Λ-Calculus

Pure: e ::= x | e e′ | λx.e
Applied: e ::= x | e e′ | λx.e | c
Combinator: e ::= e e′ | c
Syntactic equality: ≡
Define bound, free variables in the obvious way

Substitution

$$x[L/y] \equiv \begin{cases} L & x \equiv y \\ x & \text{else} \end{cases}$$

$$(\lambda x.M)[L/y] \equiv \begin{cases} (\lambda x.M) & x \equiv y \\ (\lambda x.M[L/y]) & \text{else} \end{cases}$$

$(MN)[L/y] = (M[L/y], N[L/y])$

$M[N/x]$ ok if $BV(M) \cap FV(N) = 0$

Conversion

$(\lambda x.M) \rightarrow_{\alpha} (\lambda y.M[y/x])$

$((\lambda x.M)N) \rightarrow_{\beta} M[N/x]$

$(\lambda x.(Mx)) \rightarrow_{\eta} M$

Watch for variable capture in α,η
Obvious context rules allow reduction inside λ, and to left and right of application

$M \rightarrow N$ if $M \rightarrow_{\beta} N$ or $M \rightarrow_{\eta} N$

$(\Rightarrow) = (\rightarrow)^*$ (reflexive, transitive)

Normal Forms

A term with no reductions is in normal form, some terms have no normal form (e.g. $(\lambda x.xx)(\lambda y.yy)$)
A term is in WHNF if the only reduction it admits is inside λ
A term is in HNF if it looks like $\lambda x_1 ... x_m . y M_1 ... M_k$
NF□HNF, HNF□WHNF

Equality

$(=) = ((\Rightarrow) \cup (\Rightarrow)^{-1})^*$

This is an equivalence relation

$M = N \Rightarrow C[M] = C[N]$

Church-Rosser

If $M = N$ then $\exists L . M \Rightarrow L \wedge N \Rightarrow L$
i.e. confluence
If N in NF then $M \Rightarrow N$
If M,N in NF then $M \equiv N$
If M,N in NF and distinct then there is no way of transforming one to another

Normal Order Reduction

This gives a normal form if one exists: at each step perform the leftmost outermost β reduction first (leave η until last)
Almost call-by-name

Booleans

$true \equiv \lambda xy.x$, $false \equiv \lambda xy.y$

$if \equiv \lambda pxy.pxy$

Operators are easy

Pairs

$pair \equiv \lambda xyf.fxy$

$fst \equiv \lambda p.ptrue$, $snd \equiv \lambda p.pfalse$

Sums

$inl \equiv \lambda x.pairtruex$

$inr \equiv \lambda y.pairfalsey$

$case \equiv \lambda sfg.if(fsts)(f(snds))(g(snds))$

Natural Numbers

$0 = \lambda fx.x$, $1 = \lambda fx.fx$

$add = \lambda mnfx.mf(nfx)$

$mult = \lambda mnfx.m(nf)x$

$\exp = \lambda mnfx.nmfx$

$suc = \lambda nfx.f(nfx)$

$iszero = \lambda n.n(\lambda x.false)true$

$pre = \lambda nfx.snd(n(\lambda y.if(fsty)(pairfalse(sndy))(pairfalse(f(sndy))))(pairtruex))$

$sub \lambda mn.nprem$

Lists

Can encode lists as pairs, with one cons cell being two nested pairs

Recursion

Can't do this directly because such terms would have infinite symbols
Use fixed point combinator Y such that Y F = F (Y F)

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

ISWIM

Sugaring: let, letrec, where, if, pattern matching, n-tuples
Constants: integers, arithmetic, Booleans, relations (e.g. >, not)
δ-reductions reduce constants
Mutable state, so call by value: desugaring for if delays evaluation
Values: constants and functions (expressions in WHNF: call-by-value)

Closures

Package λ-abstraction with its current environment and bound variable (variable to bind into environment when the closure is called)

SECD

Stack, environment, control (list of commands: λ-terms or "app"), dump (empty or machine state, SECD form)
Initial state has (S, E, C, D) = (-, -, M, -) for program M
Do state transition on top of control

SECD Transitions

(S, E, λx.M;C, D) □ (Clo(x, M, E); S), E, C, D)
(S, E, MN;C, D) □ (S, E, N;M;app;C)
(f;a;S, E, app;C, D) □ (f(a);S,E,C,D)
(Clo(x,M,E′);a;S, E, app;C, D) □ (-, x=a;E′, M, (S,E,C,D))
(a, E′, -, (S,E,C,D)) □ (a;S, E, C, D)

Compiled SECD

Pre-compile the λ-term for speed e.g. $[MN] = [N];[M];app$
Could add many commands, e.g. if, tailapp ((Clo(x,C,E′);a, E, tailapp, D) □ (-, x=a;E′, C, D))

SECD

The usual Y fails due to call-by-value.

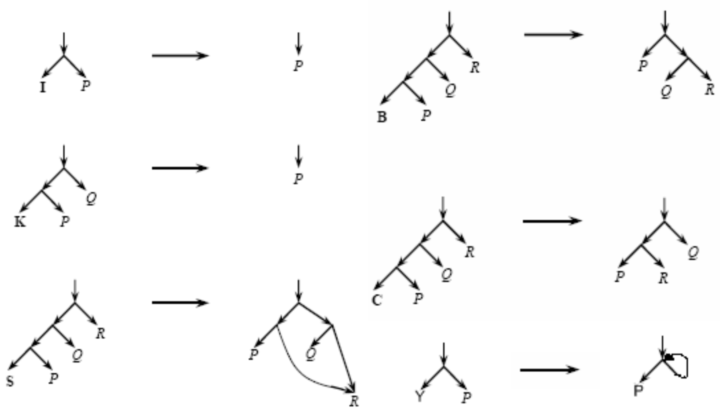
Recursion Can use modified Y combinator $\lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$ but it is slow: make closures with environ. pointing back to closure (does not work for non-funs)

Laziness When a function is called, the argument is stored unevaluated in a thunk, evaluated when a strict built in function is called
Update the environment with the value of the argument after the first evaluation (call-by-need)

Combinators $KPQ \to_\omega P$
$SPQR \to_\omega PR(QR)$
Weak reduction since partially applied combinators do not reduce
$I \equiv SKK$ or define directly
$\lambda^* x.x \equiv I$
$\lambda^* x.P \equiv KP$
$\lambda^* x.PQ \equiv S(\lambda^* x.P)(\lambda^* x.Q)$
$(-)_{CL}$ recursively applies λ^* to a λ-term (innermost terms first) to convert the entire term
$(-)_\lambda$ is trivial (and only causes linear code size increase)
Free variables but not equality is preserved (due to weak reduction)
Adding extensionality (a new rule for proving equality in the combinatory logic) means that equality IS preserved!

Turners Translation $BPQR \to_\omega P(QR)$
$CPQR \to_\omega (PR)Q$
As before, but with:
$\lambda^T x.Px \equiv P$ (x nf in P)
$\lambda^T x.PQ \equiv BP(\lambda^T x.Q)$ (x nf in P)
$\lambda^T x.PQ \equiv C(\lambda^T x.P)Q$ (x nf in Q)
$\lambda^T x.PQ \equiv S(\lambda^T x.P)(\lambda^T x.Q)$ (else)
Only quadratic blowup in size

Graph Reduction Represent λ as graph with sharing, destructively transform
Transform leftmost branch, if function is a strict constant then recursively transform its args



Continuations Functions that never return but exists by calling another function is in continuation passing style:
$[x] = \lambda k.kx$
$[c] = \lambda k.kc$
$[\lambda x.M] = \lambda k.k(\lambda x.[M])$
$[MN] = \lambda k.[M](\lambda m.[N](\lambda n.(mn)k))$
The CPS transform produces an expression with only one reduction possible (CPS encodes control)
Reverse transform by applying I
Slightly less cheat-y transform:
$[x] = \lambda k.kx$
$[c] = \lambda k.kc$
$[\lambda x.M] = \lambda k.k(\lambda(k',x)([M]k')$
$[MN] = \lambda k.[M](\lambda m.[N](\lambda n.m(k,n)))$
Analogous to ret. addr., arg. pair

Side Effects Model using a world threaded through the program
Use each world exactly once
Make things simpler w/ MONADS! return, >>=

Type Inference Use unification: give all variables unknown type variables, constants their declared type then propagate
Need let polymorphism to allow multiple unifications, but has interesting corner cases (y in λx.let y = (x, λz.z) in e’)