Lambda račun

Funkcijski predpis

V matematiki poznamo zapis za *funkcijski predpis*:

`x ↦ e`

To preberemo "`x` se slika v `e`", pri čemer je `e` neki izraz, ki lahko vsebuje `x`. Primer:

`x ↦ x² + 3·x + 7`

Kadar imamo funkcijski predpis, ga lahko *uporabimo* na argumentu. Denimo, če je

`f := (x ↦ x² + 3·x + 7)`

Kadar pišemo

`f(x) := x² + 3·x + 7`

je to v bistvu okrajšava za `f := (x ↦ x² + 3·x + 7)`.

Funkcijski predpis lahko *uporabimo* na argumentu. Na primer, `f` lahko uporabimo na `3` in dobimo izraz `f(3)`, ki mu pravimo *aplikacija*.

Pravzaprav ni nobene potrebe, da funkcijski predpis poimenujemo `f`, lahko bi ga kar neposredno uporabljali in tvorili aplikacijo:

`(x ↦ x² + 3·x + 7)(3)`

To se morda zdi nenavadno, a je lahko koristno v programiranju (kot bomo videli kasneje). Nekateri programski jeziki imajo funkcijske predpise:

- Python: `lambda x: x**2 + 3*x + 7`
- Haskell: `\x -> x**2 + 3*x + 7`
- OCaml: `fun x -> x*x + 3*x + 7`
- Racket: `(lambda (x) (+ (* x x) (* 3 x) 7))`
- Mathematica: `#² + 3*# + 7 &` ali `Function[x, x² + 3*x + 7]`

Računsko pravilo (β-redukcija) za funkcijski predpis

Vsi znamo računati s funkcijskimi predpisi in aplikacijami, čeprav se tega morda ne zavedamo. Računsko pravilo, ki se iz zgodovinski razlogov imenuje "β-redukcija", pravi

$(x \mapsto e_1)(e_2) \;=\; e_1[x \leftarrow e_2]$

in ga preberemo:

Če uporabimo funkcijski predpis $x \mapsto e_1$ na argumentu e_2, dobimo izraz e_1, v katerem x zamenjamo z e_2.

Zamenjavi spremenljivke x za neki izraz pravimo *substitucija*. Primer:

$(x \mapsto x^2 + 3 \cdot x + 7)(3) = 3^2 + 3 \cdot 3 + 7$

Pozoro: pravilo za funkcijski zapis *ne* trdi $(x \mapsto x^2 + 3 \cdot x + 7)(3) = 25$, ampak samo, da lahko x zamenjamo s 3 in dobimo $3^2 + 3 \cdot 3 + 7$. Torej se pogovarjamo o računskih pravilih, ki so bolj osnovna kot računanje s števili!

Vezane in proste spremenljivke

V funkcijskem predpisu

$x \mapsto x^2 + 3 \cdot x + 7$

se x imeuje *vezana* spremenljivka. S tem želimo povedati, da je x veljavna samo znotraj funkcijskega predpisa, je kot neke vrste lokalna spremenljivka. Če jo preimenujemo, se funkcijski zapis ne spremeni:

$a \mapsto a^2 + 3 \cdot a + 7$

Poudarimo, da štejemo funkcijska predpisa za enaka, če se razlikujeta le po tem, kateri simbol je uporabljen za vezano spremenljivko.

V funkcijskem predpisu lahko nastopa tudi kaka dodatna spremenljivka, ki ni vezana. Pravimo ji *prosta* spremenljivka, na primer:

$x \mapsto a \cdot x^2 + b \cdot x + c$

Tu so a, b in c proste spremenljivke. Teh ne smemo preimenovati, ker bi se pomen izraza spremenil, če bi to storili. (Pravzaprav imamo še proste spremenljivke $\cdot$, + in ^{2}!)

Vezane in proste spremnljivke se pojavljajo tudi drugje v matematiki in računalništvu:

- v integralu $\int (x^2 + a \cdot x)\, dx$ je x vezana spremenljivka, a je prosta
- v vsoti $\sum_i i*(i-1)$ je i vezana spremenljivka
- v limiti $\lim_{x \to a} (x - a)/(x + a)$ je x vezana spremenljivka, a je prosta
- v formuli $\exists x \in R . x^3 = y$ je x vezana spremenljivka, y je prosta

- v programu

  ```
  for (int i = 0; i < 10; i++) {
      s += i;
  }
  ```

  je i vezana spremenljivka, s je prosta.

- v programu

```
if (false) {
   int s = 0 ;
   for (int i = 0; i < 10; i++) {
      s += i;
   }
}
```

sta `s` in `i` vezani spremenljivki.

Proste spremenljivke se ne smejo ujeti

Kadar imamo proste in vezane spremenljivke, moramo paziti, da se prosta spremenljivka ne "ujame", kar pomeni, da bi zaradi preimenovanja vezane spremenljivke prosta spremenljivka postala vezana. Na primer:

1. `x ↦ a + x` – "prištej a"
2. `y ↦ a + y` – "prištej a"
3. `a ↦ a + a` – "podvoji"

Vidimo, da se je v tretjem primeru `a` "ujela", ko smo `x` preimenovali v `a`. Mimogrede, treba je ločiti med tema dvema izrazoma:

- `y ↦ a + y` – "prištej a"
- `a ↦ a + y` – "prištej y"

Gnezdeni funkcijski predpisi

Funkcijske predpise lahko gnezdimo, ali jih uporabljamo kot argumente. Primeri:

1. `(x ↦ (y ↦ x · x + y))(42) = (y ↦ 42 · 42 + y)`
2. `((x ↦ (y ↦ x · x + y))(42))(1) = (y ↦ 42 · 42 + y)(1) = 42 · 42 + 1`
3. `(f ↦ f (f (3))) (n ↦ n · n + 1) = (n ↦ n · n + 1) ((n ↦ n · n + 1) (3)) = (n ↦ n · n + 1) (3 · 3 + 1) = (3 · 3 + 1) · (3 · 3 + 1) + 1`

Lahko se zgodi, da se zaradi vstavljanja enega funkcijskega predpisa v drugega kakšna vezana spremenljivka ujame. V takem primeru predhodno preimenujemo vezano spremenljivko. Primer:

- pravilno: `(x ↦ (y ↦ x · y²)) (z + 1) = (y ↦ (z + 1) · y²)`
- narobe: `(x ↦ (y ↦ x · y²)) (y + 1) = (y ↦ (y + 1) · y²)`
- pravilno: `(x ↦ (y ↦ x · y²)) (y + 1) = (x ↦ (a ↦ x · a²)) (y + 1) = (a ↦ (y + 1) · a²)`

λ-račun

Zapis `x ↦ e` postane dolgovezen, ko funkcijske zapise gnezdimo. Uporabili bomo *λ-zapis*:

```
λ x . e
```

To je prvotni zapis funkcijskih predisov, kot ga je zapisal Alonzo Church, vaš akademski praded! Temu zapisu pravimo *abstrakcija* izraza `e` glede na spremenljivko `x`.

Poleg tega bomo aplikacijo `f(x)` pisali brez oklepajev `f x`. Seveda pa oklepaje dodamo, kadar bi lahko prišlo do zmede. Dogovorimo se, da je aplikacija *levo asociativna*, torej

```
e₁ e₂ e₃  =  (e₁ e₂) e₃
```

V abstrakciji λ vedno veže največ, kolikor lahko. Torej je `λ x . e₁ e₂ e₃` je enako `λ x . (e₁ e₂ e₃)` in ni enako `(λ x . e₁) e₂ e₃`.

Kadari imamo gnezdene abstrakcije

```
λ x . λ y . λ z . e
```

to pomeni `λ x . (λ y . (λ z . e))`. Dogovorimo se še, da lahko tako gnezedeno abstrakcijo krajše zapišemo

```
λ x y z . e
```

Evaluacijske strategije

Pravilo za računanje lahko uporabimo na različne načine. Primer:

```
(λ x . (λ f . f x) (λ y . y)) ((λ z . g z) u)
```

je enak

```
(λ x . (λ f . f x) (λ y . y)) (g u)
```

in prav tako

```
(λ x . (λ y . y) x) ((λ z . g z) u)
```

Vendar pa velja lastnost *konfluence*, ki pravi, da vrstni red računanja ni pomemben. Natančneje, če ima `e` dva možna računska koraka, `e ↦ e₁` in `e ↦ e₂`, potem lahko v `e₁` in v `e₂` izvedemo take računske korake, da se bosta pretvorila v isti izraz.

V zgornjem primeru:

```
(λ x . (λ f . f x) (λ y . y)) (g u) =
(λ x . (λ y . y) x) (g u)  =
(λ x . x) (g u) =
g u
```

in

```
(λ x . (λ y . y) x) ((λ z . g z) u) =
(λ x . x) ((λ z . g z) u) =
(λ z . g z) u =
g u
```

Dobili smo izraz, v katerem ne moremo več narediti računskega koraka. Pravimo, da je tak izraz v *normalni* obliki.

Postavi se vprašanje, kako sistematično računati. Poznamo nekaj strategij:

- **Neučakana (eager evaluation):** v izrazu e_1 e_2 najprej do konca izračunamo e_1 da dobimo λ x . e, nato do konca izračunamo e_2, da dobimo e2'' in šele nato vstavimo e_2' v e.
- **Lena (lazy evaluation):** v izrazu e_1 e_2 najprej izračunamo e_1, da dobimo λ x . e, nato pa takoj vstavimo e_2 v e.

Poleg tega lahko računamo znotraj abstrakcij ali ne. Programski jeziki znotraj abstrakcij ne računajo (to bi pomenilo, da se računa telo funkcije, še preden smo funkcijo poklicali).

Programiranje v λ-računu

λ-račun je splošen programski jezik, ki je po moči ekvivalenten Turingovim strojem. Ogledamo si nekaj primerov.

Identiteta

```
id := λ x . x
```

Kompozicija

```
compose := λ f g x . g (f x)
```

Konstantna funkcija

```
const := λ c x . c
```

Boolove vrednosti in pogojni stavek

```
true := λ x y . x
false := λ x y . y
if := λ b t e . b t e
```

Urejeni pari

```
pair := λ a b . λp . p a b ;
first := λ p . p (λx y . x) ;
second := λ p . p (λx y. y) ;
```

Ostale primere si ogledamo v PL Zoo, programski jezik `lambda`.