"Programming" in λ-calculus

Last time: $\lambda x.x$ Identity (returns its argument)

e.g. $(\lambda x.x)(\lambda y.y) \mapsto \lambda y.y$

$(\lambda x.x)((\lambda y.y)(\lambda z.zz)) \overset{CBN}{\mapsto} (\lambda y.y)(\lambda z.zz)$
$\mapsto \lambda z.zz$

$\underset{CBV}{\mapsto}$
$(\lambda x.x)(\lambda z.zz)$
$\mapsto \lambda z.zz$

Multiple Arguments

Functions in λ-calc can only take one argument

$\lambda x.\lambda y.x$ Function that takes an arg x and returns a <u>function</u> that takes an arg y and returns x.

$\lambda x.\lambda y.y$ returns 2^{nd} arg
$\lambda x.\lambda y.\lambda z.y$ returns 2^{nd} of 3 args

$((\lambda x.\lambda y.x)(\lambda z.z))(\lambda w.w)$ Func. app associates left to make it look more like a 2 arg. func.
$\mapsto ([\lambda z.z/x](\lambda y.x))(\lambda w.w)$
$= (\lambda y.\lambda z.z)(\lambda w.w)$ Constant func. will return $\lambda z.z$ no matter what
$\mapsto [\lambda w.w/y]\lambda z.z$
$= \lambda z.z$

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

Can just pass one arg!
"partial application"
Still waiting to take 2^{nd} arg

Booleans ("Church Booleans" after Alonzo Church)

Need: true, false, if

if true then e_1 else e_2 fi $\equiv e_1$

if false then e_1 else e_2 fi $\equiv e_2$

Only have functions and application

Try: if e then e_1 else e_2 $\overset{\text{define as}}{\triangleq}$ $e\ e_1\ e_2$

What are true and false?

true $\triangleq \lambda t.\,\lambda f.\,t$

false $\triangleq \lambda t.\,\lambda f.\,f$

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

$\equiv \lambda x.x$

Recursion

Got a hint last time: $(\lambda x.xx)(\lambda x.xx) \mapsto (\lambda x.xx)(\lambda x.xx) \mapsto \dots$

"Self-application"

Pairs:

$$(\_,\_) \triangleq \lambda x.\lambda y.\lambda s.\ s\,x\,y \qquad (x,y) \equiv \lambda s.s\,x\,y$$

↑ "selector"

$$\text{fst} \triangleq \lambda p.\ p\,(\lambda f.\lambda s.\ f)$$

$$\text{snd} \triangleq \lambda p.\ p\,(\lambda f.\lambda s.\ s)$$

λ encodings cont'd

Church Numerals

(based somewhat on "A Tutorial Introduction to the Lambda Calculus" by Raul Rojas)

What do we do with a number?

0     $n+1$ "successor"

$\lambda s.\ \lambda z.$ ——

what to do w/ succ   what to do with zero

Apply s n times

$0 \triangleq \lambda s.\ \lambda z. z$ $1 \triangleq \lambda s. \lambda z.\ s\, z$ $2 \triangleq \lambda s.\ z.\ s\,(s\, z)$ ...

Basically like "for $i = 0$ to n ..."

Addition

$$\boxed{\begin{array}{l} succ \triangleq \lambda n.\ \lambda s.\ \lambda z.\ s\,(n\, s\, z) \\ succ\ n \equiv \lambda s.\ \lambda z.\ s\,(n\, s\, z) \longrightarrow n+1 \end{array}}$$

$n + m =$ do s n times, then m more times

$plus \triangleq \lambda m.\ \lambda n.\ n\ succ\ m$

1st arg   2nd arg   apply successor function n times to m

Predecessor

$pred\,(0) \equiv 0$ $pred\,(n+1) \equiv n$

Idea: apply n times a function that computes the pair $(pred\,(n), n)$

$predpair \triangleq \lambda n.\ n\ (\lambda m.\ snd\ m,\ succ\ (snd\ m))$

gets passed the pair from the prev. call

$s(n, n+1) = (n+1, n+2)$

$pair \triangleq \lambda n.\ \cancel{predp}\ fst\ (predpair\ n)$

Recursion, part 2

Let's say we have numbers (yeah, those can be programmed in λ too)

fact $\triangleq \lambda n.$ if $n=0$ then 1 else $n *$ fact $(n-1)$

oops, not defined
no "let rec" in λ-calculus

Let's take another fact function as an argument

fact' $\triangleq \lambda f\, \lambda n.$ if $n=0$ then 1 else $n * f\,(n-1)$
fact $\triangleq$ fact' fact

oops, same problem

Fixed point of a function f = value x such that $fx = x$
Fixed point combinator: A function "fix"
such that fix $f \equiv f$ (fix f)

Let's say we have a "fix"
fact $\triangleq$ fix fact'
$\equiv$ fact' (fix fact') ($\equiv$ fact' fact)
Is this good enough?

fact' (fix fact')
$\equiv \lambda n.$ if $n=0$ then 1 else $n * \underbrace{\text{fact' (fix fact')}}_{\equiv \text{ fix fact}' \ \triangleq \text{ fact}}(n-1)$

Looks good

$Y = \lambda f.(\lambda x. f(x\,x))(\lambda x. f(x\,x))$ – most famous fixed pt. comb.
$Y f \equiv_\beta (\lambda x. f(x\,x))(\lambda x. f(x\,x))$
$\equiv_\beta f((\lambda x. f(x\,x))(\lambda x. f(x\,x)))$
$= f\,(Y f)$ ✓