Učilnica FRI 17/18 JERNEJ VIVOD

Rešitve

Na teh vajah bomo uporabljali λ-račun brez tipov kot programski jezik. V pomoč vam bo λ-račun v brskalniku, ki je implementacija jezika lambda iz Programming languages Zoo v brskalniku. (V repozitoriju `repl-in-browser` najdete tudi kodo za λ-račun v brskalniku.)

Naloga: λ-račun v brskalniku

Odprite λ-račun v brskalniku in se ga naučite uporabljati. Preizkusite kak izraz s predavanj.

Namesto `λ` je treba pisati `^` ali `\`. Če pišete ukaze v zgornji urejevalnik, jih morate med seboj ločiti s `;`.

Naloga: boolove vrednosti in logični vezniki

Ponovimo, kako v λ-računu implementiramo boolove vrednosti. Potrebujemo izraze `true`, `false` in `if`, da za vse `x` in `y` velja

```
if true x y ≡ x
if false x y = y
```

Na predavanjih smo jih definirali takole:

```
true  := λ x y . x ;
false := λ x y . y ;
if := λ p x y . p x y ;
```

Ideja je naslednja: `true` in `false` sta podatka, ki nam povesta, kako se odločimo med dvema možnostma. Torej lahko nanju gledamo kot na funkciji, ki sprejmeta obe možnosti `x` in `y`, nato pa vrneta tisto, ki sta jo izbrala. Pogojni stavek `if` je preprost, saj enostavno aplicira pogoj `p` na obeh možnostih, pogoj `p` pa izbere pravo.

S pomočjo pogojnega stavka lahko definiramo osnovne logične veznike `and`, `or`, `imply` in `not`. Skupaj naredimo `and`, ostale boste naredili sami. Začnemo z resničnostno tabelo za `and`:

```
and false false ≡ false
and  true false ≡ false
and false  true ≡ false
and  true  true ≡ true
```

Te enačbe lahko zapišemo bolj učinkovito takole:

```
and true  y ≡ y
and false y ≡ false
```

kar lahko implementiramo s pogojnim stavkom:

```
and' := λ x y . if x y false
```

Če upoštevamo definicijo `if`, lahko to še poenostavimo:

```
and := λ x y . x y false
```

Ta definicija neposredno uporablja dejstvo, da je logična vrednost `x` funkcija, ki izbere eno od dveh možnosti. V izrazu `x y false` tako `x` izbere bodisi `y` bodisi `false`. Če je `x ≡ true`, potem izbere `y`. Če pa je `x ≡ false`, potem izbere `false`.

Naloga: definirajte še disjunkcijo `or`, implikacijo `imply`, ekvivalenco `iff` in negacijo `not`. Rešitve poskusite zapisati brez uporabe `if`.

Rešitev

```
-- Vezniki z uporabo if
not'   := ^ p . if p false true ;
and'   := ^ p q . if p q flase ;
or'    := ^ p q . if p true q ;
imply' := ^ p q . if p q true ;
iff'   := ^ p q . and' (imply' p q) (imply' q p) ;

-- Vezniki brez uporabe if
not   := ^ p . p false true ;
and   := ^ p q . p q false ;
or    := ^ p q . p true q ;
imply := ^ p q . p q true ;
iff   := ^ p q . (p q true) (q p true) false ;
```

Za `iff` je možnih več rešitev, na primer `^ p q . if p q (not q)`.

Scott-Churchova števila

Na predavanjih smo spoznali **Churchova števila**: število `n` predstavimo kot funkcijo, ki sprejme funkcijo `f` in vrne `n`-kratni kompozitum `f ∘ f ∘ ⋯ ∘ f`:

```
0' := λ x f . x ;
1' := λ x f . f x ;
2' := λ x f . f (f x) ;
3' := λ x f . f (f (f x)) ;
4' := λ x f . f (f (f (f x))) ;
5' := λ x f . f (f (f (f (f x)))) ;
```

Če želimo neko funkcijo `f` uporabiti `n'`-krat na argumentu `x`, enostavno zapišemo `n' f x`. Danes Churchovih števil ne bomo uporabljali.

Spoznajmo še **Scott-Churchova** števila, ki imajo sicer bolj zapleteno definicijo, a je z njimi lažje programirati. Definirana so takole:

```
0 := λ x f . x ;
1 := λ x f . f 0 x ;
2 := λ x f . f 1 (f 0 x) ;
3 := λ x f . f 2 (f 1 (f 0 x)) ;
4 := λ x f . f 3 (f 2 (f 1 (f 0 x))) ;
5 := λ x f . f 4 (f 3 (f 2 (f 1 (f 0 x)))) ;
```

Število `n` smo spet predstavili kot `n`-kratno uporabo funkcije, le da funkciji dodatno podamo še *števec*, ki ji pove, katera po vrsti je.

Primitivna rekurzija

Denimo, da bi radi implementirali rekurzivno funkcijo `g`, ki pri argumentu `0` vrne vrednost `x`,

```
 g 0 ≡ x
```

pri argumentu `n+1` pa naredi en rekurzivni na predhodniku `n`:

```
 g (n+1) ≡ f n (g n)
```

Pomožna funkcija `f` izračuna končni odgovor s pomočjo `n` in rezultata rekurzivnega klica `g n`. Taki rekurziji pravimo *primitivna* ali *strukturna rekurzija* (poznamo tudi *splošno rekurzijo*, v kateri lahko `g` naredi povsem poljubne rekurzivne klice).

Mnoge funkcije lahko izračunamo s pomočjo primitivne rekurzije. Na primer, faktorielo

```
n! = 1 · 2 · ⋯ · n
```

definiramo takole:

```
fact 0 ≡ 1
fact (n+1) ≡ (n+1) · fact n
```

Vidimo, da je baza rekurzije `x ≡ 0`. Rekurzivni korak moramo še predelati v obliko

```
fact (n+1) ≡ f n (fact n)
```

za ustrezno izbrani `f`, ki je v tem primeru

```
f ≡ λ m r . (m+1) · r
```

Res dobimo

```
fact (n+1) ≡ f n (fact n)
           ≡ (λ m r . (m+1) · r) n (fact n)
           ≡ (λ r . (n+1) · r) (fact n)
           ≡ (n+1) · fact n
```

Scott-Churchova števila omogočajo računanje funkcij, ki so definirane s primitivno rekurzijo. Če je `g` definirana kot

```
g 0 ≡ x
g (n+1) ≡ f n (g n)
```

jo s Scott-Churchovimi števili definiramo kot

```
λ n . n x f
```

Poglejmo, kako to deluje na primeru `n ≡ 4`. Iz definicije `g` sledi:

```
g 4 ≡ f 3 (g 3)
    ≡ f 3 (f 2 (g 2))
    ≡ f 3 (f 2 (f 1 (g 1)))
    ≡ f 3 (f 2 (f 1 (f 0 (g 0))))
    ≡ f 3 (f 2 (f 1 (f 0 x)))
```

Po drugi strani pa iz definicije števila `4` sledi

```
4 x f ≡ (λ y h . h 3 (h 2 (h 1 (h 0 y)))) x f
      ≡ f 3 (f 2 (f 1 (f 0 x)))
```

Primitivno rekurzivne funkcije bomo torej v splošnem implementirali kot

```
λ n . n x (λ m r . …)
```

kjer je `x` baza rekurzije, funkcija `λ m r . …` pa pomožna funkcija, ki pove, kako iz rezultata rekurzivnega klica `r` pri argumentu `m` izračunamo rezultat pri `m+1`.

Predhodnik

Posebej preprost primer primitivne rekurzije je funkcija predhodnik `pred`. Dogovorimo se, da za predhodnik števila `0` vzamemo kar `0` (v λ-računu ne moremo javiti napake). Primitivno rekurzivna definicija se glasi:

```
pred 0 ≡ 0
pred (n+1) ≡ n
```

Morda bi kdo rekel, da ta definicija sploh ni rekurzivna. A lahko si mislimo, da je rekurzivna, le da rezultat rekurzivnega klica zavržemo:

```
pred 0 ≡ 0
pred (n+1) ≡ (λ m r . m) n (pred n)
```

Torej lahko `pred` zapišemo kot

```
pred ≡ λ n . n 0 (λ m r . m)
```

Preizkusimo, kako to deluje v `lambda`. Najprej definiramo nekaj Scott-Churchovih števil (kodo sproti preizkušajte v brskalniku):

```
0 := ^ x f . x ;
1 := ^ x f . f 0 x ;
2 := ^ x f . f 1 (f 0 x) ;
3 := ^ x f . f 2 (f 1 (f 0 x)) ;
4 := ^ x f . f 3 (f 2 (f 1 (f 0 x))) ;
5 := ^ x f . f 4 (f 3 (f 2 (f 1 (f 0 x)))) ;
```

Nato definiramo `pred`,

```
pred := λ n . n 0 (λ m r . m)
```

in preizkusimo:

```
lambda> pred 4
λ x f . f 2 (f 1 (f 0 x))
```

To ni preveč čitljivo, čeprav se vidi, da smo dobili `3`. Definirajmo si pomožno funkcijo `show`, ki število `n` predela v `n`-kratno uporabo konstante `S` na konstanti `Z`

```
:constant S
:constant Z
show := ^ n . n Z (^ m r . S r) ;
```

in poskusimo še enkrat:

```
lambda> show (pred 4)
S ((λ _ r . S r) 1 ((λ _ r . S r) 0 Z))
```

To je še slabše, a če vklopimo neučakano računanje, da bo `lambda` vse izračunal do konca, dobimo želeni učinek:

```
lambda> :eager
I will evaluate eagerly.
lambda> show (pred 4)
S (S (S Z))
```

Ko boste preizkušali vaše rešitve, uporabljajte `show`.

Naloga: naslednik

Kaj pa funkcija naslednik `succ`? Poizkusimo:

```
succ 0 ≡ 1
succ (n+1) = (n+2)
```

Težava je v tem, da nimamo definicije seštevanja. Funkcijo naslednik moramo implementiramo neposredno, izhajajoč iz definicije Scott-Churchovih števil:

```
succ n  ≡  λ x f . f n (f (n-1) (⋯ f 1 (f 0 x) ⋯)
```

Torej je

```
succ n  ≡  λ x f . f n (n x f)
```

Naloga: Definirajte `succ` v jeziku `lambda` in ga preizkusite.

Rešitev:

```
succ := ^ n . ^ x f . f n (n x f) ;
```

Naloga: seštevanje

Peanovi aksiomi za seštevanje se glasijo:

```
0 + k ≡ k
succ n + k ≡ succ (n + k)
```

To je primitivna rekurzija v prvem argumentu, kar je razvidno, če namesto `a + b` pišemo `plus a b`:

```
plus 0 k ≡ k
plus (n+1) k ≡ succ (plus n k)
```

Predelajmo jo tako, da je razvidna pomožna funkcija `f`:

```
plus 0 k ≡ k
plus (n+1) k ≡ (λ m r . succ r) n (plus n k)
```

Jezik `lambda` nam omogoča, da namesto `plus a b` pišemo `+ a b`, kar bomo tudi naredili:

```
+ 0 k ≡ k
+ (n+1) k ≡ (λ m r . succ r) n (+ n k)
```

Naloga: V `lambda` definirajte funkcijo `+` in jo preizkusite.

Rešitev

```
+ := ^ n k . n k (^ m r . succ r) ;
```

Naloga: množenje

Peanovi aksiomi za množenje se glasijo

```
0 · k ≡ 0
(succ n) · k ≡ k + n · k
```

Naloga: V `lambda` definirajte množenje `*` in ga preizkusite. Seveda je treba namesto `a * b` pisati `* a b`. Kako veliko število lahko izračunate, ne da bi vaš brskalnik pokleknil pod težo izredno neučinkovite implementacije aritmetike? (Če boste izzivali vaš brskalnik, da bo crknil, si rešitve najprej shranite v ločeno datoteko.)

Rešitev

```
* := ^ n k . n 0 (^ m r . + k r) ;
```

Naloga: odštevanje

Odštevanje lahko rekurzivno definiramo s pomočjo predhodnika. Dogovorimo se, da je `n - m` enak `0`, če je `m` večji od `n`. Tedaj dobimo:

```
n - 0 ≡ n
n - (k+1) ≡ pred (n - k)
```

Tokrat imamo primitivno rekurzijo na drugem argumentu, saj se v rekurzivnem klicu zmanjša `k`.

Naloga: V `lambda` definirajte odštevanje in ga preizkusite.

Rešitev

```
- := ^ n k . k n (^ m r . pred r) ;
```

Primerjava z `0`

Tudi predikate lahko definiramo s primitivno rekurzijo. Na primer, da želimo funkcijo `iszero`, ki vrne `true`, če je njen argument `0`, sicer `false`:

```
iszero 0 ≡ true
iszero (n+1) ≡ false
```

Tudi to je primitivna rekurzija, čeprav rekurzivnega klica ne vidimo! Lahko si mislimo, da je bil rekurzivni klic zavržen:

```
iszero 0 ≡ true
iszero (n+1) ≡ (λ m r . false) n (iszero n)
```

Od tod dobimo definicijo v jeziku `lambda`:

```
iszero := ^ n . n true (^ m r . false) ;
```

Relacija `≤`

Sedaj smo že zgradili nekaj osnovnih funkcij, s pomočjo katerih lahko definiramo nove, ne da bi vedno znova uporabljali primitivno rekurzijo. Denimo, relacijo `a ≤ b` lahko sestavimo s pomočjo odštevanja in `iszero`, ker velja

```
a ≤ b  ⇔  a-b = 0
```

Pri tem smo s pridom upoštevali dejstvo, da smo definirali `a - b = 0`, če je `a < b`.

Naloga: v `lambda` definirajte funkcijo `<=`, tako da je `<= a b` enako `true`, če je `a` manjši ali enak `b`, sicer pa `false`. Uporabite že prej definirani funkciji `-` in `iszero`.

Rešitev

```
<= := ^ n m . iszero (- n m) ;
```

Relaciji `<` in `=`

Definirajte še funkcijo `<`, ki računa relacijo "strogo manjše". Namig:

```
a < b  ⇔  a+1 ≤ b
```

Definirajte tudi funkcijo `==`, ki ugotovi, ali sta števili enaki.

Rešitev:

```
< := ^ n m . <= (succ n) m ;
== := ^ n m . and (<= n m) (<= m n) ;
```

Naloga: iskanje števila, ki zadošča pogoju

Denimo, da imamo predikat `p`, ki za vsako število vrne `true` ali `false` in da želimo med števili `1`, `2`, …, `n-1` poiskati največje, ki zadošča `p`. Če takega števila ni, vrnemo `0`. To lahko izrazimo z rekurzivno funkcijo `find`:

- `find p 0` je enako `0`, ker ni števila med `1` in `-1`

- `find p (n+1)` je enak `n`, če velja `p n` enako `true`
- `find p (n+1)` je enak `find p n k`, če ne velja `p n`

Naloga: Definirajte funkcijo `find`, ki sprejme predikat `p` in število `n`. Funkcija vrne največje izmed števil `1`, `2`, …, `n-1`, ki zadošča pogoju `p`. Če takega števila ni, naj vrne `0`. Funkcijo preizkusite na naslednjih primerih:

- med števili od `1` do `5` poišči največje število, ki je manjše od `3`:

```
lambda> show (find (^ k . < k 3) 5)
S (S Z)
```

- med števili od `1` do `5` poišči (največje) število, katerega kvadrat je enak devet:

```
lambda> show (find (^ k . == (* k k) (+ 4 5)) 5)
S (S (S Z))
```

Rešitev:

```
find := ^ p n . n 0 (^ m r . if (p m) m r) ;
```

Naloga: deljenje

Definirajte funkcijo `/`, ki deli naravna števila. Če se deljenje ne izide, naj funkcija vrne `0`. Pri tej nalogi se je lahko zmotiti, da deljenje z `1` ne deluje pravilno, zato vsekakor preizkusite `/ 5 1`, ki mora vrniti `5`.

Rešitev:

```
/ := ^ n m . find (^ k . == (* m k) n) (succ n) ;
```