Czech Technical University
Faculty of Informatics
Department of Theoretical Informatics

BI-PPA 2017

Cvi Introduction to the λ-calculus

Date: 3. 11. 2017

What are we going to cover Order of evaluation (normal vs applicative) and the normal form, arithmetics and logic in λ-calculus, recursion and Y combinator. **Ex. 1**. From last exercise (normal vs applicative evaluation): Evaluate the expressions using normal and then applicative order:

a) (λa. λb. a) c ((λd. e) d) **Solution:** Normal ordering: (λa. λb. a) c ((λd.e) d) → (λb.a)[aλc] → ((λd.e) d) → λb.c ((λd.e) d) → (c)[bλ((λd.e) d)] → c
Applicative ordering: (λa. λb. a) c ((λd. e) d) → (λa.λb. a) c (e[dλd]) → (λa.λb.a) c (e)
(λa.λb.a) c (e) → (λb.a)[aλc] (e) → (λb.c) e
(λb.c) e → c[bλe] → c

b) (λx. a)((λx. x x)(λy. y y)) **Solution:** Normal ordering: (λx. a)((λx. x x)(λy. y y)) → a [xλ((λx. x x)(λy. y y))] → a
Applicative ordering: (λx. a)((λx. x x)(λy. y y)) → (λx . a)(xx)[x λ(ly. yy)]
→ (λx . a)((ly . yy) (ly . yy))→ (λx . a)((ly . yy) (ly . yy))
→ (λx . a)(yy)[y λ(ly . yy)] → (λx . a)((ly . yy) (ly . yy))

Representation of arithmetics We are going to remove everything that conflicts with the basic rules of λcalculus (see the first lesson), after which we are left with only λfunctions and variables. We use these to create the so called Church numerals. Every natural number n is represented by a function with two arguments. The first one will be repeated n times, the second argument is a form of a sentinel:

- 0 = (λs. (λz. z))
- 1 = (λs. (λz. (s z)))
- 2 = (λs. (λz. (s (s z))))
- 3 = (λs. (λz. (s (s (s z)))))

and all operations are defined as λ-functions. The most basic ones for successor and predecessor and operations +, * and − are defined below:

- x + 1: (λx. λs. λz. s (x s z))
- x − 1: (λx. λs. λz. x (λf.λg.g (f s)) (λg.z) (λm.m)) g.λk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0) 0
- x + y: (λx. λy. λs. λz. x s (y s z))
- x * y: (λx. λy. λz. x (y z)) or (λx. λy. λs. λz. x (y s) z)
- x − y: (λx. λy. λs. λz. y (x s) z), or (λa. λb. λs. λz. b a) or (λb. λa.a b)

Ex. 2. (hard): Rewrite expressions from the first exercise (except 147) into λ-calculus using only Church numerals and functions for operators +, −, *. .

Representation of Logic Similarly to numbers, we can represent basic logical operators using only λ-functions. We start by defining functions for true and false values, from which the rest follows. Function for boolean values also takes two arguments, where true returns the first argument and false returns its second argument.

- true (T) : (λt. λf. t)
- false (F) : (λt. λf. f)
- not: (λx . x (λt.λf. f) (λt.λf. t)) or (λx. λt. λf. x f t)
- and: (λx. λy. x y (λt. λf. f))
- or: (λx. λy. x (λt. λf. t) y)
- xor: (λx. λy. x (y (λt. λf. t) (λt. λf. f)) (y (λt. λf. t)(λt. λf. f)))

Both sections show how useful it is to start using symbollic names instead the λ-functions, i.e. numbers 0, 1, 2, . . . instead od Z obou partií je zřejmé, jak pohodlné je začít používat symbolické názvy místo λ-funkcí, tedy čísla 0, 1, 2, . . . instead of (λs. (λz. z)), (λs. (λz. sz)), . . . and similarly for logical operators:

- not: (λx . x false true) or (λx. λt. λf. x f t)
- and: (λx. λy. x y false)
- or: (λx. λy. x true y)
- xor: (λx. λy. x (y true false) (y true false)),
  or the obligatory ((x or y) and (not (x and y))) – try to write down with pure λfunctions

which makes the notation much easier to read. We are therefore going to use the symbollic notation in the future as well (it will be especially important for recursion. It is also obvious, that λcalculus required creation of symbollic languages to be practical, and Lisp is one of them.

Recursion Let us remember the following two functions: ((λx. xx)(λx. xx)) a ((λx. R(xx))(λx. R(xx))). What do they mean? An infinite loop. All we need now is a condition and we can use recursion.

- **Comparison with 0 (zero x)**: (λx. x false not false) = (λx. x (λt. λf. f) (λx. λt. λf. x f t) (λt. λf. f))
- **Condition (if b then x else y)**: (λb. λx. λy. b x y) . . . if b is true, x is applied, otherwise applies y.
- **Infinite loop - so called Y-combinator**: (λf. (λx. f (xx))(λx. f (xx))) . . . where f is any recurrent function, that we want to repeat infinitely.

We now have everything we need to play with recurrent expressions: **Factorial of n:**
First the recursive definition in C:

fac(n) { if (n = 0) then 1 else n* fac(n-1) },

which we rewrite into λ-calculus as

$$(\lambda f.\ \lambda n.\ zero\ n\ 1(*\ n(f(-\ n\ 1)))).$$

However this misses an application that would copy it, so we add a Y combinator:

$$Y(\lambda f.\ \lambda n.\ zero\ n\ 1\ (*\ n(f(-\ n\ 1)))) = (\lambda r.\ (\lambda x.\ r\ (xx))(\lambda x.\ r\ (xx)))(\lambda f.\ \lambda n.\ zero\ n\ 1(*\ n(f(-\ n\ 1))))$$

or shortened as Y R, where Y is the Y-combinator (λr. (λx. r (xx))(λx. r (xx)))
and R is a recursive factorial function (λf. λn. zero n 1 (* n (f (- n 1)))). In general, R can be any function. **Ex. 3**. Try writing λfunctions using the Y combinator for the following:

- nth square of number k using the equation $f(n) = k \cdot f(n-1)$, where, $f(1) = 1$. **Solution:** Y (λf. λn. zero n 1 (* f (-n 1) k))
- nth Fibonacci number, **Solution:** Y (λf. λn. zero n 1 (zero (- n 1) 1 (+ f (-n 1) f(- n 2))))
- Ackerman's function, defined as:

$$A(m,n) = \begin{cases} n+1, & m=0 \\ A(m-1,n), & n=0 \\ A(m-1, A(m,n-1)) & jinak. \end{cases}$$

  Solution: Y (λa. λm. λn. zero m (+ n 1) (zero n (a (- m 1) n) (a (- m 1) (a m (- n 1)))))
- recurrent function $f(n) = 3 \cdot f(n-2) + n$, where, $f(0) = 0, f(1) = 2$. **Solution:** Y (λf. λn. zero n 0 (zero (- n 1) 2 (+ f(- n 2) n)))

Defined abbreviations of Y and R will become useful in applying recursive functions, as is shown in the following, and last, example:

Application of a factorial function to number 4 (Y R 4) We are using symbolic names, but when evaluating, all are replaced with λ-functions with the exception of arithmetics:) Y R 4 → (λr.

(λx. r (xx))(λx. r (xx))) R 4 → (λx. R (xx))(λx. R (xx)) 4
→ R ((λx. R (xx))(λx. R (xx))) 4
→ R (Y R) 4 . . . we expand the first R into
→ (λf. λn. zero n 1 (* n (f (- n 1)))) (YR) 4 . . . then apply parenthesis (YR) to f
→ (λn. zero n 1 (* n ((YR) (- n 1)))) 4. . . now apply 4 to n
→ zero 4 1 (* 4 ((YR) (- 4 1))). . . and apply 4 to function zero, evaluate to false F
→ F 1 (* 4 ((YR) (- 4 1))). . . F means we do not continue 1, but with the next argument

→ (* 4 ((YR) (- 4 1)))) → (* 4 (R (Y R) (- 4 1)))) . . . again expand first R into
→ (* 4 ((λf. λn. zero n 1 (* n (f (- n 1)))) (YR) (- 4 1)))). . . apply (YR) to f
→ (* 4 ((λn. zero n 1 (* n ((YR) (- n 1)))) (- 4 1)))). . . (- 4 1) = 3 apply for n
→ (* 4 (zero 3 1 (* 3 ((YR) (- 3 1))))). . . zero 3 is false F
→ (* 4 (F 1 (* 3 ((YR) (- 3 1))))). . . so we evaluate the second option: * 3 (YR)...

→ (* 4 (* 3 ((YR) (- 3 1)))). . . again apply (YR) na f
→ (* 4 (* 3 (R (Y R) (- 3 1)))). . . expand left R
→ (* 4 (* 3 ((λf. λn. zero n 1 (* n (f (- n 1)))) (Y R) (- 3 1)))). . . apply (YR)
→ (* 4 (* 3 ((λn. zero n 1 (* n ((YR) (- n 1)))) (- 3 1)))). . . apply (- 3 1) = 2 to n
→ (* 4 (* 3 (zero 2 1 (* 2 ((YR) (- 2 1)))))). . . apply 2 to zero, which is false F
→ (* 4 (* 3 (F 1 (* 2 ((YR) (- 2 1)))))). . . so evaluate the second part again: * 2 (YR)...

→ (* 4 (* 3 (* 2 ((YR) (- 2 1))))). . . expand (YR) again
→ (* 4 (* 3 (* 2 (R (YR) (- 2 1))))). . . expand R
→ (* 4 (* 3 (* 2 ((λf. λn. zero n 1 (* n (f (- n 1)))) (YR) (- 2 1))))). . . apply (YR) to f
→ (* 4 (* 3 (* 2 ((λn. zero n 1 (* n ((YR) (- n 1)))) (- 2 1))))). . . apply (- 2 1) = 1 to n
→ (* 4 (* 3 (* 2 (zero 1 1 (* 1 ((YR) (- 1 1))))))). . . 1 applied to zero yields false F
→ (* 4 (* 3 (* 2 (F 1 (* 1 ((YR) (- 1 1))))))). . . so the second part is applied again: * 1 (YR)...

→ (* 4 (* 3 (* 2 (* 1 ((YR) (- 1 1)))))). . . (YR) applied yields R (YR)
→ (* 4 (* 3 (* 2 (* 1 (R (YR) (- 1 1)))))). . . R expands
→ (* 4 (* 3 (* 2 (* 1 ((λf. λn. zero n 1 (* n (f (- n 1)))) (YR) (- 1 1)))))). . . (YR) is applied to f
→ (* 4 (* 3 (* 2 (* 1 ((λn. zero n 1 (* n ((YR) (- n 1)))) (- 1 1)))))). . . (- 1 1) = 0 for n
→ (* 4 (* 3 (* 2 (* 1 (zero 0 1 (* 0 ((YR) (- 0 1)))))))). . . zero 0 is true T
→ (* 4 (* 3 (* 2 (* 1 (T 1 (* 0 ((YR) (- 0 1)))))))). . . so the first argument is used (i.e. applies 1, the second part disappears)
→ (* 4 (* 3 (* 2 (* 1 (1))))). . . arithmetics now
→ (* 4 (* 3 (* 2 (1))))→ (* 4 (* 3 (2))) → (* 4 (6)) → (24)