Logika kombinátorov$^{(SK)}$

Už dávnejšie sme sa stretli s tromi dôležitými príkladmi:

- S = λxyz.(xz)(yz)
- K = λxy.x
- I = λx.x

inými slovami:

S f g x Ÿ (f x) (g x)
K c x Ÿ c
I x Ÿ x

pričom vieme, že platí (veta o zbytočnosti identity I):

- S K K = I
- S K S = I -- toto sme možno netušili, tak si to dokážme...

S K K x = ((S K) K) x = (K x) (K x) = x
S K S x = ((S K) S) x = (K x) (S x) = x

Logika kombinátorov(SK)

Ukážeme, že ľub. uzavretý λ-výraz (neobsahujúci voľné premenné) vieme prepísať pomocou kombinátorov S, K, (I) tak, že zápis neobsahuje abstrakciu (t.j. neobsahuje premenné (pozn. ktoré robia problémy v interpreteri)).

data Ski = S | K | I | APL Ski Ski

Intuitívne smerujeme k transformácii LExp -> Ski (pre uzavreté λ-termy).

Kým v λ-teórii sme mali hlavné pravidlo β-redukcie, v Ski-teórii máme tri nové redukčné pravidlá (vlastne definície operátorov):

- S-redukcia   S f g x   Ÿ (f x) (g x)
- K-redukcia   K c x   Ÿ c
- I-redukcia   I x   Ÿ x

S = λxyz.(xz)(yz)
K = λxy.x
I = λx.x

S, K, (I) transformácie

Transformácia do SK(I)

- λx.x Ÿ I = S K K
- λx.c Ÿ K c ak x ∉ Free(c)
- λx.(M N) Ÿ S (λx.M) (λx.N)

poslednú rovnosť si overme:

- λx.(M N) y Ÿ (M[x:y] N[x:y])
- S (λx.M) (λx.N) y Ÿ (λx.M y) (λx.N y) Ÿ (M[x:y] N[x:y]) platí...☺

Zmyslom S, K, (I) transformácií je eliminácia abstrakcie,
t.j. dostaneme výraz neobsahujúci abstrakciu, viazané premenné.

Skôr, než si to sami naprogramujete, vyskúšajte

http://tromp.github.io/cl/cl.html

Príklad transformácie

Transformácia do SK(I)

λx.x	Ÿ	I
λx.c	Ÿ	K c
λx.(M N)	Ÿ	S (λx.M) (λx.N)

- λx.λy.(y x) Ÿ $_{S}$
- λx.(S (λy.y) (λy.x)) Ÿ $_{I}$
- λx.(S I (λy.x)) Ÿ $_{K}$
- λx.(S I (K x)) Ÿ $_{S}$
- S (λx.(S I)) (λx.(K x)) Ÿ $_{K}$
- S (K (S I)) (λx.(K x)) Ÿ $_{S}$
- S (K (S I)) (S (λx.K) (λx.x)) Ÿ $_{K}$
- S (K (S I)) (S (K K) (λx.x)) Ÿ $_{I}$
- S (K (S I)) (S (K K) I). = ((S (K (S ((S K) K)))) ((S (K K)) ((S K) K)))

Väčší príklad:

- Y Ÿ

((S ((S ((S (K S)) ((S (K K)) I))) (K ((S I) I)))) ((S ((S (K S)) ((S (K K)) I))) (K ((S I) I))))

Výpočet v SK(I) teórii

S-redukcia	((S f) g) x	Ÿ	(f x) (g x)
K-redukcia	(K c) x	Ÿ	c
I-redukcia	I x	Ÿ	x

((λx.λy.(y x) 5) (+1)) – pri výpočte nemáme viazané premenné, t.j. nepotrebujeme pojem substitúcie

S (K (S I)) (S (K K) I) 5 (+1) Ÿ $_S$

((K (S I)) 5) ((S (K K) I) 5) (+1) Ÿ $_K$

(S I) ((S (K K) I) 5) (+1) Ÿ $_S$

(I (+1)) ((S (K K) I) 5) (+1) Ÿ $_I$

(+1) ((S (K K) I) 5) (+1) Ÿ $_S$

(+1) (((K K) 5) (I 5) (+1)) Ÿ $_I$

(+1) (((K K) 5) 5 (+1)) Ÿ $_K$

(+1) (K 5 (+1)) Ÿ $_K$

(+1) 5 Ÿ $_+$

6

Domáca úloha

Naprogramujte konvertor do SK(I)
Naprogramujte SK(I) redukčný stroj

Implementačná poznámka:

Funkcia LExp Ÿ Ski sa zle píše, lebo proces transformácie do SKI má medzistavy, keď výraz už nie je LExp a ešte nie je Ski.

```
data LExpSki =
        LAMBDA String LExp |
        ID String |
        APL LExp LExp |
        CON String |
        CN Integer |
        S | K | I
        deriving(Show, Read, Eq)
toSki   :: LExpSki -> LExpSki
```

Vlastnosti operátorov
(príklady SKI výrazov)

Komutatívnosť operátora f:

Ak C f x y = f y x, potom musí platiť, že C f = f (aby f bola komutatívna oper.)

- v λ-teórii je C = λf.λx.λy.(f y) x
- v SKI-teórii
  C = ((S ((S (K S)) ((S (K K)) ((S (K S)) ((S ((S (K S)) ((S (K K)) I))) (K I)))))) (K ((S (K K)) I)))

naozaj: ((S ((S (K S)) ((S (K K)) ((S (K S)) ((S ((S (K S)) ((S (K K)) I))) (K I)))))) (K ((S (K K)) I))) f a b Ÿ ((f b) a)

Asociatívnosť operátora f:

A_1 f = A_2 f

A_1 f x y z = (f (f x y) z) = λf.λx.λy.λz.(f (f x y) z) =

((S ((S (K S)) ((S (K (S (K S)))) ((S (K (S (K (S (K S)))))) ((S (K (S (K (S (K K)))))) ((S ((S (K S)) ((S (K K)) ((S (K S)) ((S (K K)) I))))) ((S ((S (K S)) ((S (K (S (K S)))) ((S (K (S (K K)))) ((S ((S (K S)) ((S (K K)) I))) (K I)))))) (K (K I))))))))) (K (K (K I))))

naozaj: A1 f a b c Ÿ ((f((fa)b))c)

A_2 f x y z = (f x (f y z)) = λf.λx.λy.λz.(f x (f y z)) =

((S ((S (K S)) ((S (K (S (K S)))) ((S (K (S (K K)))) ((S (K (S (K S)))) ((S (K (S (K K)))) ((S ((S (K S)) ((S (K K)) I))) (K I)))))))) ((S (K K)) ((S ((S (K S)) ((S (K (S (K S)))) ((S (K (S (K K)))) ((S ((S (K S)) ((S (K K)) I))) (K I)))))) (K (K I)))))

naozaj: A2 f a b c Ÿ ((fa)((fb)c))

Vlastnosti SK(I) teórie

S-redukcia	((S f) g) xŸ	(f x) (g x)
K-redukcia	(K c) x Ÿ	c
I-redukcia	I x Ÿ	x

SKI redukcie spĺňajú Church-Rosserovu vlastnosť, t.j.

SKI term má najviac jednu normálnu formu vzhľadom na redukcie S, K, I

Vážny problém: dve **rôzne** SKI-normálne formy predstavujú rovnaký λ-term

SKK = I = SKS

Existuje nekonečné odvodenie, Ω = ((S I I) (S I I)) Ÿ$_S$ ((I (S I I)) (I (S I I)))
Ÿ$_I$ ((S I I) (I (S I I))) Ÿ$_I$ ((S I I) (S I I))

Je systém SK minimálny ?

Či existuje verzia kombinátorovej logiky aj s jedným kombinátorom ?
Je to čisto teoretická otázka, v praxi potrebujeme opak...

Nech X = λx.(x K S K)

- potom vieme ukázať, že K = X X X = (X X) X
- A tiež, že S = X . X X = X (X X)

Skúste si to ako cvičenie...

Iná možnosť je, ak X = λx.((x S) K)

- potom K = X (X (X X))
- a S = X (X (X (X X)))

Skúste si to ako cvičenie...

Môže sa zdať, že ide o čisto teoretický výsledok, ale existuje programovací jazyk (Iota - pokročilé čítanie pre extrémisticky ladených nadšencov) používajúci X ako jedinú jazykovú konštrukciu.
http://semarch.linguistics.fas.nyu.edu/barker/Iota/

λx.(M N) Ÿ S (λx.M) (λx.N)

B, C kombinátory

Praktický problém pri používaní kombinátorov je v tom, že výsledné SK výrazy sú veľké, napr. λx.(+ 1) Ÿ S (K +) (K 1) pričom aj K (+ 1) by stačilo.

Problém je λ-abstrakcia pri S transformácii, ktorá sa množí do M aj N.

Špecializujme S kombinátor na dve verzie:

B = λxyz.x (yz)	B-redukcia B f g x = f (g x)
C = λxyz.(x z) y	C-redukcia C f g x = (f x) g

Následne potom zavedieme dve nové transformácie:

Transformácia do SK(I)BC

λx.x	Ÿ	I = S K K	
λx.c	Ÿ	K c	ak x ∉ Free(c)
λx.(M N)	Ÿ	S (λx.M) (λx.N)	ak x∈Free(M) & x ∈ Free(N)
λx.(M N)	Ÿ	B M (λx.N)	ak x∈Free(N) & x ∉ Free(M)
λx.(M N)	Ÿ	C (λx.M) N	ak x∈Free(M) & x ∉ Free(N)

Cvičenie: doprogramujte BC transformácie a BC redukcie do interpretera

B = λxyz.x (yz)   B-redukcia B f g x = f (g x)
C = λxyz.(x z) y   C-redukcia C f g x = (f x) g

Optimalizácia výsledného kódu

Problém SK termov je ich veľkosť, preto sa sústredíme na to, ako ju zmenšiť pri zachovaní jeho sémantiky.

Uvedieme niekoľko príkladov a z toho odvodených pravidiel:

λx.(+ 1) Ÿ S (K +) (K 1) Ÿ K (+ 1)

- **S (K p) (K q)   = K (p q)**

λx.- x Ÿ S (K -) I Ÿ B – I

- **S (K p) q   = B p q**
- **S (K p) I   = B p I = p**

- **S p (K q)   = C p q**

S p (K q) x Ÿ   C p q x Ÿ
(p x) (K q x) Ÿ   (p x) q
(p x) q

Simon Peyton Jones, 1987,
The Implementation of Functional Programming Languages
http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/

S,K,I,B,C kombinátory

- $\lambda x.x$ Ÿ I
- $\lambda x.C$ Ÿ $K\ C$ $\quad x \notin Free(C)$
- $\lambda x.(M\ N)$ Ÿ $S\ (\lambda x.M)\ (\lambda x.N)$
- $\lambda x.(M\ x)$ Ÿ M $\quad x \notin Free(M)$
- $S\ (K\ M)\ N$ Ÿ $B\ M\ N$
- $S\ M\ (K\ N)$ Ÿ $C\ M\ N$
- $S\ (K\ M)\ (K\ N)$ Ÿ $K\ (M\ N)$
- $S\ (K\ M)\ I$ Ÿ M

p^1 = toSki $\lambda x_1.p$, q^1 = toSki $\lambda x_1.q$
p^2 = toSki $\lambda x_2.\lambda x_1.p$, q^2 = toSki $\lambda x_2.\lambda x_1.q$
…

S′ kombinátor

- $\lambda x_n \ldots \lambda x_3.\lambda x_2.\lambda x_1.(p\ q)$ Ÿ
- $\lambda x_n \ldots \lambda x_3.\lambda x_2.(S\ p^1\ q^1)$ Ÿ
- $\lambda x_n \ldots \lambda x_3.(S\ (B\ S\ p^2)\ q^2)$ Ÿ
- $\lambda x_n \ldots \lambda x_4.(S\ (B\ S\ (B\ (B\ S)\ p^3))\ q^3)$ Ÿ
- $\lambda x_n \ldots \lambda x_5.(S\ (B\ S\ (B\ (B\ S)\ (B\ (B\ (B\ S))\ p^4)))\ q^4)$ Ÿ rastie kvadraticky od n

často vyskytujúci sa vzor S (B x y) z nahradíme novým kombinátorom S′ x y z
teda $\underline{S\ B}$ nahradíme S′

$\lambda x_n \ldots \lambda x_4.(\underline{S\ (B}\ S\ (B\ (B\ S)\ p^3))\ q^3)$ Ÿ
$\lambda x_n \ldots \lambda x_4.(S'\ (\underline{S\ (B}\ (B\ S)\ p^3))\ q^3)$ Ÿ
$\lambda x_n \ldots \lambda x_4.(S'\ (S'\ (B\ S)\ p^3)\ q^3)$ Ÿ B f g x = f (g x)
$\lambda x_n \ldots \lambda x_4.(S'\ (S'\ S)\ p^3\ q^3)$

- $\lambda x_n \ldots \lambda x_3.\lambda x_2.\lambda x_1.(p\ q)$ Ÿ
- $\lambda x_n \ldots \lambda x_3.\lambda x_2.(S\ p^1\ q^1)$ Ÿ
- $\lambda x_n \ldots \lambda x_3.(S'\ S\ p^2\ q^2)$ Ÿ
- $\lambda x_n \ldots \lambda x_4.(S'\ (S'\ S)\ p^3\ q^3)$ Ÿ
- $\lambda x_n \ldots \lambda x_5.(S'\ (S'\ (S'\ S))\ p^4\ q^4)$ Ÿ rastie už len lineárne...

B = λxyz.x (yz) B-redukcia B f g x = f (g x)
C = λxyz.(x z) y C-redukcia C f g x = (f x) g

S′, B′, C′ kombinátory

často vyskytujúci sa vzor (S (B x y) z) nahradíme novým kombinátorom S′ x y z

Keď dosadíme S B, dostaneme S′ kombinátor ako

S′ c f g x = S (B c f) g x = (B c f x) (g x) = (c (f x)) (g x)

S′ c f g x = c (f x) (g x)

Analogicky potom zavedieme dva obmedzené kombinátory podľa vzoru B′, C′
B′ c f g x = c f (g x)
C′ c f g x = c (f x) g

Cvičenie:
Pridajte špecialne kombinátory pre IF, =, MOD, DIV a definujte rekurzívne NSD. Vypočítajte NSD 18 24.

Nerozhodnuteľnosť SK(I) teórie

B = λxyz.x (yz) B-redukcia B f g x = f (g x)
C = λxyz.(x z) y C-redukcia C f g x = (f x) g

Je nerozhodnuteľné, či SKI term má normálnu formu (pekný dôkaz viď
http://en.wikipedia.org/wiki/Combinatory_logic)

Nech existuje také N, čo počíta, či x má n.f. alebo nie:

(N x) => True, if x ak má normálnu formu [True = K]
F, inak. [False = (K I)]

Dar nebies: Z = (C (C (B N (S I I)) Ω) I)

(S I I Z) Ÿ $_{S}$

((I Z) (I Z)) Ÿ $_{I}$

(Z (I Z)) Ÿ $_{I}$

(Z Z) Ÿ $_{Z}$

(C (C (B N (S I I)) Ω) I Z) Ÿ $_{C}$

(C (B N (S I I)) Ω Z I) Ÿ $_{C}$

((B N (S I I) Z) Ω I) Ÿ $_{B}$

(N (S I I Z) Ω I)

Ak (N (S I I Z)) = True (má n.f.) tak výsledok je Ω, t.j. nemá

Ak (N (S I I Z)) = False (nemá n.f.) tak výsledok je I, t.j. má

Super-kombinátor

Uzavretý term $\lambda x_1 \lambda x_2 \ldots \lambda x_n.E$ je super-kombinátor, ak

- E nie je λ-abstrakcia
- všetky λ-abstrakcie v E sú super-kombinátory

Príklady:

- λx.x
- λx.λy.(- x y)
- λf.(f λx.(* x x))

Príklady termov, ktoré nie sú super-kombinátory:

- λx.y - nie je uzavretý
- λf.(f λx.(f x x)) - nie je λx.(f x x) super-kombinátor
- λx.(x (λy.y (λz.z y))) - nie je (λz.z y) super-kombinátor

Transformácia termu do super-kombinátorov

- λx.(<u>λy.(+ y x) </u>x) 4

λy.(+ y x) nie je super-kombinátor, ale (po η-konverzii) dostaneme

(λx.λy.(+ y x)) x (λ-lifting)

a R = (λx.λy.(+ y x)) je super-kombinátor, dosadíme R x, teda
λx.((R x) x) 4
Q = λx.(R x x) je super-kombinátor, takže celý program zredukujeme
Q 4
pričom
Q = λx.(R x x)
R = (λx.λy.(+ y x))

Rekurzia

Opäť ale máme problém s rekurziou:

1) riešenie pomocou Y operátora:
   - $f\ x = g\ (f\ (x-1))\ 0$
   - $f = \lambda F.\ \lambda x.(g\ (F\ (x-1))\ 0)\ f$
   - $f = Y\ (\lambda F.\ \lambda x.(g\ (F\ (x-1))\ 0))$

2) riešenie pomocou rekurzívnych super-kombinátorov

Lifting lambda

```
sumInts m      = suma (count 1) where
   count n | n > m   = []   -- lokálna definícia funkcie
   count n | otherwise      =  n : count (n+1)

suma []        = 0
suma (n:ns)    = n + suma ns
---------------------------- zavedieme let-in
sumInts' m     =
   let count' n = if n > m  then [] else n : count' (n+1)
   in  suma' (count' 1)

suma' ns = if ns == [] then 0 else (head ns) + suma' (tail ns)
---------------------------- prehodíme definíciu suma dovnútra
```

Lifting lambda / 1

```
sumInts'' m  =
   let
       count'' n = if n > m  then [] else n : count'' (n+1)
       suma'' ns =
          if ns == [] then 0 else (head ns) + suma'' (tail ns)
   in
       suma'' (count'' 1)
---------------------------- zavedieme λ abstrakcie
sumInts''' m =
   let
     count''' = \n->if n > m  then [] else n : count''' (n+1)
     suma'''  = \ns->if ns == [] then 0
                     else (head ns)+suma''' (tail ns)
   in suma''' (count''' 1)
---------------------------- count''' nie je super-kombinátor
```

Lifting lambda / 2

```
---------------------------- λ-lifting
sumInts'''' m =
   let
     count = \c->\m->\n->if n > m  then [] else n:c (n+1)
     count'''' = count count'''' m        -- rekurzia
     suma''''  = \ns -> if ns == [] then 0
                          else (head ns) + suma'''' (tail ns)
   in suma'''' (count'''' 1)
------------------------ definícia pomocou super-kombinátorov
sumInts''''' m =
   let
    sc1 =\c->\m->\n->if n > m  then [] else n : c (n+1)
    sc2 =\ns->if ns == [] then 0 else (head ns) + sc2 (tail ns)
    sc3 = sc1 sc3 m -- rekurzia
   in sc2 (sc3 1)
```