Lambda Calculus - Recursions (9A)

Young Won Lim
4/19/24

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

This document was produced by using LibreOffice.

Encoding Conditionals (1)

consider how to encode a conditional expression of the form:

if P then A else B

i.e., the value of the whole expression is either **A** or **B**,

depending on the value of **P**

this conditional expression can be represented by

using a lambda expression as follows

COND P A B

where **COND**, **P**, **A** and **B** are all lambda expressions.

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (2)

COND P A B

COND is a function of 3 arguments

that works by applying **P** to (**A** and **B**)

(i.e., **P** itself chooses **A** or **B**):

COND == λp.λa.λb.p a b

(where == means "is defined to be").

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (3)

To make this definition work correctly,

we must define the representations of **true** and **false** carefully

since the lambda expression **P**

that **COND** applies to its arguments **A** and **B**

will reduce to either **TRUE** or **FALSE**

when **TRUE** is applied to **a** and **b** we want it to <u>return</u> **a** (first)

when **FALSE** is applied to **a** and **b** we want it to <u>return</u> **b**. (second)

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (4)

let **TRUE** be a function of two arguments
that ignores the second argument
and returns the first argument,

let **FALSE** be a function of two arguments
that ignores the first argument
and returns the second argument:

TRUE == λx.λy.x
FALSE == λx.λy.y

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (5)

COND TRUE M N

Note that this expression should evaluate to **M**.

substituting our definitions for **COND** and **TRUE**,

and evaluating the resulting expression

the sequence of beta-reductions is shown below

in each case, the redex about to be reduced is indicated

by *underlining* the formal parameter and

the argument that will be substituted in for that parameter. NO

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (6)

$(\underline{\lambda p}.\lambda a.\lambda b.\ p\ a\ b)\ (\underline{\lambda x.\lambda y.\ x})\ M\ N \rightarrow \beta$

$(\underline{\lambda a}.\lambda b.\ (\lambda x.\lambda y.x)\ a\ b)\ \underline{M}\ N \rightarrow \beta$

$(\underline{\lambda b}.\ (\lambda x.\lambda y.x)\ M\ b)\ \underline{N} \rightarrow \beta$

$(\underline{\lambda x}.\lambda y.\ x)\ \underline{M}\ N \rightarrow \beta$

$(\underline{\lambda y}.\ M)\ \underline{N} \rightarrow \beta$

M

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Division (1-1)

Division of natural numbers may be implemented by,

n / m = if n ≥ m then 1 + (n − m) / m
else 0

Calculating **n – m** takes many beta reductions.

Unless doing the reduction by hand,
this doesn't matter that much,
but it is preferable to not have to do
this calculation **(n – m)** twice.

9 / 3 = 1 + (9 – 3) / 3
= 1 + (1 + (6 – 3) / 3)
= 1 + (1 + (1 + 0 / 3))
= 1 + (1 +(1 + 0))

n / m = if n ≥ m then 1 + (n – m) / m
else 0

computing the condition **(n ≥ m)**
involves **(n – m)** calculation

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

Division (1-2)

The simplest predicate for testing numbers is **IsZero**

so consider the condition.

IsZero (minus n m)

But this condition is equivalent to **n ≤ m**, not **n < m**.

minus n m = m pred n = 0 if n ≤ m

If this expression is used

then the mathematical definition of division given above

is translated into function on Church numerals as,

minus m n = n pred m

minus 4 3 = 3 pred 4

= (pred (pred (pred 4)))

= (pred (pred 3))

= (pred 2)

= 1

IsZero (minus 3 1) = 0	**3 > 1**	**2**
IsZero (minus 3 2) = 0	**3 > 2**	**1**
IsZero (minus 3 3) = 1	**3 = 3**	**0**
IsZero (minus 3 4) = 1	**3 < 4**	**0**
IsZero (minus 3 5) = 1	**3 < 5**	**0**

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

Division (2-1)

n / m	=	if	n ≥ m	then	1 + (n − m) / m
				else	0
n / m	=	if	n < m	then	0
				else	1 + (n − m) / m
(n-1)/m	=	if	n ≤ m	then	0
				else	1 + (n − m) / m

If **IsZero** **(minus n m)** is used

a single call to **(minus n m)** is possible

but the result gives the value of **(n-1) / m**.

(minus n m) can be utilized

in computing **1 + (n − m) / m**

correct condition: **n < m**

modified condition: **n ≤ m**

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

Division (2-2)

divide1 n m f x =
    (λd. IsZero d (0 f x) (f (divide1 d m f x)))   (minus n m)

d ← n − m

IsZero d ⟹ IsZero (minus n m)

TRUE ⟹ (λx.λy.x) (0 f x) (f (divide1 d m f x))
= (0 f x)

FALSE ⟹ (λx.λy.y) (0 f x) (f (divide1 d m f x))
= (f (divide1 d m f x))

(n-1)/m = if n ≤ m then 0
else 1 + (n − m) / m

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

Division (2-3)

```
divide1 n m f x =
    (λd. IsZero d (0 f x) (f (divide1 d m f x)))      (minus n m)
```

```
divide1 9 3 f x
    = IsZero 6 (0 f x) (f (divide1 6 3 f x)) = (f (divide1 6 3 f x))
divide1 6 3 f x
    = IsZero 3 (0 f x) (f (divide1 3 3 f x)) = (f (divide1 3 3 f x))
 divide1 3 3 f x
    = IsZero 0 (0 f x) (f (divide1 0 3 f x)) = (0 f x) = x
```

```
9 / 3  = 1 + (9 – 3) / 3
       = 1 + (1 + (6 – 3) / 3)
       = 1 + (1 + (1 + 0 / 3))
       = 1 + (1 +( 1 + 0))
```

```
divide1 9 3 f x
       = (f (divide1 6 3 f x))

       = (f (f (divide1 3 3 f x)))

       = (f (f (0 f x)))

       = (f (f x))
```

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

Division (3-1)

add **1** to **n** before calling **divide**.

divide n = divide1 (succ n)

divide1 10 3 f x

= IsZero 7 (0 f x) (f (divide1 7 3 f x)) = (f (divide1 7 3 f x))

divide1 7 3 f x

= IsZero 4 (0 f x) (f (divide1 4 3 f x)) = (f (divide1 4 3 f x))

divide1 4 3 f x

= IsZero 1 (0 f x) (f (divide1 1 3 f x)) = (f (divide1 1 3 f x))

divide1 1 3 f x

= IsZero 0 (0 f x) (f (divide1 1 3 f x)) = (0 f x) = x

divide1 9 3 f x

= (f (divide1 7 3 f x))

= (f (f (divide1 4 3 f x)))

= (f (f (f (divide1 1 3 f x))))

= (f (f (f x)))

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

Division (3-2)

add **1** to **n** before calling **divide.**

divide n = divide1 (succ n)

divide1 is a recursive definition.

divide1 n m f x =

(λd. IsZero d (0 f x) (f (divide1 d m f x)))   (minus n m)

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

Division (4)

The **Y combinator** may be used to implement the recursion.

Create a new function called **div** by;

In the left hand side **divide1** → **div c**

In the right hand side **divide1** → **c**

divide1 n m f x =

(λd. IsZero d (0 f x) (f (divide1 d m f x))) (minus n m)

div = λc. λn. λm. λf. λx.

(λd. IsZero d (0 f x) (f (c d m f x))) (minus n m)

div c = λn. λm. λf. λx.

(λd. IsZero d (0 f x) (f (c d m f x))) (minus n m)

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

Division (5)

Then,

divide = λn. divide1 (succ n)

where,

divide1 = Y div succ = λn. λf. λx. f (n f x) Y

= λf. (λx. F (x x)) (λx. f (x x)) 0

= λf. λx. x IsZero

= λn. N (λx. False) true

true ≡ λa. λb. a false ≡ λa. λb. b

minus = λm. λn. n pred m pred

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

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

Division (6)

Gives,

```
divide =
    λn. ((λf. (λx. x x) (λx. f (x x)))
        (λc. λn. λm. λf. λx.
            (λd. (λn. n (λx. (λa. λb. b)) (λa. λb . a))
                    d ((λf. λx. x) f x) (f (c d m f x)))
            ((λm. λn. n (λn. λf. λx . n (λg. λh. h (g f))
                        (λu. x) (λu. u)) m) n m)
        ))
        ((λn. λf. λx. f (n f x)) n)
```

Or as text, using \ for λ,

```
divide =
(\n.((\f.(\x.x x) (\x.f (x x)))
    (\c.\n.\m.\f.\x.
        (\d.(\n.n (\x.(\a.\b.b)) (\a.\b.a))
                d ((\f.\x.x) f x) (f (c d m f x)))
        ((\m.\n.n (\n.\f.\x.n (\g.\h.h (g f))
                    (\u.x) (\u.u)) m) n m)
    ))
    ((\n.\f.\x. f (n f x)) n))
```

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

Division (6)

Gives,

divide = λn. ((λf. (λx. x x) (λx. f (x x))) (λc. λn. λm. λf. λx. (λd. (λn. n (λx. (λa. λb. b)) (λa. λb. a)) d ((λf. λx. x) f x) (f (c d m f x))) ((λm. λn. n (λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)) m) n m))) ((λn. λf. λx. f (n f x)) n)

Or as text, using \ for λ,

divide = (\n.((\f.(\x.x x) (\x.f (x x))) (\c.\n.\m.\f.\x.(\d.(\n.n (\x.(\a.\b.b)) (\a.\b.a)) d ((\f.\x.x) f x) (f (c d m f x))) ((\m.\n.n (\n.\f.\x.n (\g.\h.h (g f)) (\u.x) (\u.u)) m) n m))) ((\n.\f.\x. f (n f x)) n))

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

Division (7)

For example, 9/3 is represented by

divide (\f.\x.f (f (f (f (f (f (f (f (f x))))))))) (\f.\x.f (f (f x)))

Using a lambda calculus calculator,

the above expression reduces to 3, using normal order.

(\f.\x.f (f (f (x))))

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

Recursion (1-1)

recursion:

the definition of a function using the function itself.

A function definition containing itself inside itself, by value,

leads to the whole value being of infinite size.

Other notations which support recursion natively overcome this

by referring to the function definition by name.

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

Recursion (1-2)

Lambda calculus cannot express this:

all functions are anonymous in lambda calculus,

so we can't refer by name to a value which is yet to be defined,

inside the lambda term defining that same value.

however, a lambda expression can receive itself

as its own argument, for example in **(λx.x x) E**.

Here **E** should be an abstraction,

applying its parameter to a value to express recursion.

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

Recursion (1-3)

Consider the factorial function **F(n)** recursively defined by

F(n) = 1, if n = 0;   else n * F(n-1).

In the lambda expression which is to represent the function **F(n)**,
a parameter (typically the first one) will be assumed
to receive the lambda expression itself as its value,
so that calling it - applying it to an argument
will amount to recursion.

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

Recursion (2-1)

Thus to achieve recursion,

the *intended-as-self-referencing* argument

(called **r** here) must always be passed to itself

within the function body, at a call point:

G := λr. λn. (1, if n = 0;   else n × (r r (n−1)))

with **r r x = F x = G r x** to hold,

so **r = G** and

F := G G = (λx.x x) G

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

Recursion (2-2)

F(n) = 1, if n = 0; else n × F(n − 1).

G := λr. λn.(1, if n = 0; else n × (r r (n−1)))

with **r r x = F x = G r x** to hold, so **r = G** and

F := G G = (λx.x x) G

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

Recursion (3-1)

The self-application achieves replication here,

passing the function's lambda expression

on to the next invocation as an argument value,

making it available to be referenced and called there.

This solves it but requires re-writing

each recursive call as self-application.

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

Recursion (3-2)

We would like to have a generic solution,

<u>without</u> a need for any <u>re-writes</u>:

G := λr. λn.(1, if n = 0; else n × (r (n−1)))

with **r x = F x = G r x** to hold, so **r = G r =: FIX G** and

F := FIX G where **FIX g :=** (**r** where **r = g r**) **= g (FIX g)**

so that

FIX G = G (FIX G) = (λn.(1, if n = 0; else n × ((FIX G) (n−1))))

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

Recursion (4)

Given a lambda term with first argument representing recursive call (e.g. **G** here), the fixed-point combinator **FIX** will return a self-replicating lambda expression representing the recursive function (here, **F**).

The function does not need to be explicitly passed to itself at any point, for the self-replication is arranged in advance, when it is created, to be done each time it is called.

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

Recursion (5)

Thus the original lambda expression (**FIX G**) is re-created inside itself, at call-point, achieving self-reference.

In fact, there are many possible definitions for this **FIX** operator, the simplest of them being:

$$\mathbf{Y := \lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))}$$

$$\begin{aligned}\mathbf{Y\ g} &\mathbf{= (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))} \\ &\mathbf{= g\ (\lambda x.\ (x\ x))\ (\lambda x.g\ (x\ x))}\end{aligned}$$

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

Recursion (6)

In the lambda calculus, **Y g** is a fixed-point of **g**, as it expands to:

Y g

(λh.(λx.h (x x)) (λx.h (x x))) g

(λx.g (x x)) (λx.g (x x))

g ((λx.g (x x)) (λx.g (x x)))

g (Y g)

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

Recursion (7)

Now, to perform our recursive call to the factorial function,

we would simply call **(Y G) n**, where **n** is the number

we are calculating the factorial of.

Given **n = 4**, for example, this gives:

(Y G) 4

G (Y G) 4

(λr.λn.(1, if n = 0; else n × (r (n−1)))) (Y G) 4

(λn.(1, if n = 0; else n × ((Y G) (n−1)))) 4

1, if 4 = 0; else 4 × ((Y G) (4−1))

4 × (G (Y G) (4−1))

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

Recursion (8)

4 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (4−1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3−1)))
4 × (3 × (G (Y G) (3−1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (3−1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2−1))))
4 × (3 × (2 × (G (Y G) (2−1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (2−1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1−1)))))
4 × (3 × (2 × (1 × (G (Y G) (1−1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (1−1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0−1))))))
4 × (3 × (2 × (1 × (1))))
24

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

Recursion (9)

Every recursively defined function can be seen
as a fixed point of some suitably defined function
closing over the recursive call with an extra argument,
and therefore, using **Y**, every recursively defined function
can be expressed as a lambda expression.

In particular, we can now cleanly define the subtraction, multiplication
and comparison predicate of natural numbers recursively.

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