

Section 4

Plan:

- ① Review λ -calculus
- ② Encoding natural numbers in λ
- ③ Encoding recursion in λ

* * *

λ calculus is a Turing complete programming language (PL)

- can simulate a Turing machine.

- can simulate any modern PL, like C, Java, Python, OCaml, etc.

The syntax of λ calculus consists of

1. Variables: x, y, z, \dots

2. Function abstraction: $\lambda x. x$

3. Function application: $(\lambda x. x) y$

The semantics of λ calculus can be given by:

how a λ calculus program/term should be executed
evaluated.

let rec eval (t: term) : term =

match t with

| $v \rightarrow v$

| $\lambda v. t' \rightarrow \lambda v. t'$

| $t_1, t_2 \rightarrow$

(match eval t_1 , eval t_2 with

subst $v t_2' t_1'$

$[v \mapsto t_2'] t_1'$

| $\lambda x. t', t_2' \rightarrow$ replace v with t_2' in t'

| $- , - \rightarrow$ stuck)

The semantics of λ -calculus terms crucially depend on the substitution function

$$[\nu \rightarrow t_\nu] t$$

Attempt #1 $[\nu \rightarrow t_\nu] t = \text{match } t \text{ with}$

$$| u \rightarrow \text{if } u \neq \nu \text{ then } t_\nu \text{ else } t$$

$$| (t_1, t_2) \rightarrow ([\nu \rightarrow t_\nu] t_1, [\nu \rightarrow t_\nu] t_2)$$

$$| \lambda u. t' \rightarrow \lambda u. [\nu \rightarrow t_\nu] t'$$

Problem ?

What if $\nu = u$, e.g.

$$\begin{aligned} \text{eval}(\lambda x. \lambda x. x) y &= [x \mapsto y](\lambda x. x) \\ \Downarrow \\ \lambda x. x &= \lambda x. y \end{aligned}$$

Functions are equiv
up to renaming.

Attempt #2 $[\nu \rightarrow t_\nu] t = \text{match } t \text{ with}$

$$| u \rightarrow \dots$$

$$| (t_1, t_2) \rightarrow \dots$$

$$| \lambda u. t' \rightarrow \text{if } \nu \neq u \text{ then } \lambda u. [\nu \rightarrow t_\nu] t' \text{ else } t$$

Problem ?

What if u is free in t_ν , e.g.

$$\text{eval}(\lambda y. \lambda x. y) x = [y \mapsto x](\lambda x. y)$$

$$(\lambda y. \lambda z. y) x = \lambda x. x$$

$$\Downarrow$$

A variable is free if it isn't a formal param.

Attempt #3 : $[v \rightarrow t_v] t$ = match t with

$\lambda u. t' \rightarrow$

if $u \neq v$ then

if $u \notin FV(t_v)$ then

$\lambda u. [v \rightarrow t_v] t'$

else

?

else t .



Problem ?

We're stuck in the second case !

eval $(\lambda y. \lambda x. y) x = [y \mapsto x] \lambda x. y$

We need to perform λ -renaming.

Encoding things in λ -calculus.

- Booleans :

$$\text{true} = \underline{\lambda x. \lambda y. x}$$

$$\text{false} = \underline{\lambda x. \lambda y. y}$$

$$\text{if-then-else} = \lambda b. \lambda x. \lambda y. \underline{b \ x \ y}$$

- Integers :

n $\xrightleftharpoons[\text{decode}]{\text{encode}}$ apply a function n times

(demo)

		apply f
0	\longleftrightarrow	$\lambda f. \lambda x. x$ 0 times
1	\longleftrightarrow	$\lambda f. \lambda x. fx$ 1 time
2	\longleftrightarrow	$\lambda f. \lambda x. f(fx)$ 2 times
3	\longleftrightarrow	$\lambda f. \lambda x. f(f(fx))$
:		:
n	\longleftrightarrow	$\lambda f. \lambda x. f(\underbrace{\dots}_{n \text{ times}}(fx))$

Church encoding

Addition :

$$\lambda a. \lambda b. (\lambda f. \lambda x. \underline{b \ f \ (af \ x)})$$

church encoding of $a+b$

" $(a \ f) \ x$ " applies f to x a times

" $(b \ f) \ y$ " applies f to y b times

" $b \ f \ (af \ x)$ " applies f to x $(a+b)$ times

Note: The encode & decode functions are not part of λ -calculus; they just help us read an encoding more easily

Multiplication:

$$\lambda a. \lambda b. (\lambda f. \lambda x. \underline{b \ (af) \ x})$$

applies f to x a · b times

Hint: Partial application.

Exponentiation:

$$\lambda a. \lambda b. (\lambda f. \lambda x. \underline{(b \ a) \ f \ x})$$

applies f to x a^b times

Encoding let expressions:

let $x = e_1$ "Substitute x for e_1 in e_2 "
in e_2

$[x \mapsto e_1] e_2$

↑

$(\lambda x. e_2) e_1$

Encoding recursion

Consider our favorite recursive function, factorial :

$$n! = n \times (n-1) \times (n-2) \times \dots \times 1$$

let fact = $\lambda n.$

if $n=0$ then 1

else $n * \text{fact}(n-1)$

Cannot define this in λ because the recursive call to fact is an unbound reference.

* But we can define what's called the generator for a recursive function:

$$\boxed{\text{mystery} = \lambda f. \quad \lambda n. \quad \text{if } n=0 \text{ then } 1 \text{ else } f * (n-1)}$$

"lifting the recursive call into a formal parameter"

$$\text{Then, } \text{fact} = \underbrace{\text{mystery}(\text{mystery}(\dots(\text{mystery} \text{ bad})))}_{\infty \text{ many times}} \uparrow \text{e.g. } \lambda x. 0$$

But programs are finite.

Hope : There's a magical term "magic" in λ -calculus s.t. (magic mystery) behaves just like fact.

What properties does "magic" have ?

Properties of "magic"

magic mystery \leftrightarrow fact

< hopefully >

$$= \lambda n. \text{if } n=0 \text{ then } 1 \quad < \text{expand def of fact} >$$
$$\text{else } n * \underline{\text{fact}(n-1)}$$

$$= \lambda n. \text{if } n=0 \text{ then } 1 \quad < \text{by our hope} >$$
$$\text{else } n * (\text{magic mystery})(n-1)$$

$$\text{mystery} = \lambda f. \lambda n.$$

$$\text{if } n=0 \text{ then } 1$$
$$\text{else } n * f(n-1)$$

$$= (\lambda f. \lambda n. \text{if } n=0 \text{ then } 1) (\text{magic mystery})$$
$$\text{else } n * f(n-1) \quad < \text{by } \lambda\text{-abstraction and application} >$$

$$= \text{mystery}(\text{magic mystery}) \quad < \text{by def of mystery} >$$

$$\boxed{\text{magic mystery} \leftrightarrow \text{mystery}(\text{magic mystery})}$$

1. (\Rightarrow) Tells us the behavior of (magic mystery)

It unrolls the body of mystery by supplying a copy of itself as the argument.

$$\text{magic mystery} = (\lambda n. \text{if } n=0 \text{ then } 1) \\ \text{else } n * (\text{magic mystery})(n-1)$$

Recursion via self-replication.

magic mystery = mystery (magic mystery)

2. (\Leftarrow) Let's say $x = \underline{\text{magic mystery}}$.

Then $\text{mystery}(x) = x$.

x is called a fixed-point of function mystery .



If f is a function, and $f(x) = x$ for some x ,
then x is called a fixed-point of f

Thus, magic "computes" the fixed-point of any generator.

In λ -calculus, 'magic' are the fixed-point combinators,
such as the Y-combinator and Z-combinator :

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

Summary, to encode a recursive function f in λ -calculus

Example.

1. Write down its generator, which is
non-recursive. Call it g .

$f = \text{fact}$

$g = \text{mystery}$

2. Apply your favorite fixed-point combinator \mathcal{F}
to the generator g .

$\mathcal{F} = \text{magic}$

3. The application $\mathcal{F}g$ is a term
in λ -calculus that behaves just like
the recursive function f .

$\mathcal{F}g = \text{magic}$
 mystery

$\Leftrightarrow \text{fact}$

Recursive functions are fixed points of their generators

Note: in λ^+ , magic is the built-in fix operator.