程式人

用十分鐘搞懂 λ-Calculus

從 Functional Programming 到 Alonzo Church 的神奇程式

陳鍾誠

2022 年 10 月 12 日

對於程式人而言

- 1936 年是個神奇的年份，因為在這一年
  - 4 月：Church 發表了論文說兩函數正規式是否相等也是不可計算的
  - 11 月： 圖靈發表論文說停止問題不可能正確計算

圖靈的證明

- 是建構在圖靈機的模型之上

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

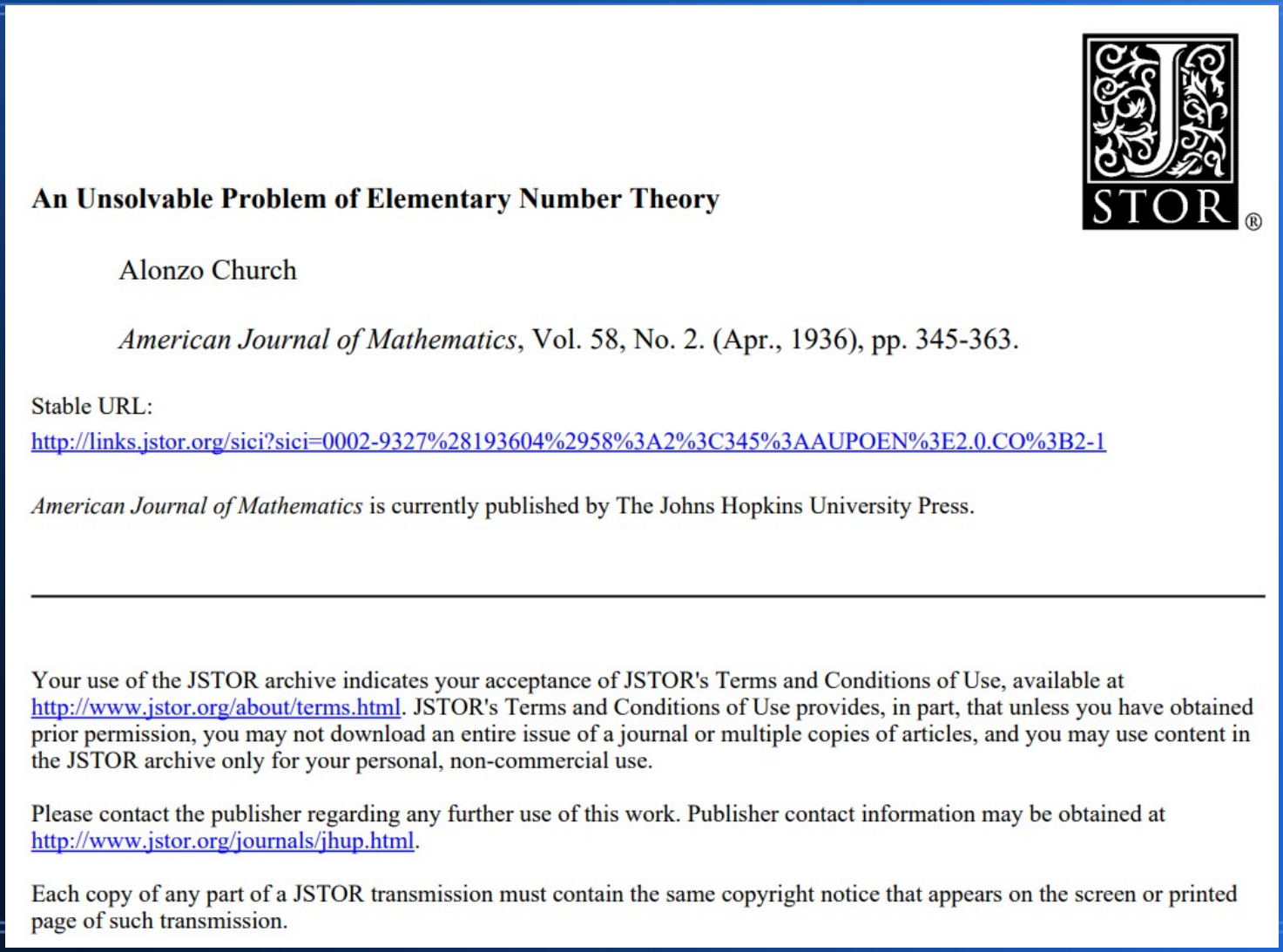
[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

而 church 的證明

- 則是建構在 λ-Calculus 這個函數系統之上



An Unsolvable Problem of Elementary Number Theory

Alonzo Church

American Journal of Mathematics, Vol. 58, No. 2. (Apr., 1936), pp. 345-363.

在那個還沒有電腦也沒有程式的年代

- 我們很難想像他們是如何創造出這兩組抽象的機器與抽象的程式

即使到了今天

- 我都很難寫出像 λ-Calculus 這樣優美的純粹的函數式程式

不過

- 我雖然沒有很懂
  還是想為各位介紹
  λ-Calculus

但是要學 Lambda Calculus

- 我們要先學會程式世界的
  - 函數 function
  - 遞迴 recursive
  - 函數式編程 function programming

如果你用 python 寫下列函數

- 那麼應該很容易懂

```
1    def square(x):
2        return x*x
3
4    print('square(3)=', square(3))
5
```

問題　輸出　終端機　偵錯主控台

```
$ python function.py
square(3)= 9
```

但是如果你寫成這樣

```
1   square = lambda x: x*x
2
3   print('square(3)=', square(3))
```

問題　輸出　終端機　偵錯主控台

```
$ python lambda.py
square(3)= 9
```

- 可能就有些人會看不懂了！

其實 lambda 就是沒有名字的函數

- 所以下列兩者是等價的

```
lambda x: x*x
```

=

```
def square(x):
    return x*x
```

所以下列兩者都是費氏數列

- 只是普通函數和 lambda 的差別而已

```
def fibonacci (n):
    if n < 0: raise
    if n == 0: return 0
    if n == 1: return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```


```
fibonacci = lambda n: \
    0 if n == 0 else  \
    1 if n == 1 else  \
    fibonacci(n-1) + fibonacci(n-2)
```

利用 lambda

- 我們甚至可以定義出基本的 If 函數

```
1   If = lambda cond, a, b: a if cond else b
2   Max = lambda a,b: If(a>b, a, b)
3   Min = lambda a,b: If(a<b, a, b)
4
5   print('Max(3,5)=', Max(3,5))
6   print('Min(3,5)=', Min(3,5))
7
```

問題　輸出　終端機　偵錯主控台

```
$ python if.py
Max(3,5)= 5
Min(3,5)= 3
```

然後利用這個 If 設計出任何程式

- 完全不需要使用 for, while 這些迴圈語法

但是

- 當我們用上面的 If 函數設計費氏數列 Fibonacci 函數時，遇到了一個問題！

Fibonacci 在遞迴後當掉了

```python
If = lambda cond, a, b: a if cond else b

Fibonacci = lambda n: \
  If(n<2, n, Fibonacci(n-1)+Fibonacci(n-2))

print('Fibonacci(8)=', Fibonacci(8))
```

問題　輸出　終端機　偵錯主控台　bash

```
    If(n<2, n, Fibonacci(n-1)+Fibonacci(n-2))
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded in comparison
```

當掉的原因是

```
1  If = lambda cond, a, b: a if cond else b
2
3  Fibonacci = lambda n: \
4      If(n<2, n, Fibonacci(n-1)+Fibonacci(n-2))
5
6  pri  ('Fibonacci(8)=', Fibonacci(8))
```

Fibnacci(n-1) 必須在函數呼叫前就計算出來，結果就 8=>7=>6=>5=>4=>3=>2=>1=>0=>-1=>-2...

n<2 的檢查並沒有阻止 Fibonacii 遞迴的無窮運行

端機　偵錯主控台　bash

```
    If(n<2, n, Fibonacci(n-1)+Fibonacci(n-2))
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded in comparison
```

解決辦法是

- 我們多加一個懶惰版的 If_lazy

```python
# 如果 Fibonacci 用 If 而非 If_lazy,
# 那麼 Fibonacci(n-1) 會馬上運算
# 結果就是 8=>7=>....0=>-1=>-2=>.... 當掉

If_lazy = lambda cond, a, b: a() if cond else b()
# 採用 If_lazy, 並要求 a, b 必須是函數，這樣就可以阻止 a, b 立即算出

Fibonacci = lambda n: \
    If_lazy(n<2, lambda:n, lambda:Fibonacci(n-1)+Fibonacci(n-2))

print('Fibonacci(8)=', Fibonacci(8))
```

問題 輸出 終端機 偵錯主控台 bash

```
Fibonacci(8)= 21
```

於是 Fibonacci 就不會當掉了

If(cond, a, b) 有三個參數

程式如下

```
If = lambda cond, a, b: a() if cond else b()
```

如果我們把它改寫成

```
If = lambda cond: lambda a: lambda b: a() if cond else b()
```

那麼呼叫方法就會變成

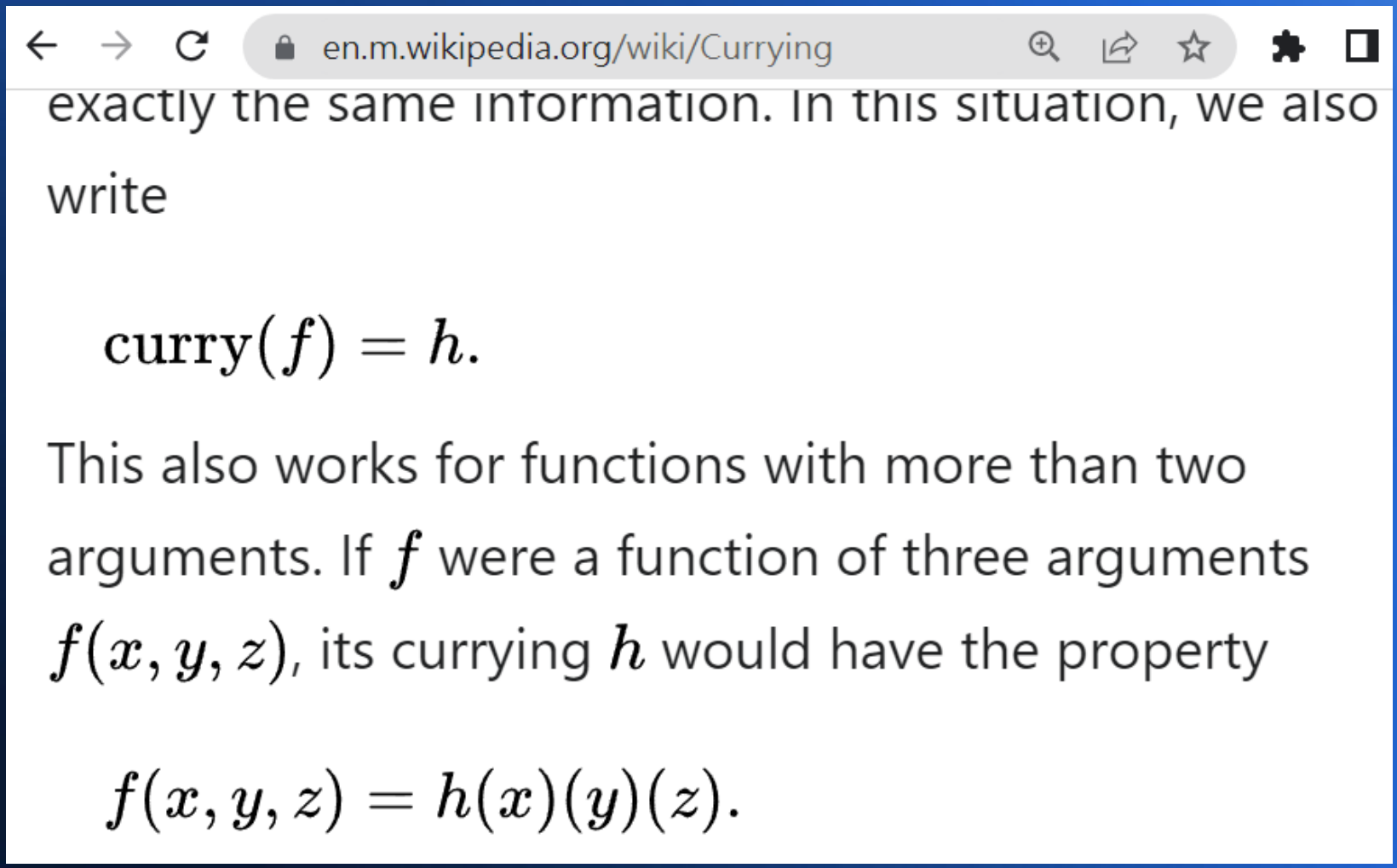
```
Fibonacci = lambda n: \
  If(n<2)(lambda:n)(lambda:Fibonacci(n-1)+Fibonacci(n-2))
```

而不是原來的那樣

```
Fibonacci = lambda n: \
  If(n<2, lambda:n, lambda:Fibonacci(n-1)+Fibonacci(n-2))
```

這種讓參數一個一個傳進來的改寫方法

- 稱為柯里化 Currying



Lambda Calculus 裏

- 所有的函數都只有一個參數
- 對兩個以上參數的函數
  都必須用 Currying 改寫

而且

- 在 Lambda Calculus 中
  連 true/false/0/1/2…
  通通都被定義成函數
  而非二進位資料

於是我們看到

- 邏輯的世界變成這樣

```
# Church Booleans : Logic
IF    = lambda c:lambda x:lambda y:c(x)(y) #  if: λ c x y. c x y
TRUE  = lambda x:lambda y:x # if true then x # 兩個參數執行第一個
FALSE = lambda x:lambda y:y # if false then y # 兩個參數執行第二個
AND   = lambda p:lambda q:p(q)(p) # if p then q else p
OR    = lambda p:lambda q:p(p)(q) # if p then p else q
XOR   = lambda p:lambda q:p(NOT(q))(q) #  if p then not q else q
NOT   = lambda c:c(FALSE)(TRUE) # if c then false else true
```

https://gitlab.com/cccnqu111/alg/-/blob/master/18c-lambdaCalculus/lambdaCalculus.py

整數世界則變成這樣

```
# Church Numerals
_zero  = lambda f:IDENTITY # 0      : 用 λf. λx. x 當 0
_one   = SUCCESSOR(_zero)  # 1=S(0) : λf. λf. λx. x 當 1
_two   = SUCCESSOR(_one)   # 2=S(1) : λf. λf. λf. λx. x 當 2
_three = SUCCESSOR(_two)   # 3=S(2)
_four  = MULTIPLICATION(_two)(_two)  # 4 = 2*2
_five  = SUCCESSOR(_four)            # 5 = S(4)
_eight = MULTIPLICATION(_two)(_four) # 8 = 2*4
_nine  = SUCCESSOR(_eight)           # 9 = S(8)
_ten   = MULTIPLICATION(_two)(_five) # 10 = 2*5
```

然後你可以比大小

```
# Comparison
IS_ZERO              = lambda n:n(lambda _:FALSE)(TRUE)
IS_LESS_THAN         = lambda m:lambda n:NOT(IS_LESS_THAN_EQUAL(n)(m))
IS_LESS_THAN_EQUAL   = lambda m:lambda n:IS_ZERO(SUBTRACTION(m)(n))
IS_EQUAL             = lambda m:lambda n:AND(IS_LESS_THAN_EQUAL(m)(n))(I
IS_NOT_EQUAL         = lambda m:lambda n:OR(NOT(IS_LESS_THAN_EQUAL(m)(n)
IS_GREATER_THAN_EQUAL = lambda m:lambda n:IS_LESS_THAN_EQUAL(n)(m)
IS_GREATER_THAN      = lambda m:lambda n:NOT(IS_LESS_THAN_EQUAL(m)(n))
IS_NULL              = lambda p:p(lambda x:lambda y:FALSE)
NIL                  = lambda x:TRUE
```

接著將資料結構
隱藏在函數的 Closure 中

```
CONS = lambda x:lambda y:lambda f:f(x)(y)
CAR  = lambda p:p(TRUE)
CDR  = lambda p:p(FALSE)
```

加上一種稱為 Y-Combinator 的奇妙設計

```
Y = lambda f:\
  (lambda x:f(lambda y:x(x)(y)))\
  (lambda x:f(lambda y:x(x)(y)))
```

就可以讓你定義出

- Functional Programmig的基本函數，包含
  - range(from, to)
  - map(list, f)

Range(from, to)

```
RANGE = lambda m:lambda n:Y(lambda f:lambda m:IF(IS_EQUAL(m)(n))\
  (lambda _: CONS(m)(NIL))\
  (lambda _: CONS(m)(f(SUCCESSOR(m))))\
(NIL))(m)
```

Map(array, f)

```
MAP = lambda x:lambda g:Y(lambda f:lambda x:IF(IS_NULL(x))\
  (lambda _: x)\
  (lambda _: CONS(g(CAR(x)))(f(CDR(x))))\
(NIL))(x)
```

然後你就可以寫出階層函數 Factorial

```
FACTORIAL = Y(lambda f:lambda n:IF(IS_ZERO(n))\
  (lambda _:SUCCESSOR(n))\
  (lambda _:MULTIPLICATION(n)(f(PREDECESSOR(n))))\
(NIL))
```

也可以寫出費式數列
Fibonacci

```
FIBONACCI = Y(lambda f:lambda n:\
  IF(IS_LESS_THAN_EQUAL(n)(SUCCESSOR(lambda f:IDENTITY)))\
  (lambda _:n)\
  (lambda _:ADDITION\
    (f(PREDECESSOR(n)))\
    (f(PREDECESSOR(PREDECESSOR(n)))))\
(NIL))
```

雖然這些程式看來很可怕

```
TEST('RANGE')(ASSERT(AND(\
  AND\
    (IS_EQUAL(CAR(RANGE(_three)(_five)))(_three))\
    (IS_EQUAL(CAR(CDR(RANGE(_three)(_five))))(_four)))(\
  AND\
    (IS_EQUAL(CAR(CDR(CDR(RANGE(_three)(_five)))))(_five))\
    (IS_NULL(CDR(CDR(CDR(RANGE(_three)(_five)))))))))
```

但驗正後發現都是對的

```
$ python lambdaCalculus.py
[√] _TRUE
[√] _FALSE
[√] _AND
[√] _OR
[√] _XOR
[√] _NOT
[√] _IDENTITY
[√] _SUCCESSOR
[√] _PREDECESSOR
[√] _ADDITION
[√] _SUBTRACTION
[√] _MULTIPLICATION
[√] _POWER
[√] _ABS_DIFFERENCE
[√] _IS_ZERO
[√] _IS_LESS_THAN
[√] _IS_LESS_THAN_EQUAL
[√] _IS_EQUAL
[√] _IS_NOT_EQUAL
[√] _IS_GREATER_THAN_EQUAL
[√] _IS_GREATER_THAN
[√] _IS_NULL
[√] _CAR
[√] _CDR
[√] _CONS
[√] _RANGE
[√] _MAP

--- Examples ---

[√] _FACTORIAL: 5! = 120
[√] _FIBONACCI: 10 = 55
```

如果你不信

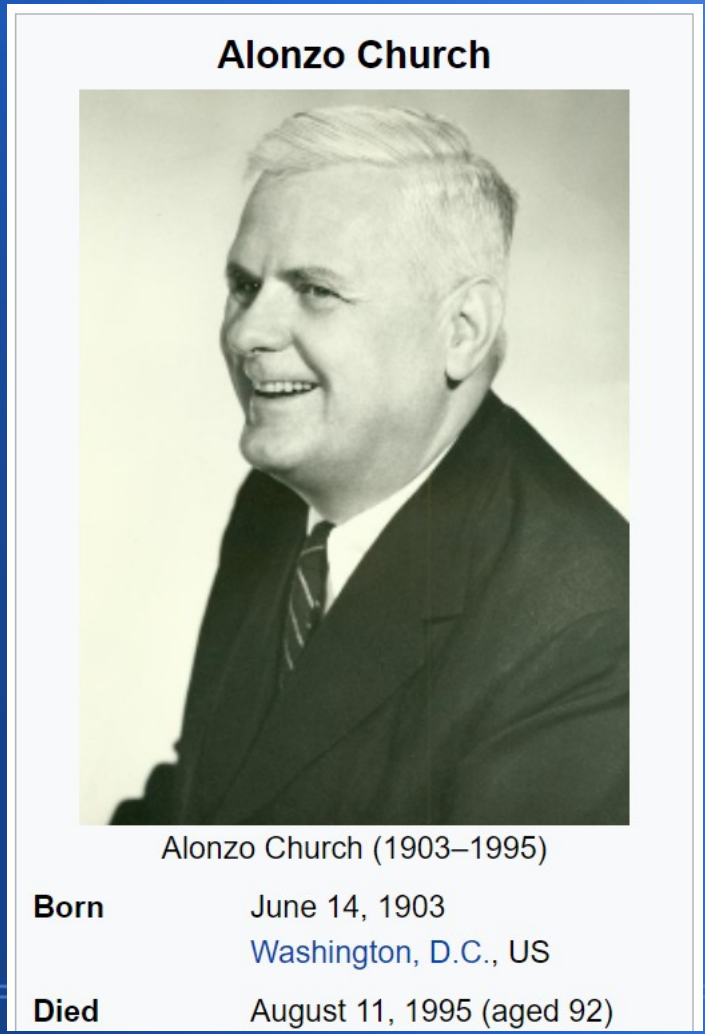
- 可以自己跑跑看！

這個 Lambda Calculus 程式

- 當然不是我寫的 ...

原創者是 Alonzo Church

- 他在還沒有電腦的年代
  就透過他的大腦和筆
  寫下了這樣的《程式》

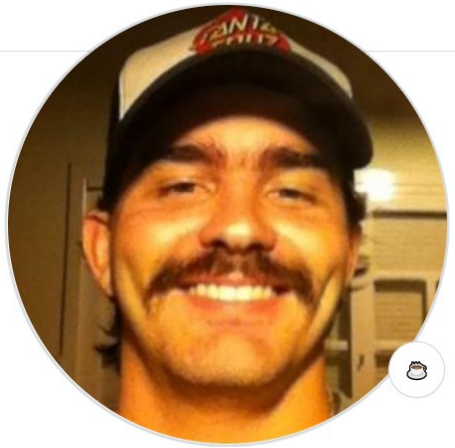


而我給各位看的程式

- 則是github上一位名叫gtramontina的老兄所寫的

github.com/gtramontina/lambda/blob/master/lambda.js

```
1   // Arithmetics ---------------------------------------------------------------
2
3   IDENTITY       = x => x
4   SUCCESSOR      = n => f => x => f(n(f)(x))
5   PREDECESSOR    = n => f => x => n(g => h => h(g(f)))(_ => x)(u => u)
6   ADDITION       = m => n => n(SUCCESSOR)(m)
7   SUBTRACTION    = m => n => n(PREDECESSOR)(m)
8   MULTIPLICATION = m => n => f => m(n(f))
9   POWER          = x => y => y(x)
10  ABS_DIFFERENCE = x => y => ADDITION(SUBTRACTION(x)(y))(SUBTRACTION(y)(x))
11
12  // Logic ---------------------------------------------------------------------
13
14  TRUE  = t => f => t
15  FALSE = t => f => f
16  AND   = p => q => p(q)(p)
17  OR    = p => q => p(p)(q)
18  XOR   = p => q => p(NOT(q))(q)
19  NOT   = c => c(FALSE)(TRUE)
20  IF    = c => t => f => c(t)(f)
```



Guilherme J. Tramontina
gtramontina

而我

- 則只不過是把 JavaScript 改成 Python 而已

gitlab.com/cccnqu111/alg/-/blob/master/18c-lambdaCalculus/lambdaCalculus.py

Search GitLab /

mbdaCalculus.py 6.68 KiB

```python
# Lambda Calculus 當中的所有資料結構都是用 closure 達成的
# 換言之，所有的資料結構都是函數閉包。

# Church Booleans : Logic
IF    = lambda c:lambda x:lambda y:c(x)(y) #  if: λ c x y. c x y # if c then x else y.
TRUE  = lambda x:lambda y:x # if true then x # 兩個參數執行第一個
FALSE = lambda x:lambda y:y # if false then y # 兩個參數執行第二個
AND   = lambda p:lambda q:p(q)(p) # if p then q else p
OR    = lambda p:lambda q:p(p)(q) # if p then p else q
XOR   = lambda p:lambda q:p(NOT(q))(q) #  if p then not q else q
NOT   = lambda c:c(FALSE)(TRUE) # if c then false else true

ASSERT = lambda truth: (IF(truth)\
    (lambda description:f'[\x1b[32m✓\x1b[0m] _{description}')\
    (lambda description:f'[\x1b[31m✗\x1b[0m] _{description}')
)
```

這樣的寫程式方法

- 完全顛覆了我原本的程式觀念

看完之後只覺得

- 這種程式真的不像是人寫的

```
TEST   = lambda description:lambda assertion:\
    print(assertion(description))

TEST('TRUE')\
    (ASSERT(TRUE))

TEST('FALSE')\
    (REFUTE(FALSE))

TEST('AND')\
  (ASSERT(AND(TRUE)(TRUE)))

TEST('OR')(ASSERT(AND\
  (AND(OR(TRUE)(FALSE))(OR(FALSE)(TRUE)))\
  (NOT(OR(FALSE)(FALSE)))))
```

應該是

- 上帝所寫下來的吧！

更厲害的是

- Alonzo Church 竟然還發現了一些定理

ics.uci.edu/~lopes/teaching/inf212W12/readings/church.pdf

.pdf 1 / 20 125%

An Unsolvable Problem of Elementary Number Theory

Alonzo Church

American Journal of Mathematics, Vol. 58, No. 2. (Apr., 1936), pp. 345-363.

Stable URL:
http://links.jstor.org/sici?sici=0002-9327%28193604%2958%3A2%3C345%3AAUPOEN%3E2.0.CO%3B2-1

American Journal of Mathematics is currently published by The Johns Hopkins University Press.

定理 1, 2, 3, 4, 5

THEOREM I. *If a formula is in normal form, no reduction of it is possible.*

THEOREM II. *If a formula has a normal form, this normal form is unique to within applications of Operation* I, *and any sequence of reductions of the formula must (if continued) terminate in the normal form.*

THEOREM III. *If a formula has a normal form, every well-formed part of it has a normal form.*

THEOREM IV. *If F is a recursive function of two positive integers, and if for every positive integer x there exists a positive integer y such that $F(x, y) > 1$, then the function F^*, such that, for every positive integer x, $F^*(x)$ is equal to the least positive integer y for which $F(x, y) > 1$, is*

THEOREM V. *If F is a recursive function of one positive integer, and if there exist an infinite number of positive integers x for which $F(x) > 1$, then the function F^0, such that, for every positive integer n, $F^0(n)$ is equal to the n-th positive integer x (in order of increasing magnitude) for which $F(x) > 1$, is recursive.*

定理 6, 7, 8, 9, 10

THEOREM VI. *The property of a positive integer, that there exists a well-formed formula of which it is the Gödel representation is recursive.*

THEOREM VII. *The set of well-formed formulas is recursively enumerable.*

This follows from Theorems V and VI.

THEOREM VIII. *The function of two variables, whose value, when taken of the well-formed formulas* $\mathbf{F}$ *and* $\mathbf{X}$*, is the formula* $\{\mathbf{F}\}(\mathbf{X})$*, is recursive.*

THEOREM IX. *The function, whose value for each of the positive integers* $1, 2, 3, \cdots$ *is the corresponding formula* $1, 2, 3, \cdots$*, is recursive.*

THEOREM X. *A function, whose value for each of the formulas* $1, 2, 3, \cdots$ *is the corresponding positive integer, and whose value for other well-formed formulas is a fixed positive integer, is recursive. Likewise the function, whose value for each of the formulas* $1, 2, 3, \cdots$ *is the corresponding positive integer*

定理 11, 12, 13, 14, 15

THEOREM XI. *The relation of immediate convertibility, between well-formed formulas, is recursive.*

THEOREM XII. *It is possible to associate simultaneously with every well-formed formula an enumeration of the formulas obtainable from it by conversion, in such a way that the function of two variables, whose value, when taken of a well-formed formula* $\mathfrak{A}$ *and a positive integer n, is the n-th formula in the enumeration of the formulas obtainable from* $\mathfrak{A}$ *by conversion, is recursive.*

THEOREM XIII. *The property of a well-formed formula, that it is in principal normal form, is recursive.*

THEOREM XIV. *The set of well-formed formulas which are in principal normal form is recursively enumerable.*

This follows from Theorems V, VII, XIII.

THEOREM XV. *The set of well-formed formulas which have a normal form is recursively enumerable.*[15]

定理 16, 17, 18, 19

THEOREM XVI. *Every recursive function of positive integers is λ-definable.*[16]

THEOREM XVII. *Every λ-definable function of positive integers is recursive.*[17]

THEOREM XVIII. *There is no recursive function of a formula* **C**, *whose value is* 2 *or* 1 *according as* **C** *has a normal form or not.*

THEOREM XIX. *There is no recursive function of two formulas* **A** *and* **B**, *whose value is* 2 *or* 1 *according as* **A** *conv* **B** *or not.*

最後這條定理

- 基本上是說
  判斷兩個程式是否相同
  是不可計算的問題！

THEOREM XIX. *There is no recursive function of two formulas* ***A*** *and* ***B****, whose value is* 2 *or* 1 *according as* ***A*** *conv* ***B*** *or not.*

雖然博士班的時候

- 我唸過計算理論
  也考過資格考...

但是關於那些數學

- 我已經全部都還給老師了

所以

- 不要問我為甚麼？

想了解Lambda Calculus

- 還有 Church 的定理

就去看下面的那些程式

https://github.com/gtramontina/lambda/blob/master/lambda.js

https://gitlab.com/cccnqu111/alg/-/blob/master/18c-lambdaCalculus/lambdaCalculus.py

還有 Church 的論文吧！

https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/church.pdf

這就是我們今天的

- 十分鐘系列

希望你會喜歡

我們下次見！

Good Bye !