Reduktsiooni järjekorrad

- Reduktsiooni järjekord ei oma tähtsust!
- Church-Rosseri teoreem: iga λ-termi e_0, e_1 ja e_2 korral, kui $e_0 \twoheadrightarrow_\beta e_1$ ja $e_0 \twoheadrightarrow_\beta e_2$, siis leidub e_3 selline et $e_1 \twoheadrightarrow_\beta e_3$ ja $e_2 \twoheadrightarrow_\beta e_3$.
- Järeldus: λ-termide normaalkujud on unikaalsed.
- NB! Normaalkuju leidumine ei ole garanteeritud!
- Näide:

$$\begin{array}{lll} (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) & \to_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \to_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \to_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \to_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \ldots & \end{array}$$

Reduktsiooni järjekorrad

- Reduktsiooni järjekord on oluline!
- Normaaljärjekord: alati redutseerida vasakpoolne väline reedeks.

$$\underline{(\lambda x.\, y)((\lambda x.\, x\, x)(\lambda x.\, x\, x))} \;\to_\beta\; y$$

- Aplikatiivne järjekord: alati redutseerida vasakpoolne sisemine reedeks.

$$\begin{array}{ll} & (\lambda x.\, y)(\underline{(\lambda x.\, x\, x)(\lambda x.\, x\, x)}) \\ \to_\beta & (\lambda x.\, y)(\underline{(\lambda x.\, x\, x)(\lambda x.\, x\, x)}) \\ \ldots & \end{array}$$

- Normaliseerimisteoreem: kui λ-termil leidub normaalkuju, siis on see saavutatav normaaljärjekorraga.

Reduktsiooni järjekorrad

- Normaaljärjekord võib olla ebaefektiivne!
- Normaljärjekord:

$$\begin{array}{lll} \underline{(\lambda x.\, x\, x)\ ((\lambda x.\, x)\ (\lambda x.\, x))} & \to_\beta & \underline{((\lambda x.\, x)\ (\lambda x.\, x))}\ ((\lambda x.\, x)\ (\lambda x.\, x)) \\ & \to_\beta & \underline{(\lambda x.\, x)\ ((\lambda x.\, x)\ (\lambda x.\, x))} \\ & \to_\beta & \underline{(\lambda x.\, x)\ (\lambda x.\, x)} \\ & \to_\beta & \lambda x.\, x \end{array}$$

- Aplikatiivne järjekord:

$$\begin{array}{lll} (\lambda x.\, x\, x)\ \underline{((\lambda x.\, x)\ (\lambda x.\, x))} & \to_\beta & \underline{(\lambda x.\, x\, x)\ (\lambda x.\, x)} \\ & \to_\beta & \underline{(\lambda x.\, x)\ (\lambda x.\, x)} \\ & \to_\beta & \lambda x.\, x \end{array}$$

- Laisk väärtustamine = normaaljärjekord + graafireduktsioon

Konstantidega lambda-arvutus

- Konstantidega λ-termide süntaks:

$$\begin{array}{lcll} e & ::= & x & \text{muutuja} \\ & | & c & \text{konstant} \\ & | & (e_1\ e_2) & \text{aplikatsioon} \\ & | & (\lambda x.\ e) & \text{abstraktsioon} \end{array}$$

- Iga konstandiga seotakse mingi arv δ-reduktsiooni reegleid.
- Näide: Naturaalarvud $(0, 1, 2, \ldots)$ ja liitmine $(+)$.

$$\begin{array}{llll} + \ 0\ 0 & \twoheadrightarrow_\delta & 0 & \qquad + \ 1\ 0 \ \twoheadrightarrow_\delta \ 1 \\ + \ 0\ 1 & \twoheadrightarrow_\delta & 1 & \qquad + \ 1\ 1 \ \twoheadrightarrow_\delta \ 2 \\ \ldots & & & \qquad \ldots \end{array}$$

- δ-reeglite lisamine võib kokkuvoolavuse ära rikkuda!

Tüübiklassid I

Idrisis tuleb tihti kirjutada funktsioone, mis erinevad ainult natuke tüübi poolest:

```
equalChar   :   Char →   Char → Bool
equalInt    :    Int →    Int → Bool
equalString : String → String → Bool
```

Sellist koodi *ei saa* kirjutada parameetrilise polümorfse funktsiooniga, kuna funktsioonide implementatsioon on erinev:

```
equal : a → a → Bool
equal = ???  -- pole def. mis töötaks iga tüübi korral
```

Selleks saame teha liidese (Haskellis tüübiklassi)

```
interface Equal a where
    (==) : a → a → Bool

Equal Int where
    a == b = ...  -- Int'ide võrdsus
Equal Bool where
    a == b = ...  -- Bool'ide võrdsus
```

Tüübiklassid II

Idrise standardteegis on defineeritud tüübiklass `Eq`:

```
interface Eq a where
  (==), (≠) : a → a → Bool

    -- Minimaalne definitioon peab sisaldama:
    -- (==) või (≠)
  x ≠ y = not (x == y) -- vaikedefinitsioon
  x == y = not (x ≠ y) -- vaikedefinitsioon
```

Võrldusoperaatori tüüp on:

```
(==) : Eq a ⇒ a → a → Bool
```

s.t me saame operaatorit kasutada, kui tema argumentitüübil on defineeritud `Eq` instants!

Kõikidel polümorfsetel funktsioonidel, mis kasutavad võrdust peab tüübi *kontekst* olema `Eq`:

```
lookup : Eq a ⇒ a → List (a, b) → Maybe b
```

Näide

```
data ValgusFoor = Punane | Kollane | Roheline

Eq ValgusFoor where
  Punane   == Punane   = True
  Kollane  == Kollane  = True
  Roheline == Roheline = True
  _        == _        = False

test : Bool
test = Kollane == Roheline   -- False
```

Range tüübiklass

Enumeratsioone kirjeldab järgnev tüübiklass:

```
interface Range a where
  rangeFromTo : a → a → List a           -- [x..y]
  rangeFromThenTo : a → a → a → List a   -- [x,y..z]

  rangeFrom : a → Stream a               -- [x..]
  rangeFromThen : a → a → Stream a       -- [x,y..]
```

Sisend-väljund Idrises

- Idrise puhtad funktsioonid ei võimalda teha mittepuhtaid arvutusi.
  - Puhas — funktsiooni tulemus sõltub ainult argumentide väärtusest.
  - Ei saa teha näiteks juhuarvude funktsiooni `random : () → Int`
- Lahendus: `IO` *monaad*
  - '`IO a`' tüüpi väärtus — „masin mis arvutab `a` tüüpi väärtuse“
  - `pure : a → IO a` — masin tagastab esimese argumendi väärtuse
  - `(>>=) : IO a → (a → IO b) → IO b` — masin käivitab esimese argumendi ja rakendab tulemuse teisele
  - ... lisaks baasfunktsioonid nagu `putStrLn : String → IO ()` ja `getLine : IO String`.
- Nii saab kombineerida olemasolevaid `IO` „masinaid“. Näiteks:

```
main : IO ()
main = randomRIO (1, 10) >>= classify >>= putStrLn
  where classify : Int → IO String
        classify x = if x `mod` 2 == 1 then
                         pure "paaritu"
                     else
                         pure "paaris"
```

`do`-süntaks I

Eelneval slaidil olnud koodi on keeruline lugeda ja kirjutada:

```
main : IO ()
main = randomRIO (1, 10) >>= classify >>= putStrLn
  where classify : Int → IO String
        classify x = if x `mod` 2 == 1 then
                         pure "paaritu"
                     else
                         pure "paaris"
```

Sama saab saavutada järgnevalt

```
main : IO ()
main = do
  r ← randomRIO (1, 10)
  c ← classify r
  putStrLn c
     where classify : Int → IO String
           classify x = ...
```

või

```
main : IO ()
main = do
  r ← randomRIO (1, 10)
  if odd r
    then putStrLn "paaritu"
    else putStrLn "paaris"
```

do-süntaks II

Näide

```
proc : IO ()
proc = do
  s ← getLine
  let n : Int
      n = read s
      n2 : Int
      n2 = 2*n
  putStrLn ("Kaks korda " ++ s ++ " on " ++ show n2)
```

Do-süntaks algab `do`-võtmesõnaga, millele järgnevad *järjest töödeldavad* laused.

- Laused mustriga $x \leftarrow p$, kus p : `IO` a siis $x : a$,
- `let` laused ning
- avaldised e, mille tüüp on `IO a`.

Mitme `do` kasutamine

- `do` seob kokku IO-avaldised, kuid ei saa vaadata konstruktsioonide sisse
- S.t. ühe avaldise jaoks pole do-d vaja
  - `main = putStrLn "Hello World!"`
- Hargenmise puhul võib olla vaja kasutada mitut do-d:

```
main = do
  putStrLn "Kirjuta midagi!"
  xs ← getLine
  if (xs=="")
    then putStr "Sõnakuulmatu!"
    else do
      putStrLn "Tänan!"
      putStrLn ("Kirjutasid: " ++ xs)
```

`do` tähendus

- `a >>= f` on sama mis

```
do x ← a
   f x
```

- `a >> b` on sama mis

```
do a
   b
```

```
do a                    do do a
   b    on sama mis           b
   c                       c
```

- Fixity:

```
infixl 1 >>
infixl 1 >>=
```

Näide

```
main : IO ()
main = do
 putStrLn "Kirjuta midagi!"
 xs ← getLine
 if (xs=="")
   then putStr "Sõnakuulmatu!"
   else do
     putStrLn "Tänan!"
     putStrLn ("Kirjutasid: " ++ xs)
```

on sama mis

```
main : IO ()
main =
 putStrLn "Kirjuta midagi!" >>
 getLine  >>= (λ xs ⇒
 if (xs=="")
   then putStr "Sõnakuulmatu!"
   else
     putStrLn "Tänan!" >>
     putStrLn ("Kirjutasid: " ++ xs)
 )
```

Näide

```
main : IO ()
main =
 putStrLn "Kirjuta midagi!" >>
 getLine  >>= (λ xs →
 if (xs=="")
   then putStr "Sõnakuulmatu!"
   else
     putStrLn "Tänan!" >>
     putStrLn ("Kirjutasid: " ++ xs)
 )
```

on sama mis

```
main : IO ()
main =
  putStrLn "Kirjuta midagi!" >>  getLine  >>= ifThenElse
    where ifThenElse xs = if (xs=="") then case1 else case2 xs
          case1         = putStr "Sõnakuulmatu!"
          case2 xs      = putStrLn "Tänan!" >> putStrLn ("Kirjutasid: "++xs)
```