Lambda Calculus - Functions of Church Numerals (7A)

Young Won Lim
12/30/23

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

This document was produced by using LibreOffice.

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

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

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 = **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))))**

This representation is refered to as

CHURCH NUMERALS.

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

Church numeral (4)

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

next = λn.λf.λx.(f ((n f) x)) **= 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

Calculation with Church Numerals

Arithmetic operations on numbers may be represented by **functions** on Church numerals.

These **functions** may be defined in lambda calculus, or implemented in most functional programming languages

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

Functions on Church numerals (1) **plus**

The **addition function plus(m, n) = m + n**

uses the identity $f^{\circ(m+n)}(x) = f^{\circ(m)}(f^{\circ(n)}(x))$

$$\text{plus} \equiv \lambda m.\ \Lambda n\ .\lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$

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

Functions on Church numerals (2) **succ, mult**

The **successor function succ(n) = n + 1**
is β-equivalent to **(plus 1)**

succ ≡ λn. λf. λx. f (n f x)

The **multiplication function mult(m, n) = m * n**
uses the identity $f^{\circ(m * n)}(x) = (f^{\circ(n)})^{\circ(m)}(x)$

mult ≡ λm. λn. λf. λx. m (n f) x

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

Functions on Church numerals (3) exp

The **exponentiation function** **exp(m, n) =** $\mathbf{m}^{\mathbf{n}}$

is given by the definition of Church numerals,

$$n\ h\ x = h^n\ x$$

In the definition substitute **h** → **m**, **x** → **f** to get **n m f =** $\mathbf{m}^{\mathbf{n}}$ and,

$$n\ m\ f = m^n\ f$$

$$\text{exp}\ m\ n = m^n = n\ m$$

which gives the lambda expression,

$$\text{exp} \equiv \lambda m.\ \lambda n.\ n\ m$$

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

Functions on Church numerals (4) pred

The **pred(n)** function is more difficult to understand.

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

A Church numeral applies a function **n** times.

The predecessor function must return
a **function** that applies its parameter **n - 1** times.

This is achieved by building a container around **f** and **x**,
which is initialized in a way that omits
the application of the function the first time.

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

Functions on Church numerals (5) minus

The subtraction function can be written

based on the predecessor function.

minus ≡ λm. λn. (n pred) m

minus m n = n pred m

minus 4 3 = 3 pred 4
= (pred (pred (pred 4)))
= (pred (pred 3))
= (pred 2)
= 1

minus 3 2 = 2 pred 3
= (pred (pred 3))
= (pred 2)
= 1

minus 2 2 = 2 pred 2
= (pred (pred 2))
= (pred 1)
= 0

minus 1 2 = 2 pred 1
= (pred (pred 1))
= (pred 0)
= 0

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

Summary : functions on Church numerals (1)

Function	Algebra	Identity	Function definition	
Successor	n + 1	f^{n+1} x = f (f^{n} x)	succ n f x	= f (n f x)
Addition	m + n	f^{m+n} x = f^{m} (f^{n} x)	plus m n f x	= m f (n f x)
Multiplication	m * n	f^{m*n} x = $(f^{m})^{n}$ x	multiply m n f x	= m (n f) x
Exponentiation	m^{n}	n m f = m^{n} f x	exp m n f x	= (n m) f x
Predecessor	n - 1	inc n con = val (f^{n-1} x)	if (n == 0) 0 else (n - 1)	
Subtraction	m - n	f^{m-n} x = $(f^{-1})^{n}$ (f^{m} x)	minus m n	= (n pred) m

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

Summary : functions on Church numerals (2)

Identity	Function definition		Lambda expressions	
$f^{n+1}\ x = f\ (f^{n}\ x)$	succ n f x	= f (n f x)	λn. λf. λx. f (n f x)	...
$f^{m+n}\ x = f^{m}\ (f^{n}\ x)$	plus m n f x	= m f (n f x)	λm. λn. λf. λx. m f (n fx)	λm. λn. n succ m
$f^{m*n}\ x = (f^{m})^{n}\ \ x$	multiply m n f x	= m (n f) x	λm. λn. λf. λx. m (n f) x	λm. λn. λf. m (n f)
$n\ m\ f = m^{n}\ \ f\ x$	exp m n f x	= (n m) f x	λm. λn. λf. λx. (n m) f x	λm. λn. n m
$inc\ n\ con = val\ (f^{n-1}\ x)$	if (n == 0) 0 else (n - 1)		λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)	
$f^{m-n}\ x = (f^{-1})^{n}\ (f^{m}\ x)$	minus m n	= (n pred) m	...	λm. λn. n pred m

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

Predecessor function using **pair** (1)

We can now define the predecessor function
combining some of the functions introduced above.

When looking for the predecessor of **n**,
the general strategy will be to create a **pair** **(n, n-1)**
and then pick the <u>second</u> element (**n-1**) of the **pair** as the result.

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

Predecessor function using **pair** (2)

A **pair (a, b)** can be represented in λ-calculus
using the function **(λz. z a b)**

We can extract the first element of the pair
from the expression applying this function to **T**
(λz.z a b) T = T a b = a

and the second applying the function to **F**
(λz.z a b) F = F a b = b

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

Predecessor function using **pair** (3)

pair (n, n-1) →

p = pair n n-1 = (λz. z n n-1)

p T = (λz. z n n-1) T = T n n-1 = n -- first element

p F = (λz. z n n-1) F = F n n-1 = n-1 -- second element

A **pair (a, b)**

pair a b = (λz. z a b)

extract the first element

(λz.z a b) T = T a b = a

extract the second

(λz.z a b) F = F a b = b

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

Predecessor function using **pair** (4)

The following function

generates the **pair (n+1, n)**

from the **pair (n, n-1)** (= p)

$\Phi \equiv (\lambda p.\ \lambda z.\ z\ (S\ (p\ T))\ (p\ T)\ )$

pair (n, n-1) ➡ **pair (n+1, n)**

p ➡ Φ

the **pair (n, n-1)** is the argument p in the function

$p = pair\ n\ n\text{-}1 = (\lambda z.\ z\ n\ n\text{-}1)$

A **pair (a, b)**

$pair\ a\ b = (\lambda z.\ z\ a\ b)$

extract the first element

$(\lambda z.z\ a\ b)\ T = T\ a\ b = a$

extract the second

$(\lambda z.z\ a\ b)\ F = F\ a\ b = b$

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

Predecessor function using **pair** (5)

Φ ≡ (λp. λz. z (S (p T)) (p T))

The subexpression **p T**

extracts the first element from the pair **p**.

thus **n** from the **pair (n, n-1)**

A new pair is formed using this element **n**,

n is incremented **S (p T)**

for the first position of the new pair (**n+1**)

n is copied **(p T)**

for the second position of the new pair. (**n**)

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))))

Successor function

next = λn.λf.λx.(f ((n f) x))

S = λn.λf.λx.(f (n f x))

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

Predecessor function using **pair** (6)

Φ ≡ (λp. λz. z (S (p T)) (p T))

The predecessor of a number **n** is obtained

by applying **n** times

the function **Φ**

to the pair **(λ.z 0 0)**

thus get the new pair **(λ.z n n-1)**

and then selecting the second member **n-1**

of the new pair **(λ.z n n-1)**

n = 3

pair 0 0

pair 1 0

pair 2 1

pair 3 2 → **2**

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

Predecessor function using **pair** (7)

Φ ≡ (λp. λz. z (S (p T)) (p T))

P ≡ (λn. n Φ (λz.z 0 0) F)

Notice that using this approach the predecessor of zero is zero.

This property is useful for the definition of other functions.

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

P 1 = 1 Φ (λz.z 0 0)
= Φ (λz.z 0 0)
= (λz.z 1 0)

P 2 = 2 Φ (λz.z 0 0)
= Φ (Φ (λz.z 0 0))
= Φ (λz.z 1 0)
= (λz.z 2 1)

P 3 = 3 Φ (λz.z 0 0)
= Φ (Φ (Φ (λz.z 0 0)))
= Φ (λz.z 2 1)
= (λz.z 3 2)

P 1 F = 0
P 2 F = 1
P 3 F = 2

Predecessor function using **pair** (8)

Φ ≡ (λp. λz. z (S (p T)) (p T))

(λx. λz. z (succ (first x)) (first x))

(λx. pair (succ (first x)) (first x))

pred =
λn . second
(n (pair zero zero)
(λx . pair (succ (first x))
(first x))
)

pair a b = (λz. z a b)

https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf

Predecessor function using **pair** (9)

pred = λn . second (n (λx. pair (succ (first x)) (first x))
(pair zero zero))

P ≡ (λn. n Φ (λz.z 0 0) F)
Φ ≡ (λp. λz. z (S (p T)) (p T))

pair a b = (λz. z a b)

- How does this work?
  the **pair ⟨a, b⟩** encodes the fact that **(pred a) = b**

http://web.cecs.pdx.edu/~black/CS311/Lecture%20Notes/Lambda%20Calculus.pdf

Church numeral (11)

The lambda function pred delivers the predecessor of a Church Numeral:

pair = λx.λy.λf.((f x) y);

prefn = λf.λp.((pair (f (p first))) (p first))

pred = λn.λf.λx.(((n (prefn f)) (pair x x)) second)

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

Predecessor function using **shift** and **increment**

As an example of the use of pairs,
the shift-and-increment function
that maps **(m, n)** to **(n, n + 1)** can be defined as

Φ2 := λx.PAIR (SECOND x) (SUCC (SECOND x))

which allows us to give perhaps the most transparent version
of the predecessor function:

PRED := λn.FIRST (n Φ2 (PAIR 0 0)).

n = 3

pair 0 0

pair 0 1

pair 1 2

pair 2 3 → **2**

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

Predecessor function using **conditionals** (1)

the **predecessor** function can be defined as:

PRED := λn.n (λg. λk. ISZERO (g 1) k (PLUS (g k) 1)) (λv.0) 0

which can be verified by showing inductively

that

n (λg. λk. ISZERO (g 1) k (PLUS (g k) 1)) (λv.0)

is the

add n − 1 function for **n > 0**.

ISZERO := λn.n (λx.FALSE) TRUE

true ≡ λa.λb.a

false ≡ λa.λb.b

ISZERO (g 1)

If True,

True k (PLUS (g k) 1)

selects **k**

Else,

False k (PLUS (g k) 1)

selects **(PLUS (g k) 1)**

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

Predecessor function using **conditionals** (2)

A **predicate** is a function that <u>returns</u> a boolean value.

the **ISZERO** predicate

returns **TRUE**

if its argument is the Church numeral **0**,

returns **FALSE**

if its argument is *any other* Church numeral:

ISZERO := λn. n (λx. FALSE) TRUE

n=0: **λf.λy y (λx. FALSE) TRUE** → **TRUE**

n=1: **λf.λy f y (λx. FALSE) TRUE** → **FALSE**

λf.λx x	**0**
λf.λx f x	**1**
λf.λx f (f x)	**2**
λf.λx f (f (f x))	**3**
λf.λx f (f (f (f x)))	**4**

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

Predecessor function (1)

The **predecessor** **function** used in the Church encoding is,

$$\text{pred}(n) = \begin{cases} 0 & \text{if } n = 0, \\ n - 1 & \text{otherwise.} \end{cases}$$

a way of applying the function 1 fewer time.

A numeral **n** applies the **function f**, **n** times to **x**. **n f x**

the **predecessor** function must use the numeral **n** (**n-1**) **f x**

to apply the function **n-1** times.

0	**0 f x = x**
1	**1 f x = f x**
2	**2 f x = f (f x)**
3	**3 f x = f (f (f x))**
4	**4 f x = f (f (f (f x)))**

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

Predecessor function (2)

Before <u>implementing</u> the predecessor function,

here is a scheme

that **wraps** the **value** **x, (f x), (f (f x)), …**

in a **container** **function** **value**

value x

value (f x)

value (f (f x))

value (f (f (f x)))

value $(f^{n-1}\ x)$ = **value((n-1) f x)**

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

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

Predecessor function (3)

We will define <u>new</u> **functions** to use in place of **f** and **x**,

called **inc** and **init** , **const**

The general recurrence rule is,

inc (value v) = value (f v)

If there is also a function (called **extract**)

to retrieve the value from the container

extract (value v) = v

value x

value (f x)

value (f (f x))

inc (value x) = value (f x)

inc (value (f x)) = value (f (f x))

inc (value (f (f x)) = value (f (f (f x))

extract (value x) = x

extract (value (f x)) = (f x)

extract (value (f (f x))) = (f (f x))

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

Predecessor function (4)

value x can be either **init** or **inc const**

init = value x

inc init = value (f x)

inc (inc init) = value (f (f x))

inc const = value x

inc (inc const) = value (f x)

value x

value (f x)

value (f (f x))

value (f (f (f x)))

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

Predecessor function (5)

n-fold composition

init = value x

1 inc init = value (f x)

2 inc init = value (f (f x))

3 inc init = value (f (f (f x)))

extract(1 inc init) = f x

extract(2 inc init) = (f (f x))

extract(3 inc init) = (f (f (f x)))

extract (n inc init) = n f x

inc const = value x

1 inc const = value x

2 inc const = value (f x)

3 inc const = value (f (f x))

extract(1 inc const) = x

extract(2 inc const) = (f x)

extract(3 inc const) = (f (f x))

extract (inc const) = (n-1) f x

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

Predecessor function (6)

The left hand side of the table shows

a **numeral n** applied to **inc** and **init**.

Number	**Using init**	**using const**
0	**init = value x**	
1	**inc init = value (f x)**	**inc const = value x**
2	**inc (inc init) = value (f (f x))**	**inc (inc const) = value (f x)**
3	**inc (inc (inc init)) = value (f (f (f x)))**	**inc (inc (inc const)) = value (f (f x))**
n	**n inc init = value (f^{n} x) = value (n f x)**	**n inc const = value (f^{n-1} x) = value((n-1) f x)**

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

Predecessor function (7)

pred = λn. λf. λx. extract(n inc const)

= λn. λf. λx. extract (value ((n−1) f x))

= λn. λf. λx. (n − 1) f x

= λn. (n − 1)

1 inc const = value x = value (0 f x)

2 inc const = value (f x) = value (1 f x)

3 inc const = value (f (f x)) = value (2 f x)

extract(1 inc const) = x = (0 f x)

extract(2 inc const) = (f x) = (1 f x)

extract(3 inc const) = (f (f x)) = (2 f x)

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

Predecessor function (8)

as **inc** delegates ***calling*** of **f**
to its **container value argument**,

f v
value v

inc (value v) = value (f v)

in order to skip the first application of **f**,
we can arrange that on the first application
inc receives a special **container const**
that ignores its **argument**

inc const = value x

init = value x
inc const = value x

samenum
= λn. λf. λx. extract (n inc init)
= λn. λf. λx. extract (value (n f x))
= λn. λf. λx. n f x
= λn. n

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

Predecessor function (9)

Call this new initial container **const**.

The right hand side of the above table

shows the expansions of **n inc const**.

Then by replacing **init** with **const** in the expression

for the same function we get the **predecessor** function,

init = value x

inc const = value x

value x

value (f x)

value (f (f x))

value (f (f (f x)))

value $(f^{n-1}\ x)$ **= value((n-1) f x)**

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

1 inc const = value x

2 inc const = value (f x)

3 inc const = value (f (f x))

extract(1 inc const) = x

extract(2 inc const) = (f x)

extract(3 inc const) = (f (f x))

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

Predecessor function (10)

Then **extract** may be used to define the **samenum** function as,

samenum = λn. λf. λx. **extract** (n **inc** **init**)
= λn. λf. λx. **extract** (**value** (n f x))
= λn. λf. λx. n f x
= λn. n

n f x = f (f ... (f (f (f x))) ...) ⇒ n

The samenum function is not intrinsically useful.

value = λv. (λh. h v)
extract k = k λu. u
inc = λg. λh. h (g f)
init = λh. h x
const = λu. x

init = value x
inc const = value x

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

Predecessor function (11)

samenum = λn. λf. λx. extract (n inc init)
= λn. λf. λx. extract (value (n f x))
= λn. λf. λx. n f x
= λn. n

1 inc init = value x = value (1 f x)
2 inc init = value (f x) = value (2 f x)
3 inc init = value (f (f x)) = value (3 f x)

extract(1 inc init) = x = (1 f x) ⇒ 1
extract(2 inc init) = (f x) = (2 f x) ⇒ 2
extract(3 inc init) = (f (f x)) = (3 f x) ⇒ 3

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

Predecessor subfunctions

pred = λn. λf. λx. extract(n inc const)
= λn. λf. λx. extract (n value x)
= λn. λf. λx. extract (value ((n−1) f x))
= λn. λf. λx. (n − 1) f x
= λn. (n − 1)

1 inc const = value x
2 inc const = value (f x)
3 inc const = value (f (f x))

n inc const = value ((n-1) f x)
n value x = value ((n−1) f x)

inc const = value x
inc (value x) = value (f x)

n value x = value ((n−1) f x)

extract (value x) = x

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

Predecessor function definition

pred = λn. λf. λx. extract(n inc const)
= λn. λf. λx. extract (n value x))
= λn. λf. λx. extract (value ((n−1) f x))
= λn. λf. λx. (n − 1) f x
= λn. (n − 1)

pred = λn. λf. λx. extract (n inc const)
= λn. λf. λx. (n inc const) (λu. u)
= λn. λf. λx. (n (λg. λh. h (g f)) const) (λu. u)
= λn. λf. λx. (n (λg. λh. h (g f)) (λu. x)) (λu. u)
= λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

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

Definitions of predecessor sub-functions

the functions **inc**, **init**, **const**, **value** and **extract**

may be defined as follows

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

Which gives the lambda expression for **pred** as,

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

value x

extract (value x) = x

inc (value v) = value (f v)

init = value x

inc const = value x

1 inc const = value x

2 inc const = value (f x)

3 inc const = value (f (f x))

extract(1 inc const) = x

extract(2 inc const) = (f x)

extract(3 inc const) = (f (f x))

n inc const = value ((n-1) f x)

n value x = value ((n−1) f x)

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

value container

The **value container** applies a **function** h to its **value**.

value v h = h v

(value v) h = h v

so,

value = λv. (λh. h v)

2 argument **v** and **h**

1st argument **v**

2nd argument **h**

return value **h v**

value v h = h v

value	**= λv. (λh. h v)**
extract k	**= k λu. u**
inc	**= λg. λh. h (g f)**
init	**= λh. h x**
const	**= λu. x**

value v	
value v f	**= f v**
value (f v) f	**= f (f v)**
value (f (f v)) f	**= f (f (f v))**

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

inc (1)

The **inc** function should take a **value** containing **v**,

and return a new **value** containing **f v**.

inc (value v) = value (value v f)

= value (f v)

Letting **g** be the **value container**,

g = value v

then,

g f = value v f = f v ⇐ **value = λv. (λh. h v)**

value = λv. (λh. h v)

extract k = k λu. u

inc = λg. λh. h (g f)

init = λh. h x

const = λu. x

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

inc (2)

g = value v
g f = value v f = f v

value = λv. (λh. h v)

inc g = value (g f) = value (value v f)
= value (f v)

inc g = value (f v)
inc g h = h (f v)

g f = value v f = f v
(inc g) h = value (f v) h = h (f v)

inc = λg. λh. h (g f)
inc g = λh. h (g f)
inc g h = h (f v)

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

value x
value (f x)
value (f (f x))
value (f (f (f x)))

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

inc (3)

value = λv. (λh. h v)
value x = (λh. h x)
value x h = h x

inc = λg. λh. h (g f)
inc g = λh. h (g f) = λh. h (f x) ⇐ g = value x
inc g h = h (f x)

value = λv. (λh. h v)
extract k = k λu. u
inc = λg. λh. h (g f)
init = λh. h x
const = λu. x

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

extract (1)

The **value** may be extracted by applying the **identity** function,

I = λu. u

value v I = v

value v I = I v
= λu. u v
= v

value v h = h v

k = value v

extract k = k I = v

value	**= λv. (λh. h v)**
extract k	**= k λu. u**
inc	**= λg. λh. h (g f)**
init	**= λh. h x**
const	**= λu. x**

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

extract (2)

I = λu. u

k = value v

extract k = k λu. u

extract k = k I

= value v I

= I v

= λu. u v

= v

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

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

const (1)

To implement **pred**, the **init** function is replaced with the **const** that does not apply **f**.

We need **const** to satisfy,

inc const = value (const f) ⇐ **inc g = value (g f)**

= value x

Which is satisfied if,

const f = x

Or as a lambda expression,

const = λu. x

value = λv. (λh. h v)

extract k = k λu. u

inc = λg. λh. h (g f)

init = λh. h x

const = λu. x

init = value x

inc const = value x

inc init = value (f x)

inc init = λg. λh. h (g f) λh. h x

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

const (2)

inc const = value (const f) ⬅ inc g = value (g f)

= value x

inc = λg. λh. h (g f)

value = λv. (λh. h v)

const = λu. x

inc const = λh. h (const f)

= λh. h (λu. x f)

= λh. h x

= value x

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

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

Predecessor subfunctions verification

1 inc const = value x

2 inc const = value (f x)

3 inc const = value (f (f x))

1 inc const = 1 (λg. λh. h (g f)) (λu. x)
= λh. h (λu. x f) = λh. h x = value x

2 inc const = 2 (λg. λh. h (g f)) (λu. x) = (λg. λh. h (g f)) (λh. h x)
= λh. h ((λh. h x) f) = λh. h (f x) = value (f x)

3 inc const = 3 (λg. λh. h (g f)) (λu. x) = (λg. λh. h (g f)) (λh. h (f x))
= λh. h ((λh. h (f x)) f) = λh. h (f (f x)) = value (f (f x))

n inc const = inc ((n-1) inc const)

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

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

Multiple applications of **inc** sub-function

1 inc const = value x
2 inc const = value (f x)
3 inc const = value (f (f x))

1 inc const = λh. h (λu. x f) = λh. h x = value x
2 inc const = λh. h ((λh. h x) f) = λh. h (f x) = value (f x)
3 inc const = λh. h ((λh. h (f x)) f) = λh. h (f (f x)) = value (f (f x))

1 inc const = λh. h x
2 inc const = λh. h (f x)
3 inc const = λh. h (f (f x))

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

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

Extracting multiple applications of **inc** sub-function

extract (1 inc const) = value x = x
extract (2 inc const) = value (f x) = f x
extract (3 inc const) = value (f (f x)) = f (f x)

extract (1 inc const) = (1 inc const) I
extract (2 inc const) = (2 inc const) I
extract (3 inc const) = (3 inc const) I

(1 inc const) (λu. u) = λh. h x (λu. u) = x
(2 inc const) (λu. u) = λh. h (f x) (λu. u) = f x
(3 inc const) (λu. u) = λh. h (f (f x)) (λu. u) = f (f x)

value = λv. (λh. h v)
extract k = k λu. u
inc = λg. λh. h (g f)
init = λh. h x
const = λu. x

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

Predecessor function definition

pred = λn. λf. λx. extract (n inc const)
= λn. λf. λx. (n inc const) (λu. u)
= λn. λf. λx. (n (λg. λh. h (g f)) const) (λu. u)
= λn. λf. λx. (n (λg. λh. h (g f)) (λu. x)) (λu. u)
= λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)

g ⬅ (λu. x)
h ⬅ (λu. u)

value	= λv. (λh. h v)
extract k	= k λu. u
inc	= λg. λh. h (g f)
init	= λh. h x
const	= λu. x

extract k = k I
I = λu. u
inc = λg. λh. h (g f)
const = λu. x

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

Predecessor function verification

pred = λn. λf. λx. extract (n inc const)
= λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)

1 (λg. λh. h (g f)) (λu. x) = λh. h ((λu. x) f) = λh. h x
2 (λg. λh. h (g f)) (λu. x) = (λg. λh. h (g f)) λh. h x — different h
= (λh. h (λh. h x f)) = λh. h (f x)
3 (λg. λh. h (g f)) (λu. x) = (λg. λh. h (g f)) λh. h (f x) — different h
= λh. h (λh. h (f x) f) = λh. h (f (f x))

1 (λg. λh. h (g f)) (λu. x) (λu. u) = λh. h x (λu. u) = x
2 (λg. λh. h (g f)) (λu. x) (λu. u) = λh. h (f x) (λu. u) = (f x)
3 (λg. λh. h (g f)) (λu. x) (λu. u) = λh. h (f (f x)) (λu. u) = (f (f x))

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

PRED Predicate (1)

The predecessor function defined by

PRED n = n − 1 for a positive integer **n** and

PRED 0 = 0 when **n** is equal to zero

is considerably more difficult.

The formula

PRED := λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)

can be validated by showing *inductively*

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

PRED Predicate (2)

PRED := λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)

can be validated by showing *inductively* that

if **T** denotes **(λg.λh. h (g f))**, g f(n) → f(n-1)

T (λu.x) = (λg. λh. h (g f)) (λu.x) = (λh. h (f (x))) for **n > 0**.

g f = (λu.x) f = f

then **$T^{(n)}$ (λu.x) = (λh. h ($f^{(n-1)}$ (x)))** for **n > 0**.

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

PRED Predicate (2)

Two other definitions of **PRED** are given below,

one using **conditionals** and the other using **pairs**.

With the predecessor function, subtraction is straightforward.

Defining

SUB := λm.λn.n PRED m,

SUB m n yields **m − n** when **m > n** and **0** otherwise.

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