Lambda Calculus

Daniel Vlasits

Abstract

An introduction into the crazy world that is lambda calculus.

We will be looking at pure untyped lambda calculus. The aim of it was to develop a general theory of functions providing a foundation for logic and maths. It has introduced many powerful features of programming languages such as procedures as arguments and similarly data and algorithms being interchangeable.

Alonzo Church invented Lambda Calculus looking at what is the notion of a function from a computation perspective. PhD superviser of Alan Turing (invented Turing machine - state based model of computation) Church (basic functional notion of computation) - turned out to be equivalent For Church - function is a black box - takes input and provides output

1. Intro

3 Reasons why worth studying

1. Can encode any computation
2. Basis for functional programming (Haskell) basically compiled to a glorified lambda calc
3. Lambda syntax available in most languages

Definition 1.1. *Lambda Terms* are words over the following alphabet:

- $v_0, v_1, ...$ normal text
- λ abstractor
- (,) paranthesis

Defined inductively over the minimal class satisfying:

1. $x \in \Lambda$
2. $M \in \Lambda \implies (\lambda x M) \in \Lambda$
3. $M, N \in \Lambda \implies (MN) \in \Lambda$

Where

- $x, y, z, ...$ denote arbitrary variables
- Symbol $\equiv$ denotes syntactic equality

Definition 1.2. Equivalence Class on Λ

1. $(\lambda x.M)N = M[x := N]$ (β-conversion!)

2. $M = M$ (Reflexive)
3. $M = N, N = L \implies M = L$ (Transitive)
4. $M = N \implies MZ = NZ$
5. $M = N \implies ZM = ZN$
6. $M = N \implies \lambda x.M = \lambda x.N$ (Notice can expand as well!)

Definition 1.3. $FV(M)$ is the set of free variables in M and can be defined inductively as follows:

- $FV(x) = \{x\}$
- $FV(\lambda x.M) = FV(M) - \{x\}$
- $FV(MN) = FV(M) \cup FV(N)$

Things to note

1. M is *closed* or a *combinator* if $FV(M) = \emptyset$
2. $\Lambda^0 = \{M \in \Lambda |$ M is closed$\}$

If cannot do any further Beta-reductions it is in normal form.

Definition 1.4. The one thing I will give away: representing truth values:

$$\textbf{true} = \lambda xy.x$$

$$\textbf{false} = \lambda xy.y$$

See here that functions are being used to represent values! I will give some examples here of it being used

2. Exercise 1

We will now on paper / laptop proceed and derive as much as possible: Work in this in groups of 2 and see how far you can get

```
    def eb(bool): #eb stands for evalBool
        return bool(True)(False)
Write functions for these: you may of course use any functions as you go
```

1. negation

```
assert eb(negation(true)) == eb(false)
assert eb(negation(false)) == eb(true)
```

2. or - use name orOp (or is keyword in python)

```
assert eb(orOp(false)(false)) == eb(false)
assert eb(orOp(true)(false)) == eb(true)
assert eb(orOp(false)(true)) == eb(true)
assert eb(orOp(true)(true)) == eb(true)
```

3. and - use name andOp (and is keyword in python)

```
assert eb(andOp(true)(false)) == eb(false)
assert eb(andOp(false)(true)) == eb(false)
assert eb(andOp(true)(true)) == eb(true)
```

4. if then else i.e. a function ifte(bool, item1, item2) = if bool then item1 else item2

```
assert ifte(true)("boom")("nailed it") == "boom"
assert ifte(false)("boom")("nailed it") == "nailed it"
```

5. We now move onto using pairs so you have to come up with 3 lambda functions:
   (a) mkPair
   (b) fst
   (c) snd

   Which satisfy:

```
assert fst(mkPair(1)("val")) == 1
assert snd(mkPair("stocks")(100)) == 100
assert fst(snd(mkPair(1)(mkPair(3)(4)))) == 3
```

6. swap a pair

```
assert fst(swap(mkPair("lol")("good stuff"))) == "good stuff"
assert snd(swap(swap(mkPair(1)(2)))) == 2
```

7. We now move onto numbers
   (a) We define $\underline{0} = \lambda f x.x$ $\underline{\mathrm{n}} = \lambda f x.f^n x$ We have two helper functions:

```
def en(num): #en for evalNumber
    return num(lambda x : x + 1)(0)
# Need to define succ for this one.
def toLNum(num):
    return succ(toLNum(num-1)) if num else zero
num1 = toLNum(123)
```

```
num2 = toLNum(245)
two = toLNum(2)
three = toLNum(3)
four = toLNum(4)
```

Define successor (+1)

```
assert en(succ(succ(succ(zero)))) == 3
```

(b) Define add

```
assert en(add(num1)(num2)) == 368
```

(c) Define Multiplication

```
assert en(mult(num1)(num2)) == 30135
```

(d) Define check for Zero

```
assert eb(isZero(zero))
assert  eb(negation(isZero(one)))
```

(e) Define predecessor such that $Pred(x) = max(x-1,0)$

```
assert en(pred(three)) == 2
assert en(pred(zero)) == 0
assert en(pred(num2)) == 244
```

(f) defune sub where sub(y)(x) = x - y

```
assert en(sub(num2)(num1)) == 0
assert en(sub(num1)(num2)) == 122
```

(g) define factorial

```
assert en(fact(four)) == 24
```

3. Recursion

How to do self-referential programs without a special built-in means of accomplishing it. How to establish recursive functions without side effects. Such as giving the function a name in the global namespace Curry's fixed point combinator Y!!!

Look at how to do a factorial function recursively without Y-Combinator. Main trick to feed the function to itself? How would you get a loop?

Problem is we have no way to use the name of the function! Solution pass the function into itself!

```
fact = lambda fact2: lambda x: \
    1 if x == 0 else fact2(fact2)(x - 1) * x
fact(fact)(5)
```

And that's the whole trick basically. The rest is making this syntax easier to use:

Want a nice way to create recursive functions: We want a combinator which takes in these so called "metafunctions" and creates the recursive function:

E.g. for fact above we could have something like

$$mkRec = \lambda rec.rec(rec) \textbf{ (Mockingbird)}$$

Which is a combinator that would make it recursive. Abstract out the part about having to apply it to itself.

We don't want to have to deal with writing rec(rec) in our metafunctions: want to come up with a function to abstract that away so want a function that goes from:

```
fact = lambda rec: lambda x: \
    1 if x == 0 else rec(x - 1) * x
```

TO

```
fact = lambda rec: lambda x: \
    1 if x == 0 else rec(rec)(x - 1) * x
fact(fact)(5)
```

Call this function "doubleTheFuncs"

Now compose these two! i.e.

$$\lambda x.mkRec(doubleTheFuncs(x))$$

We get Y combinator as below!:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Origins: Russel Set :

$$R = \{x | not(x \in x)\}$$

Big contradiction, how would you do this in Lambda calc? Represent a set as its charecteristic function i.e. 1 if in set and 0 if not: so set membership like this would be applying an argument to itself

$$R = \lambda x.negation(xx)$$

Russels Paradox

$$R \in R \iff (R \notin R)$$

In Lambda Calc

$$RR = negation(RR)$$

So! We get a fixed point for the negation operation!! We get RR as being:

$$Y_{not} = RR = (\lambda x.not(xx))(\lambda x.not(xx))$$

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Y assigns fixed points to things - comes from a functional reading of sets.

$$YM = M(YM)$$

Shows lambda calc unsound.

We have a slight issue that Y-Combinator requires python to be lazy, so we have the Z-Combinator to use if you are using python which works around this issue by having the argument passed explicitly

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

$$Z = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$$

Where Z has the property that:

$$Zgv = g(Zg)v$$

https://lptk.github.io/programming/2019/10/15/simple-essence-y-combinator.html - explanation of the Z-combinator

In Haskell:

```
import Data.Function
import Unsafe.Coerce

-- We have a fix function in haskell
-- Can define it like this
fix' :: (a -> a) -> a
fix' f = let x = f x in x

infList = fix' (1:)

metafunc f x = if x == 0 then 1
            else f (x - 1) * x

--Tells haskell to not type check
screwTypes = unsafeCoerce

y :: (a -> a) -> a
y f = (\x -> f (screwTypes x x)) (\x -> f (screwTypes x x))
```

4. Excercise 2

Now we look at lists made from mkPairs: i.e. a list [1,2,3] is represented as mkPair(1,lambda : mkPair(2,lambda : mkPair(3,lambda : NIL))) where NIL = lambda x : true. From here on please use the "*expr1 if bool else expr2*" syntax in python to make sure everything is evaluated lazily. So when checking an if please wrap the middle lambda expression with an eb() to make sure it works.

1.
```
def toNList(l):
    if eb(isEmpty(l)):
        return []
    else:
        return [en(fst(l))] + toNList(snd(l)())
listOne = mkPair(one)(lambda : NIL)
listTwo = mkPair(two)(lambda : listOne)
listThree = mkPair(three)(lambda : listTwo)
```

2. Create an isEmpty operator:

```
assert eb(isEmpty(NIL)) == True
assert eb(isEmpty(listOne)) == False
```

3. Create a getTail operator (remember: you need to eval the tail with - ())

```
assert toNList(getTail(listThree)) == [2,1]
```

4. Create a length function

```
assert en(length(listThree)) == 3
```

5. take function such as take(5)(List) which takes the first 5 things from List, if List shorter than 5 then just returns List

```
assert toNList(take(two)(listThree)) == [3,2]
```

6. Is Divisible function which takes in two args and returns true false depending on if second is divisible by the first

```
assert eb(isDivis(two)(four)) == True
assert eb(isDivis(two)(three)) == False
```

7. Filter function which filters a list based of a predicate:

```
assert toNList(filterOp(isDivis(two))(listThree)) == [2]
```

8. create a function which takes a value and outputs the infinite list of natural numbers starting from that value. i.e. infList(3) = [3,4,5,6...]

```
assert toNList(take(four)(metaInf(one))) == [1,2,3,4]
```

9. Create an infinite list of primes using sieve of Eratosthenes:

```
    ten = toLNum(10)
    assert toNList(takeRec(ten)(primes(metaInf(two)))) == \
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

1. Consider the following two properties of a λ-term M:
   (a) There exist λ-terms A and B with M A = True and M B = False (I)
   (b) For all λ-terms, N, either M N = True or M N = False (II)

   Prove that M cannot have both properties above. Hint: if M has property (I), consider M(Y(λx. ifte (M x) B A))
2. deduce that there is no λ-term E such that for all λ-terms M and N:

$$EMN = \begin{cases} \text{True if M = N} \\ \text{False otherwise} \end{cases}$$

1. Consider encoding a non-empty list of λ-terms $M_1, M_2, ..., M_n$ as the λ-term

$$[M_1, M_2, ..., M_n] = \lambda x f. f M_1 (f M_2 ... (f M_n x) ...)$$

   Where the variables x and f do not occur free in $M_1, M_2, ..., M_n$. Give, with justification, λ-terms Iter, Cons, Append and Nil satisfying
   - Iter M F [M1, M2, ..., Mn] = F M1 (F M2 ... (F Mn M))
   - Cons M [M1, M2, ..., Mn] = [M, M1, M2, ..., Mn]
   - Append [M1, ..., Mm]

PROOF. SUBSTITUTION LEMMA

If $x \neq y$ and $x \notin FV(L)$ then

M[x := N][y := L] = M[y := L][x := N[y := L]]

Proof by structural induction □

5. EXCERCISE 3

Going to derive the S, and K combinators!

A set A equipped with a binary operation $@ : A \times A \to A$ is a *combinatory algebra* if there are elements $K, S \in A$ satisfying for all $a, b, c \in A$

$$@(@(K, a)) = a \tag{5.1}$$

$$@(@(@(S, a), b), c) = @(@(a, c), @(b, c) \tag{5.2}$$

1. Show that there is a binary operation on the set of equivalence classes of closed λ-terms for the equivalence relation of β-conversion that makes it a combinatory algebra

2. Show that every combinatory algebra A contains an element I satisfying

$$@(I,a) = a \tag{5.3}$$

for all $a \in A$. [Hint: what does 5.2 tess us when $a = b = K$]

3. For an arbitrary combinatory algebra A, let A[x] denote the set of expressions given by the grammar

$$e ::= x | \ulcorner a \urcorner | (ee) \tag{5.4}$$

where x is some fixed symbol not in A and a ranges over the elements of A. Given $e \in A[x]$ and $a \in A$, let $e[x := a]$ denote the element of A resulting from interpreting occurrences of x in e by a, interpreting the expressions of the form $\ulcorner a' \urcorner$ by a' and by interpreting expressions of the form ee' using @

   (a) Give the clauses in a definition of $e[x := a]$ by recursion on the structure of e

   (b) For each $e \in A[x]$ show how to define an element $\Lambda_x e \in A$ with the property that

$$@(\Lambda_x e, a) = e[x := a] \tag{5.5}$$

4. Using the usual encoding of Booleans in λ-calculus. Using Part (c)(ii) show that in any combinatory algebra A there are elements *True, False* $\in A$ and a function $If : A \times A \to A$ satisfying

$$@(If(a,b), True) = a \tag{5.6}$$

$$@(If(a,b), False) = b \tag{5.7}$$

for all $a, b \in A$

5. If X has normal form Y: Prove these two corollaries of the Church-Rosser theorem - remember having a normal form can include beta-expansion as well as beta-reductions.

   - X reduces to Y using only alpha and/or beta reductions (no expansions are needed)
   - Y is unique (up to alpha-reduction); i.e., X has no other normal form.