktikas näiteks lõpmatu list. Näiteks Haskellis:

```
sieve (p:xs) = [x | x←xs, x `mod` p ≠ 0]
primes = map head (iterate sieve [2..])

Main> take 3 primes
[2,3,5]
```

- Kui argumenti kasutatakse mitu korda, väärtustatakse seda mitu korda :(

Laisk väärtustamine on veel parem!

- Nagu normaaljärjekord aga argumenti väärtustatakse maksimaalselt üks kord.
- Graafireduktsioon, mitte puu-redutkstioon.
- (Laisal) väärtustamisel on ka kitsam tähendus, mis ei kasuta reduktsioonisammu vaid on lähedasem kompileerimisele. Sellest räägime mõnes teises loengus/kursusel.

Näide (Haskell)

```
type Queen = (Int, Int)

attacking :: Queen → Queen → Bool
attacking (x1, y1) (x2, y2) =
    (x1 == x2) || (y1 == y2) || abs (x1 - x2) == abs (y1 - y2)

placeable :: Queen → [Queen] → Bool
placeable p qs = and [ (not (attacking p q)) | q ← qs ]

nQueens :: Int → [[Queen]]
nQueens n = nQueens' n where
    nQueens' 0 = [[]]
    nQueens' k =
        [ q:qs   | qs ← nQueens' (k - 1)
                 , q  ← [(k, y) | y ← [1..n]]
                 , placeable q qs]
```

Näide (Haskell)

```
*Main> time $ queens 10
```

```
♕ . . . . . . . . .
. . ♕ . . . . . . .
. . . . . ♕ . . . .
. . . . . . . ♕ . .
. . . . . . . . . ♕
. . . . ♕ . . . . .
. . . . . . . . ♕ .
. ♕ . . . . . . . .
. . . ♕ . . . . . .
. . . . . . ♕ . . .
```

```
0.007199s
```

Näide (Haskell)

```
fib :: Integer → Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

*Main> time $ print $ fib 30
1346269
1.183209s
```

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

*Main> time $ print $ fibs !! 30
1346269
0.000059s
```

Näide (Haskell)

```
foldl (+) 0 [10,20,30]
⟶ foldl (+) (0+10) [20,30]
⟶ foldl (+) (0+10+20) [30]
⟶ foldl (+) (0+10+20+30) []
⟶ 0+10+20+30
⟶ 10+20+30
⟶ 30+30
⟶ 60
```

vs.

```
foldl' (+) 0 [10,20,30]
⟶ foldl' (+) (0+10) [20,30]
⟶ foldl' (+) 10 [20,30]
⟶ foldl' (+) (10+20) [30]
⟶ foldl' (+) 30 [30]
⟶ foldl' (+) (30+30) []
⟶ foldl' (+) 60 []
⟶ 60
```

Sama arv samme.

Näide (Haskell)

```
*Main> time $ print $ foldl  (+) 0 [0..10000000]
50000005000000
2.895953s

*Main> time $ print $ foldl' (+) 0 [0..10000000]
50000005000000
0.268244s
```

- Laiskust mittekasutava koodi jaoks aeglasem ja raskemini optimeeritav kui agar väärtustamine. :(

Aplikatiivjärjekord on parem!

- Parem jõudlus lihtsamate optimisatsioonide abil.
  - `Int` parameeter protsessori registris.
- Jõudluse analüüs lihtsam. Vähem üllatusi.
- Aga lõpmatud listid???

Lazy tüübid

Idrises on (lihtsustatult) selline andmestruktuur:

```
data Lazy : Type → Type where
     Delay : (val : a) → Lazy a
```

... ja funktsioon

```
Force : Lazy a → a
```

- `Delay` argumenti ei väärtustata.
- `Force (Delay x) = x`
- Näiteks `and : List (Lazy Bool) → Bool`

Laisad listid

Lõpmatud laisad listid ehk striimid:

```
data Stream : Type → Type where
     (::) : a → Lazy (Stream a) → Stream a
```

ja laisad listid

```
mutual
    data LList : Type → Type where
        Nil : LList a
        (::) : a → LazyList a → LList a

    LazyList : Type → Type
    LazyList a = Lazy (LList a)
```

Näide (Idris)

```
Queen : Type
Queen = (Int, Int)

attacking : Queen → Queen → Bool
attacking (x1, y1) (x2, y2) =
    (x1 == x2) || (y1 == y2) || abs (x1 - x2) == abs (y1 - y2)

placeable : Queen → List Queen → Bool
placeable p qs = and [ delay (not (attacking p q)) | q ← qs ]
-- erinevus Haskellist:
--    and töötab List (Lazy a) peal, peame lisama delay

nQueens : Int → LazyList (List Queen)
nQueens n = nQueens' n where
    nQueens' : Int → LazyList (List Queen)
    nQueens' 0 = [[]]
    nQueens' k =
        [ q::qs | qs ← nQueens' (k - 1)
                , q  ← cast [(k, y) | y ← [1..n]]
                , placeable q qs]
-- erinevus Haskellist:
--    listikomprehensioon töötab List-i peal,
--    peame konverteerima LazyList-iks (cast).
```

- Esimene element: 10 korda kiirem kui Haskellis!