Learn You Some Lambda Calculus

Bernardo Amorim

Similarities?

- λ - Lambda
- Functional Programming Languages

Anonymous Functions

Python	Ruby
`lambda x: x`	`lambda { |x| x }`

λ ≈ Functional Programming?

λ ≈ Anonymous Functions?

Alonzo Church

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By Alonzo Church.

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly the condition that the function f be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

Another example of a problem of this class is, for instance, the problem of topology, to find a complete set of effectively calculable invariants of closed three-dimensional simplicial manifolds under homeomorphisms. This problem can be interpreted as a problem of elementary number theory in view of the fact that topological complexes are representable by matrices of incidence. In fact, as is well known, the property of a set of incidence matrices that it represent a closed three-dimensional manifold, and the property of two sets of incidence matrices that they represent homeomorphic complexes, can both be described in purely number-theoretic terms. If we enumerate, in a straightforward way, the sets of incidence matrices which represent closed three-dimensional manifolds, it will then be immediately provable that the problem under consideration (to find a complete set of effectively calculable invariants of closed three-dimensional manifolds) is equivalent to the problem, to find an effectively calculable function f of positive integers, such that $f(m, n)$ is equal to 2 if and only if the m-th set of incidence matrices and the n-th set of incidence matrices in the enumeration represent homeomorphic complexes.

Other examples will readily occur to the reader.

[1] Presented to the American Mathematical Society, April 19, 1935.

[2] The selection of the particular positive integer 2 instead of some other is, of course, accidental and non-essential.

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.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.[1]

By Alonzo Church.

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, x_2, \cdots, x_n) = 2$ [2] is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, x_2, \cdots, x_n$ as free variables.

An example of such a problem is the problem to find a means of determining of any given positive integer n whether or not there exist positive integers x, y, z, such that $x^n + y^n = z^n$. For this may be interpreted, required to find an effectively calculable function f, such that $f(n)$ is equal to 2 if and only if there exist positive integers x, y, z, such that $x^n + y^n = z^n$. Clearly the condition that the function f be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

Another example of a problem of this class is, for instance, the problem of topology, to find a complete set of effectively calculable invariants of closed three-dimensional simplicial manifolds under homeomorphisms. This problem can be interpreted as a problem of elementary number theory in view of the fact that topological complexes are representable by matrices of incidence. In fact, as is well known, the property of a set of incidence matrices that it represent a closed three-dimensional manifold, and the property of two sets of incidence matrices that they represent homeomorphic complexes, can both be described in purely number-theoretic terms. If we enumerate, in a straightforward way, the sets of incidence matrices which represent closed three-dimensional manifolds, it will then be immediately provable that the problem under consideration (to find a complete set of effectively calculable invariants of closed three-dimensional manifolds) is equivalent to the problem, to find an effectively calculable function f of positive integers, such that $f(m, n)$ is equal to 2 if and only if the m-th set of incidence matrices and the n-th set of incidence matrices in the enumeration represent homeomorphic complexes.

Other examples will readily occur to the reader.

[1] Presented to the American Mathematical Society, April 19, 1935.

[2] The selection of the particular positive integer 2 instead of some other is, of course, accidental and non-essential.

Turing Completeness

and the Church-Turing thesis

Programming Challenge

Weird sub-set of JS

Or python, ruby, etc..

Weird sub-set of JS

Valid terms can be:

- Variable names such as `x`, `y`, or `my_variable`
- Anonymous functions definitions with exactly one argument like `x => BODY` where `BODY` is also a valid **term**.
- Application of functions, like `A(B)` where both `A` and `B` are valid **terms**.
- No use of externally defined variables or functions

Weird sub-set of JS

```
x => x
x => x(x)
(x => x)(x => x)
f => x => x
f => x => f(x)
f => x => f(f(f(f(x))))
```

Weird sub-set of Python

```
lambda x: x
lambda x: x(x)
(lambda x: x)(lambda x: x)
lambda f: lambda x: x
lambda f: lambda x: f(x)
lambda f: lambda x: f(f(f(f(x))))
```

Weird sub-set of Ruby

```
lambda { |x| x }
lambda { |x| x.(x) }
lambda { |x| x }.(lambda { |x| x })
lambda { |f| lambda { |x| x } }
lambda { |f| lambda { |x| f.(x) } }
lambda { |f| lambda { |x| f.(f.(f.(f.(x)))) } }
```

Weird sub-set of JS

This is **Turing-Complete**

Here is a factorial function.

```
(
  f => (x => x(x))(x => f(y => x(x)(y)))
)(
  fact => n => (
    b => tf => ff => b(tf)(ff)(b)
  )(
    (n => n(a => b => f => f)(t => f => t))(n)
  )(
    a => f => x => f(x)
  )(
    a => (
      n => m => f => x => n(m(f))(x)
    )(
      n
    )(
      fact((n => f => x => n(g => h => h(g(f)))(a => x)(u => u))(n))
    )
  )
)
```

Encoding and Decoding

```
toNumber(
  factorial(
    fromNumber(5)
  )
)
```

OK, What about λ-calculus?

λ-calculus

- A really small programming language
- Consisted of only anonymous curried functions

λ-calculus syntax

Constructor	Lambda
Variable	`x`, `y`
Abstraction	`λx. BODY`
Application	`A B`

Application is left associative

$a\ b\ c = (a\ b)\ c$

$a\ b\ c \neq a\ (b\ c)$

Remember this?

```
x => x
x => x(x)
(x => x)(x => x)
f => x => x
f => x => f(x)
f => x => f(f(f(f(x))))
```

In λ-calculus syntax

```
λx. x
λx. x x
(λx. x) (λx. x)
λf. λx. x
λf. λx. f x
λf. λx. f (f (f (f x)))
```

From λ to Factorial

From λ to Factorial

```
function fact(n) {
  if (n === 0) {
    return 1
  } else {
    return n * fact(n-1)
  }
}
```

So we need

- ☐ Encoding for Booleans
- ☐ Encoding for Natural Numbers
- ☐ Function to check if number is zero
- ☐ Multiplication
- ☐ Predecessor
- ☐ If/Then/Else
- ☐ Recursion

Encoding Booleans

That is: encode True and False

P.S.: There are infinite ways of doing this

What are booleans used for?

Branching

Pick one of two paths

```
λ?. ???
```

```
λthen. λelse. ???
```

True: `λthen. λelse. then`

False: `λthen. λelse. else`

Church Booleans

In Javascript

```
const tru = t => f => t
const fals = t => f => f
```

Operations on Booleans

Not Function

Not Function

a	not a
true	false
false	true

Not Function

```
const not = a ? false : true
```

Not Function

```
λa. a FALSE TRUE
```

Not Function

```
const not = b => b(fals)(tru)
```

Other Functions

Other Functions

```
const and = a => b => a ? b : false
const or = a => b => a ? true : b
const xor = a => b => a ? b : (b ? false : true)
```

Other Functions

```
const and = a => b => a(b)(fals)
const or = a => b => a(tru)(b)
const xor = a => b => a(b)(b(fals)(tru))
```

What we have so far

- ☑ Encoding for Booleans
- ☐ Encoding for Natural Numbers
- ☐ Function to check if number is zero
- ☐ Multiplication
- ☐ Predecessor
- ☐ If/Then/Else
- ☐ Recursion

Encoding Natural Numbers

That is: encode 0, 1, 2, ...

P.S.: There are also infinite ways of doing this

What natural numbers are used for?

Counting things

Church Numerals

Count the number of times a function is applied to a given input

```
N => λf. λx. F_APPLIED_TO_X_N_TIMES
```

Number	Encoding
0	`λf. λx. x`
1	λf. λx. f x
2	`λf. λx. f (f x)`
3	λf. λx. f (f (f x))
4	`λf. λx. f (f (f (f x)))`

Constructing Natural Numbers

- We need zero
- And a way to get N+1 given N (successor)

Zero

```
λf. λx. x
```

Zero

```
const zero = f => x => x
```

Successor Function

`λn. ???`

Successor function

```
λn. λf. λx. ???
```

Successor function

```
λn. λf. λx. ??? (n f x)
```

Successor function

```
λn. λf. λx. f (n f x)
```

Successor Function

```
const succ = n => f => x => f(n(f)(x))
```

Let's try it out

```
const one = succ(zero)
const two = succ(one)
const three = succ(two)
const hundred = succ(fromNumber(10))
```

What we have so far

- ☑ Encoding for Booleans
- ☑ Encoding for Natural Numbers
- ☐ Function to check if number is zero
- ☐ Multiplication
- ☐ Predecessor
- ☐ If/Then/Else
- ☐ Recursion

Functions on Natural Numbers

- Is Zero
- Multiplication
- Predecessor

- Sorry, only 30 minute talk 😢

Is Zero

`λn. n (λx. FALSE) TRUE`

Multiplication

`λa. λb. (λf. λx. b (a f) x)`

Predecessor Function

`λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)`

```
const isZero = n => n(x => fals)(tru)
const mul = a => b => (f => x => b(a(f))(x))
const pred = n => f => x => n(g => h => h(g(f)))(y => x)(u => u)
```

What we have so far

- ☑ Encoding for Booleans
- ☑ Encoding for Natural Numbers
- ☑ Function to check if number is zero
- ☑ Multiplication
- ☑ Predecessor
- ☐ If/Then/Else
- ☐ Recursion

If Then Else

- Isn't it just applying booleans to the branches?
- No, because Javascript is **eagerly evaluated**
- So we'll need to do one trick

```
λb. λthen. λelse. b then else ANY_VALUE
```

```javascript
const ifte = b => t => e => b(t)(e)(b)

// Example usage
ifte(myBool)(b => thenThis)(b => elseThis)
```

What we have so far

- ☑ Encoding for Natural Numbers
- ☑ Encoding for Booleans
- ☑ Function to check if number is zero
- ☑ Multiplication
- ☑ Predecessor
- ☑ If/Then/Else
- ☐ Recursion

Recursion

The problem with recursion

- All functions are anonymous
- Therefore, we cannot refer a function to itself by name
- Solution: **Abstract yourself**

Abstract Yourself

- `λx. ?CALL_MY_SELF?`
- `λmyself. λx. myself x`

```
λfact. λn.
  IFTE
    (IS_ZERO n)
    (λx. ONE)
    (λx. (MUL n (fact (PRED n))))
```

```javascript
const absFact = fact => n =>
  ifte(isZero(n))(
    (t) => one
  )(
    (e) => mul(n)(fact(pred(n)))
  )
```

Fixed-point combinators

- Must have the property:

```
fix f = f (fix f)
```

- Which if you expand once, you have:

```
      = f (f (fix f))
```

- And if you keep expanding:

```
      = f (f (f (... (f (fix f)) ...)))
```

- Good news: there is also infinite of these

Y-Combinator

```
Y = λf (λx. f (x x)) (λx. f (x x)))
```

In Javascript

```
// Can't I just do
const Y = f => (x => f(x(x)))(x => f(x(x)))

// And apply
const fact = Y(absFact)

// And use
fact(...)
```

Z-Combinator

- Works in eagerly evaluated environments
- Trick is similar to what we've done in `ifte`
- Defined as: `Z = λf. (λx. x x) (λx. f (λy. x x y))`

```
const Z = f => (x => x(x))(x => f(y => x(x)(y)))
const fact = Z(absFact)
toNumber(fact(fromNumber(3)))
toNumber(fact(fromNumber(4)))
toNumber(fact(fromNumber(5)))
```

Key Takeaways

- Lambda Calculus is about functions and functions all the way down
- And you can compute anything* with only that
- It is the theory behind functional programming languages such as Haskell

Thanks Folks!

github.com/bamorim/lambda-talk