Lambda Calculus

$M \to x \mid \lambda x.M \mid M\,M$

Semantics usually defined in terms of equivalence

$$\frac{y \notin FV(M)}{\lambda x.M \equiv_\alpha \lambda y.[y/x]M}(\alpha) \qquad \frac{}{(\lambda x.M)\,N \equiv_\beta [N/x]M}(\beta)$$

$$\frac{x \notin FV(M)}{\lambda x.M\,x \equiv_\eta M}(\eta)$$

Reduction

$\lambda x.M \longleftrightarrow \lambda y.[y/x]M$ α-conversion

$(\lambda x.M)N \longrightarrow [N/x]M$ β-reduction

$$\lambda x.M\,x \underset{\eta\text{-reduction}}{\overset{\eta\text{-expansion}}{\rightleftarrows}} M$$

β-normal form - No more β reductions are possible

Not every term has a β-normal form

$\Rightarrow$ Can perform an infinite seq. of β-reductions!

If it does, it's unique, and we only have to do β-reductions

"Computing" w/ λ-calculus = doing β-reductions

$$\frac{M_1 \mapsto M_1'}{M_1\, M_2 \mapsto M_1'\, M_2}\ (1)$$

$$\frac{M_2 \mapsto M_2'}{(\lambda x.M_1)\, M_2 \mapsto (\lambda x.M_1)\, M_2'}\ (2)$$

$$\frac{}{(\lambda x.M_1)(\lambda y.M_2) \mapsto [\lambda y.M_2/x]M_1}\ (3)$$

$\left.\right\}$ Call-by-value (rules 1–3)

$$\frac{}{(\lambda x.M_1)\, M_2 \mapsto [M_2/x]M_1}\ (4)$$ – Call-by-name

Call-by-value

$(\lambda x.\ M)$ (loops forever) $\quad x \notin FV(M)$

$\mapsto \ldots$

May have a β-normal form (M) but we'll never get there!

Call-by-name

$(\lambda x.\ x\,x\,x\,x)$ (takes a long time)

$\mapsto$ (takes a long time) (takes a long time) (takes a long time) (takes...

Theorem [Church-Rosser]

If $M_1 \mapsto^* M_2$ and $M_1 \mapsto M_3$, then there exists M_4 such that $M_2 \mapsto^* M_4$ and $M_3 \mapsto^* M_4$

$$\begin{tikzcd}
 & M_2 \arrow[dr, dashed, "*"] & \\
M_1 \arrow[ur, "*"] \arrow[dr, "*"'] & & M_4 \\
 & M_3 \arrow[ur, dashed, "*"'] &
\end{tikzcd}$$

A lang has the "Church-Rosser property" if diff. evaluation orders lead to the same answer

Substitution

$[M/x]\,x = M$

$[M/x]\,y = y \qquad x \neq y$

$[M/x]\,\lambda y.N = \lambda y.[M/x]N \qquad x \neq y,\ y \notin FV(M)$

$[M/x]\,(M_1\ M_2) = [M/x]M_1\ [M/x]M_2$

"Programming" in λ-calculus

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

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

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

$\mapsto \lambda z.zz$

$\searrow_{CBV}$

$(\lambda x.x)(\lambda z.zz)$

$\mapsto \lambda z.zz$

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 function 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 $\triangleq$ $e\ e_1\ e_2$ ← define as

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: $(\lambda x.xx)\ (\lambda x.xx) \mapsto (\lambda x.xx)(\lambda x.xx) \mapsto \ldots$

"Self-application"

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 $n * \underbrace{fact'\ (fix\ fact')}_{\equiv\, fix\ fact'\ \triangleq\, fact}(n-1)$

Looks good

$Y = \lambda f.(\lambda x. f(x\,x))(\lambda x. f(x\,x))$ - most famous fixed pt. comb.

$Y f \equiv_\beta (\lambda x. f(x\,x))(\lambda x. f(x\,x))$

$\equiv_\beta f((\lambda x. f(x\,x))(\lambda x. f(x\,x)))$

$= f\ (Y f)$ ✓