Lambda Calculus

[Alonzo Church, 1930s]

The world's simplest Turing-complete programming language

The untyped pure lambda calculus:

```
E      ::= λI.E
         | E E
         | I
```

A purely functional programming language,
stripped to its essence and central power

- Scheme based on it, more so than previous Lisps

Can study important PL issues in context of (extensions of) lambda calculus

- decidability of various properties of programs
- polymorphic type systems

Rewriting rules

α-rule: $\lambda I_1.E \Leftrightarrow \lambda I_2.[I_2/I_1]E$ (if $I_2 \notin FV(E)$)

- free to rename bound identifiers consistently

β-rule: $(\lambda I.E_1)E_2 \Leftrightarrow [E_2/I]E_1$

- application of a function is substitution

η-rule: $\lambda I.(E\ I) \Leftrightarrow E$

- can simplify/create functions

FV(E): set of **free variables** in E

$FV(I) = \{I\}$
$FV(\lambda I.E) = FV(E) - \{I\}$
$FV(E_1\ E_2) = FV(E_1) \cup FV(E_2)$

$FV(E) = \emptyset \Leftrightarrow E$ is **closed**

$[E_2/I]E_1$: **substitution** of E_2 for all *free* occurrences of I in E_1, avoiding **name capture** of free vars in E_2

$[E_2/I]I = E_2$
$[E_2/I]J = J$ (if $J \neq I$)
$[E_2/I]\lambda I.E = \lambda I.E$
$[E_2/I]\lambda J.E = \lambda K.[E_2/I][K/J]E$
(if $J \neq I$ and $K \neq I$ and $K \notin FV(E_2)$)
$[E_2/I](E_1\ E_3) = ([E_2/I]E_1)\ ([E_2/I]E_3)$

Evaluation of lambda expressions

Evaluate a lambda calculus program by applying **β-reduction**

- β-reduction: β-rule applied only in $\Rightarrow$ direction
- occurrences of λ expression applied to argument: β-redex

One β-reduction step: $E_0 \Rightarrow E_1$

Finite-length sequence of $n \geq 0$ β-reduction steps:

$E_0 \Rightarrow^* E_n$

An expression **is** in **normal form** if it has no β-redexes

- an "answer" to running a lambda-calculus program

An expression E **has** a normal form iff
there exists E′ in normal form s.t. $E \Rightarrow^* E'$

Properties of reduction

Not all expressions have normal forms:

```
((λI.I I) (λI.I I))
((λI.I (I I)) (λI.I (I I)))
λF.((λI. F (I I)) (λI. F (I I)))
```

- can program infinite loops in lambda calculus

Confluence (Church-Rosser property):

if $E \Rightarrow^* E_1$ and $E \Rightarrow^* E_1$, then there exists some E_3 s.t.
$E_1 \Rightarrow^* E_3$ and $E_2 \Rightarrow^* E_3$
(modulo application of α-rule to E_3 (α-renaming))

- all terminating evaluation orders are equivalent!

Normalization

- all terminating evaluations of a lambda program yield the same unique normal form (answer) (modulo α-renaming)

Some reduction strategies

Normal-order evaluation:
always pick leftmost-outermost redex

- corresponds to lazy evaluation (but without caching results)

Applicative-order evaluation:
always pick rightmost-innermost redex

- corresponds to eager evaluation

Standardization:
if an expression has a normal form,
normal-order evaluation will find it
(applicative-order evaluation may not)

Encoding numbers & arithmetic in lambda calculus

Numbers:

0 ≡ λS.λZ.Z

1 ≡ λS.λZ.S Z

2 ≡ λS.λZ.S (S Z)

...

n ≡ λS.λZ.S (S ...(S Z)...) (*n* S's)

Some arithmetic functions:

succ ≡ λN.λS.λZ.S (N S Z)

add ≡ λM.λN.M *succ* N

[Syntactic precedence & associativity rules:
application is tighter binding than .: λI.I I = λI.(I I)
application is left associative: I J K = (I J) K]

[Note that these definitions are just short-hands, whose meaning is just inline expansion. No recursive definitions!]

Encoding booleans in lambda calculus

Truth values:

true ≡ λT.λF.T

false ≡ λT.λF.F

A function:

if ≡ λB.λT.λF.B T F

Encoding pairs in lambda calculus

Constructor:

pair ≡ λF.λS.λP.P F S

Projections:

first ≡ λP.P (λF.λS.F)

second ≡ λP.P (λF.λS.S)

Same strategy for arbitrary tuples, lists, etc.