Lambda-arvutus

- λ-termide süntaks:

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

- Sulgudest hoidumine:

$$\begin{array}{lcl} e_1\ e_2\ \ldots\ e_n & \equiv & ((\ldots(e_1\ e_2)\ldots)e_n) \\ \lambda x.\ e_1\ e_2\ \ldots\ e_n & \equiv & (\lambda x.\ (e_1\ e_2\ \ldots\ e_n)) \\ \lambda x_1\ x_2\ \ldots\ x_n.\ e & \equiv & (\lambda x_1.\ (\lambda x_2.\ (\ldots(\lambda x_n.\ e)\ldots)) \end{array}$$

- Näited:

$$\lambda x.\ x \qquad\qquad ((\lambda x.\ (\lambda f.\ f\ x))\ y)\ (\lambda z.\ z)$$

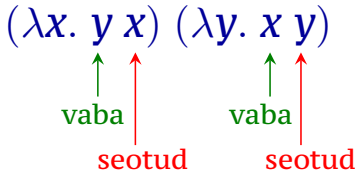
Lambda-arvutus

- Puhas λ-arvutus „räägib“ ainult funktsioonidest.
- Arve ja teisi andmetüüpe eraldi ei ole, kuna neid saab defineerida λ-termidena.
- Mugavuse pärast kasutame siiski arve ja binaarseid operaatoreid nagu nad oleksid keelde sisseehitatud.
- Samuti kasutame makro-definitsioone (peavad olema mitterekursiivsed).
- Näited:

$$\begin{array}{lcl} \mathrm{add} & \equiv & \lambda x\, y.\ x + y \\ \mathrm{dbl} & \equiv & \lambda x.\ 2 * x \\ \mathrm{I} & \equiv & \lambda x.\ x \\ \mathrm{K} & \equiv & \lambda x\, y.\ x \\ \mathrm{S} & \equiv & \lambda f\, g\, x.\ f\, x\, (g\, x) \end{array}$$

Vabad ja seotud muutujad

- Muutuja x on *vaba* λ-termis e (so. $x \in \mathrm{FV}(e)$), kui ta ei asu sümboli λx mõjupiirkonnas.
- Vastasel korral on x *seotud*.



Vabad ja seotud muutujad

- Vabade muutujate induktiivne definitsioon:

$$\begin{array}{lcl} \mathbf{FV}(x) & = & \{x\} \\ \mathbf{FV}(e_1\ e_2) & = & \mathbf{FV}(e_1) \bigcup \mathbf{FV}(e_2) \\ \mathbf{FV}(\lambda x.\ e) & = & \mathbf{FV}(e) - \{x\} \end{array}$$

- Ilma vabade muutujateta λ-termid on kinnised.
- Näited:

$$\begin{array}{lcl} \mathbf{FV}(\lambda x\, y.\ x\ y) & = & \emptyset \\ \mathbf{FV}(\lambda x.\ (\lambda y.x)\,(\lambda z.y)) & = & \{y\} \end{array}$$

α-konversioon

- Seotud muutujate nimed ei oma tähtsust!
- λ-termid e_1 ja e_2 on α-kongruentsed (tähistus $e_1 =_\alpha e_2$) kui nad on identsed seotud muutujate ümbernimetamise täpsuseni.
- Näited:

$$\begin{array}{lcl} \lambda x.\ x & =_\alpha & \lambda y.\ y \\ \lambda x.\ f\ x & =_\alpha & \lambda z.\ f\ z \\ \lambda x.\ (\lambda y.\ y)\ x & =_\alpha & \lambda y.\ (\lambda y.\ y)\ y \\ \lambda x\ y.\ x + y & \neq_\alpha & \lambda y\ y.\ y + y \end{array}$$

Lambdad Idris-es

- Süntaks:

```
λ a, b, c, d ⇒ e
```

  (λ ja ⇒ asemel kirjutada koodis \ ja =>)

- Idris kasutab seda ka siseselt.
  Defineerides funktsiooni

```
f : ...
f x y z = ...
```

  teisendab kompilaator koodi selliseks

```
f : ...
f = λ x, y, z ⇒ ...
```

- Sellist süntaksit saab ka ise kasutada funktsiooni argumendina:

```
map (λ x ⇒ x+1) [1,2,3,4]
```

Kõrgemat järku funktsioon

... on funktsioon, mis võtab argumendiks (või tagastab) funktsiooni.

FP keskne mõte: arvutada saab ka arvutustega (e. funktsioonidega).

Näiteks `map`:

```
map : (a → b) → List a → List b
map f []     = []
map f (x::xs) = f x :: map f xs
```

Funktsiooni `map` illustreerib järgmine võrdus:

$$\texttt{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

Näide:

```
inverses : List Double → List Double
inverses xs = map inverse xs
    where inverse : Double → Double
          inverse x = 1 / x

Main> inverses [1,2,4,8]
[1.0, 0.5, 0.25, 0.125]
```

Kõrgemat järku funktsioon: `foldr`

```
foldr : (a → b → b) → b → List a → b
foldr f b []     = b
foldr f b (x::xs) = f x (foldr f b xs)
```

Funktsiooni `foldr` illustreerib järgmine võrdus:

$$\texttt{foldr}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = x_1 + (x_2 + (\ldots + (x_n + b)))$$

Funktsiooni `map` saab defineerida läbi `foldr`-i

```
map : (a → b) → List a → List b
map f xs = foldr g [] xs
   where g : a → List b → List b
         g x y = (f x) :: y
```

ehk

$$\texttt{foldr}\ g\ []\ [x_1, x_2, \ldots, x_n] = x_1 \text{`}g\text{`} (x_2 \text{`}g\text{`} (\ldots \text{`}g\text{`} (x_n \text{`}g\text{`} [])\ldots)) = f\ x_1 : f\ x_2 : \ldots : f\ x_n : []$$

Kõrgemat järku funktsioon: `foldl`

```
foldl : (b → a → b) → b → List a → b
foldl f b []     = b
foldl f b (x::xs) = foldl f (f b x) xs
```

Funktsiooni `foldl` illustreerib järgmine võrdus:

$$\texttt{foldl}\ (+)\ b\ [x_1, x_2, \ldots, x_n] = (((b + x_1) + x_2) + \ldots) + x_n$$

Funktsiooni `reverse` saab defineerida läbi `foldl`-i

```
reverse : List a → List a
reverse xs = foldl g [] xs
  where g : List a → a → List a
        g x y = y :: x
```

ehk

$$\texttt{foldl}\ g\ []\ [x_1, x_2, \ldots, x_n] = ((([]\ `g`\ x_1)\ `g`\ x_2)\ `g`\ \ldots)\ `g`\ x_n = x_n : \ldots : x_2 : x_1 : []$$

Kõrgemat järku funktsioonid kui disainimustrid

```
f : List Double → Double
f (x::xs) = x + f xs
f []      = 0
```

vs.

```
f = foldr (+) 0
```

- Lihtsate funktsioonide puhul on ka rekursiivne lahendus selge.
- Pikema ülesande puhul on hea eraldada listi töötlemine.
- S.t. mitte *bittide*-tasemel vaid abstraktsemalt.
  Kumb definitsioon on selgem:

```
f : List Double → Double
f (0::_)  = 0
f (x::xs) = x + f xs
f []      = 0
```

  või

```
f : List Double → Double
f = sum ∘ takeWhile (≠0)
```