Lambda Calculus - Church Numerals (5A)

Young Won Lim
6/28/23

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Church encoding (1)

Church encoding is a means of
representing **data** and **operators** in the lambda calculus.

The **Church numerals** are a representation of
the **natural numbers** using lambda notation.

The method is named for Alonzo Church,
who first encoded data in the lambda calculus this way.

https://en.wikipedia.org/wiki/Church_encoding

Church encoding (1)

Terms that are usually considered primitive in other notations
(such as integers, booleans, pairs, lists, and tagged unions)
are mapped to higher-order functions under Church encoding.

The Church-Turing thesis asserts that
any computable **operator** (and its **operands**)
can be represented under Church encoding.

In the untyped lambda calculus
the only primitive data type is the **function**.

https://en.wikipedia.org/wiki/Church_encoding

Typed and untyped calculus (1)

Lambda calculus may be untyped or typed.

In typed lambda calculus, **functions** can be applied
only if they are capable of accepting the given input's "**type**" of data.

Typed lambda calculi are weaker than the untyped lambda calculus,
in the sense that typed lambda calculi can express less
than the untyped calculus can,
but on the other hand typed lambda calculi
allow more things to be proven;

https://en.wikipedia.org/wiki/Church_encoding

Typed and untyped calculus (2)

in the simply typed lambda calculus it is, for example, a **theorem**
that every **evaluation strategy** terminates
for every simply typed lambda-term,

whereas **evaluation** of untyped lambda-terms need not terminate.

One reason there are many different typed lambda calculi
has been the desire to do more (of what the untyped calculus can do)
without giving up on being able to prove **strong theorems**
about the calculus.

https://en.wikipedia.org/wiki/Church_encoding

Typed lambda calculus (3)

A typed lambda calculus is a typed formalism
that uses the lambda-symbol (λ \lambda)
to denote anonymous **function abstraction**.

In this context, types are usually objects of a syntactic nature
that are assigned to lambda terms;

the exact nature of a type depends
on the calculus considered
(see Kinds of typed lambda calculi).

https://en.wikipedia.org/wiki/Church_encoding

Typed lambda calculus (3)

From a certain point of view,
typed lambda calculi can be seen
as refinements of the untyped lambda calculus

but from another point of view,
they can also be considered the more fundamental theory
and untyped lambda calculus
a special case with only one type.

https://en.wikipedia.org/wiki/Church_encoding

Typed lambda calculus (4)

Typed lambda calculi are foundational programming languages
and are the base of typed functional programming languages
such as ML and Haskell and, more indirectly,
typed imperative programming languages.

Typed lambda calculi play an important role
in the design of type systems for programming languages;

here typability usually captures desirable properties of the program,
e.g. the program will not cause a memory access violation.

https://en.wikipedia.org/wiki/Church_encoding

Typed lambda calculus (5)

Typed lambda calculi are closely related
to mathematical logic and proof theory
via the Curry–Howard isomorphism and
they can be considered as
the internal language of classes of categories,
e.g. the simply typed lambda calculus is
the language of Cartesian closed categories (CCCs).

https://en.wikipedia.org/wiki/Church_encoding

Formalism

In the philosophy of mathematics,
formalism is the view that holds
that statements of mathematics and logic
can be considered to be statements
about the consequences of the manipulation of strings
(alphanumeric sequences of symbols, usually as equations)
using established manipulation rules.

https://en.wikipedia.org/wiki/Formalism_(philosophy_of_mathematics)

Classification of typed / untyped lambda calculus (1)

- Untyped lambda calculus -- no logical interpretation
- Simply typed lambda calculus -- intuitionistic propositional logic
- Polymorphic lambda calculus -- pure second-order logic

  ie, without first-order quantifiers

- Dependent types -- generalization of first-order logic
- Calculus of constructions -- generalization of higher-order logic

https://cstheory.stackexchange.com/questions/5834/classification-of-typed-untyped-lambda-calculi

Classification of typed / untyped lambda calculus (2)

Type dependency is more general than first-order quantification,
since it turns proofs into objects you can quantify over.

Lambda calculi corresponding to ordinary intuitionistic FOL exist,
but are not widely used enough to have a special name
-- people tend to go straight to dependent types.

https://cstheory.stackexchange.com/questions/5834/classification-of-typed-untyped-lambda-calculi

Classification of typed / untyped lambda calculus (3)

Pure untyped λ-calculus is Turing complete, i.e.,
a partial number-theoretic map is computable
if, and only if, it is definable in the untyped λ-calculus.

The computational power of typed λ-calculus is much smaller.

For example, if we add a type of natural numbers **nat**
to the typed λ-calculus, together with
0, successor, and primitive recursion,
we get what is commonly known as Gödel's T.

It computes the primitive recursive functions only (and they are all total).

https://cstheory.stackexchange.com/questions/5834/classification-of-typed-untyped-lambda-calculi

Classification of typed / untyped lambda calculus (4)

The untyped λ-calculus does not have a reasonable interpretation under the Curry-Howard correspondence, while the typed λ

– calculus corresponds precisely to intuitionistic propositional calculus.

Models of typed λ

– calculus are precisely the cartesian-closed categories.

Models of the untyped λ-calculus are less well-behaved.

While it is possible to talk about them,

they are certainly not studied as widely as cartesian-closed categories.

https://cstheory.stackexchange.com/questions/5834/classification-of-typed-untyped-lambda-calculi

Use (1)

A *straightforward implementation* of Church encoding
slows some access operations from *O(1)* to *O(n)*,
where *n* is the size of the data structure,
making Church encoding impractical.

Research has shown that
this can be addressed by targeted optimizations,
but most functional programming languages instead
expand their intermediate representations
to contain algebraic data types.

https://en.wikipedia.org/wiki/Church_encoding

Use (2)

Nonetheless Church encoding is
often used in theoretical arguments,
as it is a natural representation
for partial evaluation and theorem proving.

Operations can be typed using higher-ranked types,
and primitive recursion is easily accessible.

The assumption that functions are the only primitive data types
streamlines many proofs.

https://en.wikipedia.org/wiki/Church_encoding

Use (3)

Church encoding is complete but only representationally.

Additional functions are needed
to translate the representation into common data types,
for display to people.

It is not possible in general to decide
if two functions are extensionally equal
due to the undecidability of equivalence from Church's theorem.

https://en.wikipedia.org/wiki/Church_encoding

Use (4)

The translation may apply the function in some way
to retrieve the value it represents, or
look up its value as a literal lambda term.

Lambda calculus is usually interpreted as using intensional equality.

There are potential problems with the interpretation of results
because of the difference between the intensional
and extensional definition of equality.

https://en.wikipedia.org/wiki/Church_encoding

Church Numerals (1)

Church numerals are
the representations of natural numbers under Church encoding.

The higher-order function that represents natural number ***n***
is a function that maps any function ***f*** to its **n-fold composition**.

In simpler terms, the "value" of the numeral is equivalent
to the number of times the function encapsulates its argument.

$$f^{\circ n} = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ times}}$$

https://en.wikipedia.org/wiki/Church_encoding

Church Numerals (2)

Starting with 0 not applying the function at all,

proceed with 1 applying the function once,

2 applying the function twice,

3 applying the function three times, etc.:

$$f^{\circ n} = \overbrace{f \circ f \circ \cdots \circ f}^{n \text{ times}}$$

$$n\ f\ x = f^{(n)}(x)$$

https://en.wikipedia.org/wiki/Church_encoding

Church Numerals (3)

All Church numerals are functions

that take two parameters. (f and x)

Church numerals 0, 1, 2, ..., are defined in the lambda calculus.

Number	Function definition	Lambda expression
0	$0\ f\ x = f^{(0)}(x) = x$	$\lambda f.\ \lambda x.\ x$
1	$1\ f\ x = f^{(1)}(x) = f\ x$	$\lambda f.\ \lambda x.\ f\ x$
2	$2\ f\ x = f^{(2)}(x) = f\ (f\ x)$	$\lambda f.\ \lambda x.\ f\ (f\ x)$
3	$3\ f\ x = f^{(3)}(x) = f\ (f\ (f\ x))$	$\lambda f.\ \lambda x.\ f\ (f\ (f\ x))$
n	$n\ f\ x = f^{(n)}(x) = \underbrace{f\ (f\ \ldots\ (f}_{n \text{ times}}\ x)\ldots)$	$\lambda f.\ \lambda x.\ \underbrace{f\ (f\ \ldots\ (f}_{n \text{ times}}\ x)\ldots)$

https://en.wikipedia.org/wiki/Church_encoding

Church Numerals (4)

The Church numeral 3 represents

the action of applying any given **function** three times to a **value**.

The supplied **function f** is first applied

to a supplied **parameter x**

and then successively to its **own result**.

x → f x → f (f x) → f (f (f x))

3 = 3 f x

= f (f (f x))

= $f^{(3)}$ (x)

= λf. λx. f (f (f x))

The higher-order function that represents natural number ***n*** is a **function**

that maps any function ***f***

to its **n-fold composition**.

https://en.wikipedia.org/wiki/Church_encoding

Church Numerals (5)

The **end result** is not the **numeral 3**

unless the **supplied parameter** happens to be **0**

and the function is a **successor function**

The **function** itself,

and not its **end result**,

is the Church numeral 3.

The Church numeral 3 means

simply to do anything three times.

It is an ostensive demonstration of what is meant by "three times".

ostensive : directly or clearly demonstrative

The higher-order function that represents natural number ***n*** is a **function**

that maps any function ***f***

to its **n-fold composition**.

https://en.wikipedia.org/wiki/Church_encoding

Why definition (1)

Church wasn't trying to be *practical*.
He was trying to *prove* results about
the expressive power of **lambda calculus** —
that in principle *any possible computation*
can be *done* in **lambda calculus**,
hence **lambda calculus** can serve
as a theoretical foundation for the study of computability.

For this purpose, it was necessary
to encode numbers as **lambda expressions**,
in such a way that things like the successor function
are easily definable.

https://stackoverflow.com/questions/41978590/why-the-definition-of-churchs-numerals

Why definition (2)

This was a key step in showing
the equivalence of **lambda calculus** and
Gödel's recursive function theory
(which was about computable functions on the natural numbers).

Church numerals are basically a convenient
albeit not very readable encoding of numbers.

In some sense, there isn't any very deep logic to it.

The claim isn't that **1** in its essence is **λf. λx. f x**,
but that the latter is a serviceable encoding of the former.

https://stackoverflow.com/questions/41978590/why-the-definition-of-churchs-numerals

Why definition (3)

This doesn't mean that it is an *arbitrary encoding*.

There is a definite logic to it.

The most natural way to encode a number n is

by something which involves n.

Church numerals use ***n*** function applications.

The natural number ***n*** is represented

by the higher order function

which applies a function ***n times*** to an input.

https://stackoverflow.com/questions/41978590/why-the-definition-of-churchs-numerals

Why definition (4)

1 is encoded by a function applied once,

2 by a function applied twice and so on.

It is a very *natural* encoding,

especially in the context of lambda calculus.

Furthermore, the fact that it is easy

to define *arithmetic* on them

streamlines the *proof*

that **lambda calculus** is equivalent to

recursive functions.

https://stackoverflow.com/questions/41978590/why-the-definition-of-churchs-numerals

Why definition (5)

To see this in practice, you can run
the following **Python3** script:

```
#some Church numerals:

  ZERO   = lambda f: lambda x: x
  ONE    = lambda f: lambda x: f(x)
  TWO    = lambda f: lambda x: f(f(x))
  THREE  = lambda f: lambda x: f(f(f(x)))
```

https://stackoverflow.com/questions/41978590/why-the-definition-of-churchs-numerals

Why definition (6)

```
# function to apply these numerals to:
def square(x): return x**2

# so ZERO(square), ONE(square), etc. are functions
# apply these to 2 and print the results:

print(ZERO(square)(2), ONE(square)(2),
      TWO(square)(2),THREE(square)(2))

Output:
2 4 16 256
```

```
ZERO   = lambda f: lambda x: x
ONE    = lambda f: lambda x: f(x)
TWO    = lambda f: lambda x: f(f(x))
THREE  = lambda f: lambda x: f(f(f(x)))
```

Note that these numbers have been obtained
by squaring the number two 0 times, 1 times,
2 times, and 3 times respectively.

https://stackoverflow.com/questions/41978590/why-the-definition-of-churchs-numerals

Church numeral (1)

Natural numbers are non-negative.

Given a successor function, **next**, which adds one,

we can define the natural numbers

in terms of **zero** and **next**:

1 = (next 0)

2 = (next 1) = (next (next 0))

3 = (next 2) = (next (next (next 0)))

and so on.

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (2)

Therefore a number **n** will be that

number of successors of **zero**.

Just as we adopted the convention **TRUE = first**,

and **FALSE = second**, we adopt the following convention:

zero = λf.λx.x

one = λf.λx.(f x)

two = λf.λx.(f (f x))

three = λf.λx.(f (f (f x)))

four = λf.λx.(f (f (f (f x))))

1 = (next 0)

2 = (next 1) = (next (next 0))

3 = (next 2) = (next (next (next 0)))

f ← next

x ← zero

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (3)

a "unary" representation of the natural numbers,

such that **n** is represented

as **n** applications of the function **f** to the argument **x**.

zero = $\lambda f.\lambda x.x$

one = $\lambda f.\lambda x.(f\ x)$

two = $\lambda f.\lambda x.(f\ (f\ x))$

three = $\lambda f.\lambda x.(f\ (f\ (f\ x)))$

four = $\lambda f.\lambda x.(f\ (f\ (f\ (f\ x))))$

This representation is refered to as

CHURCH NUMERALS.

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (3)

We can define the function **next** as follows:

next = λn.λf.λx.(f ((n f) x))

and therefore **one** as follows:

one = (next zero)

=> (λn.λf.λx.(f ((n f) x)) zero)

=> λf.λx.(f ((zero f) x))

=> λf.λx.(f ((λg.λy.y f) x)) (* alpha conversion avoids clash *)

=> λf.λx.(f (λy.y x))

=> λf.λx.(f x)

zero	**= λf.λx.x**
one	**= λf.λx.(f x)**
two	**= λf.λx.(f (f x))**
three	**= λf.λx.(f (f (f x)))**
four	**= λf.λx.(f (f (f (f x))))**

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (4)

and **two** as follows:

two = (next one)

=> (λn.λf.λx.(f ((n f) x)) one)

=> λf.λx.(f ((one f) x))

=> λf.λx.(f ((λg.λy.(g y) f) x)) (* alpha conversion avoids clash *)

=> λf.λx.(f (λy.(f y) x)

=> λf.λx.(f (f x))

```
val next = fn n => fn f => fn x => (f ((n f) x));
    next = λn.λf.λx.(f ((n f) x))
val next = (f ((n f) x))
```

next = λn.λf.λx.(f ((n f) x))

zero	**= λf.λx.x**
one	**= λf.λx.(f x)**
two	**= λf.λx.(f (f x))**
three	**= λf.λx.(f (f (f x)))**
four	**= λf.λx.(f (f (f (f x))))**

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (5-1)

NOTE that **((two g) y) = (g (g y))**.

So if we had some function, say

one that increments n:

inc = λn.(n+1)

two = λf.λx.(f (f x))

two g = (λf.λx.(f (f x))) g
= λx.(g (g x))

((two g) y)
= (λx.(g (g x))) y
= (g (g y))

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (5-2)

then we can get a feel for a Church Numeral as follows:

((two inc) 0)
=> ((λf.λx.(f (f x)) inc) 0)
=> (λx.(inc (inc x) 0)
=> (inc (inc 0))
=> (λn.(n+1) (λn.(n+1) 0))
=> (λn.(n+1) (0 + 1))
=> ((0 + 1) + 1)
=> 2

two = λf.λx.(f (f x))

two g = (λf.λx.(f (f x))) g
= λx.(g (g x))

((two g) y)
= (λx.(g (g x))) y
= (g (g y))

inc = λn.(n+1)

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (6-1)

We are now in a position to define **addition** in terms of next:

add = λm.λn.λf.λx.((((m next) n) f) x);

next = λn.λf.λx.(f ((n f) x))

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (6-2)

Therefore four may be computed as follows:

```
four = ((add two) two)
    => ((λm.λn.λf.λx.((((m next) n) f) x) two) two)
    => (λn.λf.λx.((((two next) n) f) x) two
    => λf.λx.((((two next) two) f x)
    => λf.λx.((((λg.λy.(g (g y)) next) two) f x)
    => λf.λx.(((λy.(next (next y)) two) f) x)
    => λf.λx.(((next (next two)) f) x)
    => λf.λx.(((next (next (next  (next zero)))) f) x)

    => λf.λx.(((next (λn.λf.λx.(f ((n f) x)) two)) f) x)
```

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

```
add = λm.λn.λf.λx.((((m next) n) f) x);

next = λn.λf.λx.(f ((n f) x))

two    = λf.λx.(f (f x))
       = λg.λy.(g (g y))

two g  = (λf.λx.(f (f x))) g
       = λx.(g (g x))

((two g) y)
       = (λx.(g (g x))) y
       = (g (g y))

one = (next zero)
two = (next one)
    = (next  (next zero))
```

Church numeral (7)

```
mult = λm.λn.λx.(m (n x))

six  = ((mult two) three)
     => ((λm.λn.λx.(m (n x)) two) three)
     => (λn.λx.(two (n x) three)
     => λx.(two (three x))
     => λx.(two (λg.λy.(g (g (g y))) x))
     => λx.(two λy.(x (x (x y))))
     => λx.( λf.λz.(f (f z)) λy.(x (x (x y))) )
     => λx.λz.(λy.(x (x (x y))) (λy.(x (x (x y))) z))
     => λx.λz.(λy.(x (x (x y))) (x (x (x z))) )
     => λx.λx.(x (x (x (x (x (x z))) )))
```

```
two    = λf.λx.(f (f x))
       = λg.λy.(g (g y))

three  = λf.λx.(f (f (f x)))
       = λg.λy.(g (g (g y)))
```

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (8-1)

```
power = λm.λn.(m n);                    n^m

nine  = ((power two) three)             three^two
      => ((λm.λn.(m n) two) three)
      => (λn.(two n) three)
      => (two three)
      => (λf.λx.(f (f x)) three)
      => λx. (three (three x))
      => λx. (three (λg.λy.(g (g (g y))) x))
      => λx. (three λy.(x (x (x y))))
```

```
two     = λf.λx.(f (f x))
        = λg.λy.(g (g y))

three   = λf.λx.(f (f (f x)))
        = λg.λy.(g (g (g y)))
```

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (8-2)

=> λx. (three λy.(x (x (x y))))

=> λx. (λg.λz.(g (g (g z))) λy.(x (x (x y))))

=> λx.λz.(λy.(x (x (x y))) (λy.(x (x (x y))) (λy.(x (x (x y))) z)))

=> λx.λz.(λy.(x (x (x y))) (λy.(x (x (x y))) (x (x (x z)))))

=> λx.λz.(λy.(x (x (x y))) (x (x (x (x (x (x z)))))))

=> λx.λz.(x (x (x (x (x (x (x (x (x z)))))))))

two = λf.λx.(f (f x))

= λg.λy.(g (g y))

three = λf.λx.(f (f (f x)))

= λg.λy.(g (g (g y)))

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Church numeral (9-1)

The following lambda **function** tests for zero:

first = λx.λy.λz.x

third = λx.λy.λz.z

iszero = λn.((n third) first)

= λn.((n λx.λy.λz.z) first)

= λn.((n λx.λy.λz.z) λx.λy.λz.x)

https://www.cs.unc.edu/~stotts/723/Lambda/church.html

Arithmetic in lambda calculus (1)

the most commonly defined **Church numerals**

0 := λf.λx.x

1 := λf.λx.f x

2 := λf.λx.f (f x)

3 := λf.λx.f (f (f x))

using the alternative syntax presented above in Notation:

0 := λfx.x

1 := λfx.f x

2 := λfx.f (f x)

3 := λfx.f (f (f x))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (2)

A **Church numeral** is a **higher-order function** –
it takes a single-argument function **f**,
and returns another single-argument function.

The Church numeral **n** is a function
that takes a function **f** as argument
and returns the **n**-th composition of **f**,
i.e. the function **f** composed with itself **n** times.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (3)

This is denoted **$f^{(n)}$** and is in fact

the n-th power of **f** (considered as an operator);

$f^{(0)}$ is defined to be the identity function.

Such repeated compositions (of a single function **f**)

obey the laws of exponents,

which is why these numerals can be used for arithmetic.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (4)

0 := λf.λx.x

1 := λf.λx.f x

2 := λf.λx.f (f x)

3 := λf.λx.f (f (f x))

$f^{(0)}$ is defined to be the identity function.

In Church's original lambda calculus,

the formal parameter of a lambda expression

was required to occur at least once in the function body,

which made the above definition of 0 impossible.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (5)

One way of thinking about the **Church numeral n**,
which is often useful when analysing programs,
is as an instruction '**repeat n times**'.

For example, using the **PAIR** and **NIL** functions defined below,
one can define a function that constructs
a (linked) list of **n** elements all equal to **x**
by repeating 'prepend another **x** element' **n** times,
starting from an empty list.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (6)

The lambda term is

λn.λx.n (PAIR x) NIL

By varying what is being repeated,
and varying what argument that function
being repeated is applied to,
a great many different effects can be achieved.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (7)

We can define a successor function,

which takes a Church numeral n and returns n + 1

by adding another application of f,

where '(mf)x' means the function 'f' is applied 'm' times on 'x':

SUCC := λn.λf.λx.f (n f x)

Because the m-th composition of f composed

with the n-th composition of f gives the m+n-th composition of f,

addition can be defined as follows:

PLUS := λm.λn.λf.λx.m f (n f x)

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (8)

PLUS can be thought of as a function
taking two natural numbers as arguments
and returning a natural number; it can be verified that

PLUS 2 3 and 5

are β-equivalent lambda expressions.

Since adding m to a number n can be accomplished
by adding 1 m times, an alternative definition is:

PLUS := λm.λn.m SUCC n [23]

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Arithmetic in lambda calculus (9)

Similarly, multiplication can be defined as

MULT := λm.λn.λf.m (n f)

Alternatively

MULT := λm.λn.m (PLUS n) 0

since multiplying m and n is the same as
repeating the add n function m times
and then applying it to zero.

Exponentiation has a rather simple rendering in Church numerals, namely

POW := λb.λe.e b

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf