(fp C_2 1.0)

(iter 1.0 (C_2 1.0))

(iter 1.0 (avg 1.0 (C_1 1.0)))
2.0

(iter 1.0 (avg 1.0 2.0))

(iter 1.0 1.5)

Local Names:

"let" bindings are created simultaneously & "let" body is evaluated with respect to those binding.

Lambda Calculus

① Core of all functional languages.
② Useful for specifying semantics.
-has only 3 syntactic rules in lambda calculus.

M → x (Variable)
(term) → λx. M (abstraction) [takes variable x, body M]
to tell we defining fxn

$M_1 \cdot (M_2)$ (application.

the 4th one (M) → parenthesis to represent ~~this has to~~

optional ← the 5th one is constants - - control flow.

→ identity function

λx . x → in scheme (lambda (x) x)

to apply this function on an argument

(λx. x+1) 2

↳ fall into 3rd category $M_1 \cdot M_2$.

How is this evaluated? the ~~argument~~ parameter is bound to argument.

How, to take multiple arguments?

~~xxx~~ ~~xxx~~ (define add (lambda (x)
(lambda (y) (+ x y))))

(λx (λy. x+y))

$(\lambda x.\ \lambda y.\ x+y)\ 2\ \ 3$

$(\lambda y.\ 2+y)\ 3$

$(2+3)$

5

Precedence & Associativity rules

① Application binds tighter than abstraction.

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

$\neq (\lambda x.x)\ y$ → If we want this to be done just use '(' ')'

② Application associates to left

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

$\neq x\ (y\ z)$

β-reduction

$(\lambda x.\ M)\ N = [N/x]\ M$

↳ replace occurances of x in M with 'x' → General substitution model type.

$(\lambda x.\ x+1)\ 2 = [2/x]\ x+1$

1) $(\lambda x.x)\ ((\lambda x.x)\ (\lambda z.\ (\lambda x.x)\ z))$ M_1 applied on M_2

M_1 M_2

$=_\beta (\lambda x.x)(\lambda z.\ (\lambda x.x)\ z)$

$=_\beta \lambda z.\ (\lambda x.x)\ z$

$=_\beta \lambda z\ z$ → Normal form → Cannot reduce any further

$\lambda xyz \equiv \lambda x.\ \lambda y.\ \lambda yz$ (takes 3 arguments)

② $(\lambda xyz.\ x\ z\ (y\ z))\ (\lambda x.\ x)\ (\lambda x.\ x)$

$\rightarrow_\beta$ rule ②

$=_\beta \lambda yz.\ (\lambda x.\ x)\ z\ (y\ z)\ (\lambda x.\ x)$

Met-1

$=_\beta \lambda yz.\ z\ (y\ z)\ (\lambda x.\ x)$ leftmost innermost redex

$=_\beta \lambda z.\ z\ ((\lambda x.\ x)\ z)$ call by value.

$= \beta \lambda z.\ z\ z$

Met-2

$=_\beta \lambda z.\ (\lambda x.\ x)\ z\ ((\lambda x.\ x)\ z)$

$=_\beta \lambda z.\ z\ ((\lambda x.\ x)\ z)$ (leftmost outermost redex)

$=_\beta \lambda z.\ z\ z$ call by name

Church-Rosser theorem

↳ Normal form

Normal forms are unique, if they exist.

(For some terms normal forms don't exist.

ex: $(\lambda x.\ xx)(\lambda x.\ xx)$

$=_\beta (\lambda x.\ xx)(\lambda x.\ xx)$

Lambda Calculus (contnd)

↳ Programming based on axioms → program such that it satisfies all the axioms required

ex: programmed numbers zero satisfying add, mul, sub rules & also axioms of zero.

$(\lambda y \cdot x+y)\, x =_\beta x+x$ → acc^n to brute β reduction

but what is overlooked here is the x inside the fx^n body is a free variable & the x substituted in place of y is a bound variable. this makes the initial free variable bound.

free bound.

To avoid this we need to perform α-conversion

α-conversion: ~~Replace~~ Rename free variables before applying the ~~β~~ procedure ~~to~~ to something which cannot be passed as an argument.

$$(\lambda y \cdot x+y)\, x \underset{\alpha}{=\!=} (\lambda y \cdot z+y)\, x =_\beta z+x$$

Let's define Booleans:

↳ what are the tasks a boolean need to perform

if true M N ⇒ M

if false M N ⇒ N

$$\text{true} = \lambda x\, \lambda xy \cdot x = \lambda x \cdot \lambda y \cdot x$$

$$\text{false} = \lambda xy \cdot y = \lambda x \cdot \lambda y \cdot y$$

~~if we just change the order of passin~~

we can return the ~~car~~ correct ~~var~~ term (M / N) by applying the corresponding function.

ex: If true

$(\lambda xy.x)\ M\ N$

$(\lambda y.M)\ N$

M

If false

$(\lambda xy.y)\ M\ N$

$(\lambda xy.y)\ N$

N

OR

OR x y

λxy if x true y

$(\lambda xy$. if x true y) true

$=_\beta (\lambda y$. if true true y)

$=_\beta (\lambda y$. true) y

$=$ true

$(\lambda xy$. if x true y) false.

$=_\beta (\lambda y$. if false true y)

$=_\beta (\lambda y . y)$

Recursion in λ calculus

First of all to recursively call the same function in traditional programming we just call it by name of function. we never defined names of functions in λ calculus. → we

1) Y-combinator ← closed expression (without free variables)

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

$YF =_\beta (\lambda x . F(x\ x))(\lambda x . F(x\ x))$

How? $\circlearrowright =_\beta F((\lambda x . F(x\ x))(\lambda x . F(x\ x)))$

$= F(Y\,F) \ldots F(F(YF)) \ldots$

Ex. $F = \lambda f \cdot \lambda x \cdot$ if $(x = 0)$ then 1 else $x * f(x-$

$YF3 = \beta\,((\lambda x \cdot$ if $(x = 0)$ then 1 else $x * f(x-1))\,3$

$= F(Y\,F)\,3$

$=_\beta 3 * (Y\,F\,2)$

$= 3 * (F(Y\,F)\,2)$

$=_\beta 3 * 2 * (Y\,F1)$ ----

some thing came up as SKI calculus. → programming language based on λ calculus has only 3 variables 'S', 'K', 'I'

$I = \lambda x \cdot x$

$K = \lambda x \cdot \lambda y \cdot x$

$S = \lambda x \cdot \lambda y \cdot \lambda z \cdot x\,z(y\,z)$

$SKSK = KK(SK)$

$= K.$

there is even 'i-combinator' → user only i