Reduktsiooni järjekorrad

Aplikatiivne järjekord — väärtustatakse seest välja

$$\begin{aligned} \texttt{double}(\underline{2+3}) &\Longrightarrow \underline{\texttt{double}\ 5} \\ &\Longrightarrow \underline{2\times 5} \\ &\Longrightarrow 10 \end{aligned}$$

Normaaljärjekord — väärtustatakse väljast sisse

$$\begin{aligned} \underline{\texttt{double}(2+3)} &\Longrightarrow 2\times(\underline{2+3}) \\ &\Longrightarrow \underline{2\times 5} \\ &\Longrightarrow 10 \end{aligned}$$

Church-Rosser'i teoreem: Avaldise normaalkuju ei sõltu reduktsioonijadast.

Reduktsiooni järjekorrad

Normaalkuju leidumine sõltub reduktsioonijärjekorrast!

$$
\begin{array}{lcl}
\texttt{const}\ x\ y & = & x \\
\texttt{loop} & = & \texttt{loop}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{const}\ 5\ \underline{\texttt{loop}} & \Longrightarrow & \texttt{const}\ 5\ \underline{\texttt{loop}} \\
 & \Longrightarrow & \texttt{const}\ 5\ \underline{\texttt{loop}} \\
 & \Longrightarrow & \ldots
\end{array}
$$

$$
\underline{\texttt{const}\ 5\ \texttt{loop}} \Longrightarrow 5
$$

Normaliseerimisteoreem: Kui avaldisel leidub normaalkuju, siis normaaljärjekorras reduktsioonijada on lõplik.

Reduktsiooni järjekorrad

Normaaljärjekord võib olla ebaefektiivne!

$$\underline{\texttt{square}(2+3)} \Longrightarrow \underline{\texttt{square}\ 5} \Longrightarrow \underline{5 \times 5} \Longrightarrow 25$$

$$\underline{\texttt{square}(2+3)} \Longrightarrow (\underline{2+3}) \times (2+3) \Longrightarrow 5 \times (\underline{2+3}) \Longrightarrow 5 \times 5 \Longrightarrow 25$$

Laisk väärtustamine = normaaljärjekord + graafireduktsioon

$$\underline{\texttt{square}(2+3)} \Longrightarrow x \times x \quad \texttt{where}\ x = \underline{2+3} \Longrightarrow \underline{x \times x} \quad \texttt{where}\ x = 5 \Longrightarrow 25$$

Reduktsiooni järjekorrad

Samaselt tõesuse kontroll `foldr` abil

```
     foldr (&&) True [False, True, False]
==>  False && (foldr (&&) True [True, False])
==>  False
```

Samaselt tõesuse kontroll `foldl` abil

```
     foldl (&&) True [False, True, False]
==>  foldl (&&) (True && False) [True, False]
==>  foldl (&&) ((True && False) && True) [False]
==>  foldl (&&) (((True && False) && True) && False) []
==>  ((True && False) && True) && False
==>  False
```

Reduktsiooni järjekorrad

Listi elementide summa `foldr` abil

```
foldr (+) 0 [1,2,3]  ==>  1 + foldr (+) 0 [2,3]
                     ==>  1 + (2 + foldr (+) 0 [3])
                     ==>  1 + (2 + (3 + foldr (+) 0 []))
                     ==>  1 + (2 + (3 + 0))
                     ==>  6
```

Listi elementide summa `foldl` abil

```
foldl (+) 0 [1,2,3]  ==>  foldl (+) (0+1) [2,3]
                     ==>  foldl (+) ((0+1)+2) [3]
                     ==>  foldl (+) (((0+1)+2)+3) []
                     ==>  ((0+1)+2)+3
                     ==>  6
```

Reduktsiooni järjekorrad

- Funktsioon on agar, kui funktsiooni tulemuse leidmiseks tuleb argument alati väärtustada
- Pole vahet millist reduktsiooni järjekorda kasutada!
- Efektiivsem on aplikatiivne (või mingi sega-)järjekord!?!
- Listi elementide summa `foldl` abil (segajärjekorras)

```
foldl (+) 0 [1,2,3]  ==>  foldl (+) (0+1) [2,3]
                     ==>  foldl (+) 1 [2,3]
                     ==>  foldl (+) (1+2) [3]
                     ==>  foldl (+) 3 [3]
                     ==>  foldl (+) (3+3) []
                     ==>  foldl (+) 6 []
                     ==>  6
```

Laisk väärtustamine

- Lõpmatud listid

```
take 5 [1..]  ===> [1,2,3,4,5]

multiples :: [[Int]]
multiples = [ [ m*n | m <- [1..]] | n <- [1..]]

multiples ===> [[1, 2, 3, 4, 5, ...],
                [2, 4, 6, 8,10, ...],
                [3, 6, 9,12,15, ...],
                ......................]

take 4 (multiples !! 3) ===> [4,8,12,16]
```

Laisk väärtustamine

- Funktsioon `iterate` (eeldefineeritud)

$$iterate\ f\ x = [x,\ f\ x,\ f^2 x,\ f^3 x,\ \ldots]$$

- Definitsioon

```
iterate     :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

- Näited

```
powertables :: [[Int]]
powertables = [iterate (*n) 1 | n <- [2..]]

digits :: Int -> [Int]
digits = reverse . map (`mod` 10) . takeWhile (/= 0)
                                  . iterate (`div` 10)
```

Laisk väärtustamine

- Ruutjuur Newton–Raphson'i meetodil

$$a_{i+1} = \frac{a_i + \frac{n}{a_i}}{2}$$

```
next n x = (x + n/x)/2.0

within eps (x1:x2:xs)
           | abs(x1-x2) <= eps  = x2
           | otherwise          = within eps (x2:xs)

sqrt1 n = within eps (iterate (next n) a0)
          where eps = 0.00001
                a0  = 1.0
```

Laisk väärtustamine

- Erasthostenese sõel
  1. Kirjuta üles kõik positiivsed täisarvud alates arvust 2;
  2. Tähista jada esimene element p kui algarv;
  3. Kustuta jadast kõik arvu p kordsed;
  4. Mine tagasi sammule 2.

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

Laisk väärtustamine

- Kaheksa lippu

```
queens 0     = [[]]
queens (m+1) = [p++[n] | p<-queens m, n<-[1..8]
                                    , safe p n]

safe p n = and [not(check (i,j) (m,n))
                    | (i,j) <- zip [1..] p]
         where m = 1 + length p

check (i,j) (m,n) = j==n || (i+j==m+n)
                         || (i-j==m-n)
```

Laisk väärtustamine

- Fibonacci jada

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots$$

- n-inda fibonacci arvu leidmine

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

Laisk väärtustamine

Toodud definitsioon on väga ebaefektiivne (eksponentsiaalse keerukusega)

```
fib 8 ===> fib 7 + fib 6
      ===> (fib 6 + fib 5) + (fib 5 + fib 4)
      ===> ((fib 5 + fib 4) + (fib 4 + fib 3))
           + ((fib 4 + fib 3) + (fib 3 + fib 2))
      ===> (((fib 4 + fib 3) + (fib 3 + fib 2))
            + ((fib 3 + fib 2) + (fib 2 + fib 1)))
           + (((fib 3 + fib 2) + (fib 2 + fib 1))
             + ((fib 2 + fib 1) + (fib 1 + fib 0)))
      ...
```

Laisk väärtustamine

- Fibonacci jada

$$\begin{array}{cccccccccc} & 1 & 1 & 2 & 3 & 5 & 8 & 13 & \ldots \\ & 1 & 2 & 3 & 5 & 8 & 13 & 21 & \ldots \\ \hline + & 2 & 3 & 5 & 8 & 13 & 21 & 34 & \ldots \end{array}$$

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

- Toodud definitsioon on palju efektiivsem (lineaarse keerukusega) ja kasutab palju vähem ressursse!

Laisk väärtustamine

- Liita listi kõigile elementidele listi minimaalne element

```
incmin xs = map  (minv+) xs
          where minv = minimum xs
```

- Listi ühekordse läbimisega versioon

```
incmin [] = []
incmin xs = newlist
  where (minv,newlist) = onepass xs
        onepass [a]    = (a, [a+minv])
        onepass (a:as) = (a `min` b, (a+minv):bs)
                         where (b,bs) = onepass as
```