Lambda Calculus with the Birds !

Satyajit Ghana

November 9, 2018

Topic I
Introduction to the Birds

1 Idiot

The Identity

$$\lambda a.a$$

```
In [1]: I = a => a
```

```
Out[1]: [Function: I]
```

2 Mocking Bird

The Self Applicator

$$\lambda f.ff$$

```
In [2]: M = f => f(f)
```

```
Out[2]: [Function: M]
```

3 Kerstrel

The Truth

$$\lambda ab.a$$

```
In [3]: K = a => b => a
```

```
Out[3]: [Function: K]
```

```
In [4]: K.inspect = () => 'T / K'
```

```
Out[4]: [Function]
```

```
In [5]: K
```

```
Out[5]: T / K
```

4 Kite

The False

$$\lambda ab.b = \mathrm{K\,I} = \mathrm{C\,K}$$

```
In [6]: KI = a => b => b
```

```
Out[6]: [Function: KI]
```

```
In [7]: KI.inspect = () => 'F / KI'
```

```
Out[7]: [Function]
```

5 Cardinal

The Reverse

$$\lambda fab.fba$$

```
In [8]: C = f => a => b => f(b)(a)
```

```
Out[8]: [Function: C]
```

6 Blue Bird

The Composition

$$\lambda fga.f(ga)$$

```
In [9]: B = f => g => a => f(g(a))
```

```
Out[9]: [Function: B]
```

7 Thrush

The Flipper

$$\lambda af.fa = \mathrm{C\,I}$$

```
In [10]: T = a => f => f(a)
```

```
Out[10]: [Function: T]
```

8 Vireo

The Smallest Data Structure

$$\lambda abf.fab = \mathrm{B\,C\,T}$$

```
In [11]: V = a => b => f => f(a)(b)
```

```
Out[11]: [Function: V]
```

9 Black Bird

Blue Bird for a function that takes two arguments
It's the composition of composition of composition

$$\lambda fgab.f(gab) = \text{B B B}$$

```
In [12]: B1 = f => g => a => b => f(g(a)(b))
```

```
Out[12]: [Function: B1]
```

Topic II
Birds ! Birds ! Birds !

1 Fun with these Birds !

```
In [13]: C(K)('T')('F')
```

```
Out[13]: 'F'
```

```
In [14]: C(KI)('T')('F')
```

```
Out[14]: 'T'
```

Cardinal of Kerstrel is Kite
Cardinal of Kite is Kerstrel

2 Church Encodings

2.1 Booleans

TRUE $= \lambda ab.a =$ K

```
In [15]: TRUE = K
```

```
Out[15]: T / K
```

FALSE $= \lambda ab.b =$ KI $=$ CK

```
In [16]: FALSE = KI
```

```
Out[16]: F / KI
```

NOT $= \lambda p.pFT =$ C

```
In [17]: NOT = C
```

```
Out[17]: [Function: C]
```

```
In [18]: NOT(TRUE)('T')('F')
```

```
Out[18]: 'F'
```

AND = $\lambda pq.pqF = \lambda pq.pqp$
if p is false and it selects false, then p can select itself

```
In [19]: AND = p => q => p(q)(p)
```

```
Out[19]: [Function: AND]
```

```
In [20]: AND(TRUE)(TRUE)
```

```
Out[20]: T / K
```

```
In [21]: AND(TRUE)(FALSE)
```

```
Out[21]: F / KI
```

```
In [22]: AND(FALSE)(FALSE)
```

```
Out[22]: F / KI
```

OR = $\lambda pq.pTq = \lambda pq.ppq = \lambda pq.Mq =$ M

```
In [23]: OR = p => q => M(p)(q)
```

```
Out[23]: [Function: OR]
```

```
In [24]: OR(TRUE)(TRUE)
```

```
Out[24]: T / K
```

```
In [25]: OR(TRUE)(FALSE)
```

```
Out[25]: T / K
```

```
In [26]: OR(FALSE)(FALSE)
```

```
Out[26]: F / KI
```

$(\lambda pq.ppq)xy = xxy$
$Mxy = xxy$
OR = M

```
In [27]: M(TRUE)(TRUE)
```

```
Out[27]: T / K
```

```
In [28]: M(TRUE)(FALSE)
```

```
Out[28]: T / K
```

```
In [29]: M(FALSE)(FALSE)
```

```
Out[29]: F / KI
```

BEQ $= \lambda pq.pq(\text{NOT}q)$
This is also the `XNOR` or the Equality

```
In [30]: BEQ = p => q => p(q)(NOT(q))
```

```
Out[30]: [Function: BEQ]
```

```
In [31]: BEQ(TRUE)(TRUE)('T')('F')
```

```
Out[31]: 'T'
```

```
In [32]: BEQ(FALSE)(FALSE)('T')('F')
```

```
Out[32]: 'T'
```

```
In [33]: BEQ(FALSE)(TRUE)('T')('F')
```

```
Out[33]: 'F'
```

2.2 Numerals

ZERO $= \lambda fa.a$

```
In [34]: ZERO = f => a  => a
```

```
Out[34]: [Function: ZERO]
```

ONCE $= \lambda fa.fa$

```
In [35]: ONCE = f => a => f(a)
```

```
Out[35]: [Function: ONCE]
```

TWICE $= \lambda fa.f(fa)$

```
In [36]: TWICE = f => a => f(f(a))
```

```
Out[36]: [Function: TWICE]
```

THRICE $= \lambda fa.f(f(fa)$

```
In [37]: THRICE = f => a => f(f(f(a)))
```

```
Out[37]: [Function: THRICE]
```

FOURFOLD $= \lambda fa.f(f(f(fa)))$

```
In [38]: FOURFOLD = f => a => f(f(f(f(a))))
```

```
Out[38]: [Function: FOURFOLD]
```

```
In [39]: N0 = ZERO
         N1 = ONCE
         N2 = TWICE
         N3 = THRICE
         N4 = FOURFOLD
```

```
Out[39]: [Function: FOURFOLD]
```

$\text{SUCC} = \lambda nfa.f(nfa) = \lambda nf.\text{B}f(nf)$

```
In [40]: SUCC = n => f => x => f(n(f)(x))
```

```
Out[40]: [Function: SUCC]
```

This is the same as function composition, why not use the Blue Bird ?

```
In [41]: SUCC = n => f => B(f)(n(f))
```

```
Out[41]: [Function: SUCC]
```

```
In [42]: SUCC(THRICE)(x => x + 1)(0)
```

```
Out[42]: 4
```

$\text{ADD} = \lambda nk.n \text{ SUCC } k$

```
In [43]: ADD = n => k => n(SUCC)(k)
```

```
Out[43]: [Function: ADD]
```

Read it as n times SUCC, applied to k, same as `SUCC(SUCC(SUCC(k)))` where SUCC is n times this example was 3 added to k

```
In [44]: ADD(N3)(N2)(x => x + 1)(0)
```

```
Out[44]: 5
```

$\text{MULT} = \lambda nkf.n(kf) = \lambda nk.\text{B}nk = \text{B}$

```
In [45]: MULT = n => k => B(n)(k)
         MULT = B
```

```
Out[45]: [Function: B]
```

```
In [46]: MULT(N3)(N2)(x => x + 1)(0)
```

```
Out[46]: 6
```

$\text{POW} = \lambda nk.k \text{ MULT } n = \text{C I} = \text{T}$

```
In [47]: POW = T
```

```
Out[47]: [Function: T]
```

```
In [48]: POW(N3)(N2)(x => x + 1)(0)
```

```
Out[48]: 9
```

IS0 $= \lambda n.n(KF)T$
KF always gives a False, Kerstrel of False

```
In [49]: IS0 = n => n(K(FALSE))(TRUE)
```

```
Out[49]: [Function: IS0]
```

```
In [50]: IS0(N0)
```

```
Out[50]: T / K
```

```
In [51]: IS0(N1)
```

```
Out[51]: F / KI
```

PRED $= \lambda n.n(\lambda g.\text{IS0}(g\text{N1})\text{I}(\text{B SUCC } g))(\text{K N0})\text{N0}$
V I M
Vireo is the smallest data structure, you box in the two arguments like f(a)(b) and then whenever you want a value back you send in a function to recieve eithe a or b

```
In [52]: vim = V(I)(M)
```

```
Out[52]: [Function]
```

```
In [53]: vim(K)
```

```
Out[53]: [Function: I]
```

```
In [54]: vim(C(K)) // C(K) = KI
```

```
Out[54]: [Function: M]
```

PAIR = V

```
In [55]: PAIR = V
```

```
Out[55]: [Function: V]
```

FST $= \lambda p.pK$

```
In [56]: FST = p => p(K)
```

```
Out[56]: [Function: FST]
```

In [57]: FST(vim)

Out[57]: [Function: I]

SND $= \lambda p.p(KI)$

In [58]: SND = p => p(KI)

Out[58]: [Function: SND]

In [59]: SND(vim)

Out[59]: [Function: M]

PHI $= \lambda p.$V(SND p)(SUCC (SND)p)
copy 2nd to 1st, and increment 2nd

In [60]: PHI = p => V(SND(p))(SUCC(SND(p)))

Out[60]: [Function: PHI]

In [61]: SND(PHI(V(M)(N3)))(x => x + 1)(0)

Out[61]: 4

In [62]: FST(PHI(V(M)(N3)))(x => x + 1)(0)

Out[62]: 3

```
   N0 PHI(N0, N0) = (N0, N0)
N1 PHI(N0, N0) = (N0, N1)
N2 PHI(N0, N0) = (N1, N2)
...
N8 PHI(N0, N0) = (N7, N8)
```

Holy Cow !, you got the predecessor working !
PRED $= \lambda n =$ FST $(n\Phi($PAIR ZERO ZERO$))$

In [63]: PRED = n => FST(n(PHI)(V(N0)(N0)))

Out[63]: [Function: PRED]

FIRST of "n" application of PHI to PAIR of ZERO, ZERO

In [64]: PRED(N3)(x => x + 1)(0)

Out[64]: 2

SUB $= \lambda nk.k$ PRED n

In [65]: SUB = n => k => k(PRED)(n)

Out[65]: [Function: SUB]

```
In [66]: SUB(N4)(N3)(x => x + 1)(0)

Out[66]: 1
```

$$\text{LEQ} = \lambda nk.\text{IS0}(\text{SUB}\ nk)$$

```
In [67]: LEQ = n => k => IS0(SUB(n)(k))

Out[67]: [Function: LEQ]

In [68]: LEQ(N3)(N4)

Out[68]: T / K

In [69]: LEQ(N4)(N3)

Out[69]: F / KI
```

$$\text{EQ} = \lambda nk.\text{AND}(\text{LEQ}nk)(\text{LEQ}kn)$$

```
In [70]: EQ = n => k => AND(LEQ(n)(k))(LEQ(k)(n))

Out[70]: [Function: EQ]

In [71]: EQ(N0)(N0)

Out[71]: T / K

In [72]: EQ(N1)(N0)

Out[72]: F / KI
```

$$\text{GT} = \lambda nk.\text{NOT}(\text{LEQ}\ nk) = \text{B1 NOT LEQ}$$

```
In [73]: GT = n => k => NOT(LEQ(n)(k))

Out[73]: [Function: GT]

In [74]: GT = B1(NOT)(LEQ)

Out[74]: [Function]

In [75]: GT(N1)(N0)('T')('F')

Out[75]: 'T'

In [76]: GT(N0)(N0)('T')('F')

Out[76]: 'F'
```