

λ Calculus

Church's λ Calculus: Brief History

- One of a number of approaches to a mathematical challenge at the time (1930): Constructibility
 - (What does it mean for an object, e.g. a natural number, to be constructible?)
 - aka "effective computability", "computability"
- Work in parallel included:
 - Turing's work on Turing machines
 - Gödel's work on general recursive functions

History (continued)

- In late 30's, Church, Kleene and Turing showed equivalence of their respective notions.
- Led to Church's thesis: notion of a computable function should be identified with the notion of a general recursive function.

Church's Lambda Calculus:

- Formally specifies the difference between functions and forms.
- Form: specifies operations that are to be applied to the parameters of the form (with corresponding free variables and constants).
- e.g. of a form: $\mathbf{a X^2 + X + Y}$
 - X, Y: parameters
 - a: free variable (not parameter to form)
 - 2: constant

Forms

- Note: if actual arguments are applied to form, there is no way to specify their bindings.

$$P, Q \rightarrow a X^2 + X + Y$$

$$\begin{array}{l} P \rightarrow X \quad \& \quad Q \rightarrow Y \quad ? \\ P \rightarrow Y \quad \& \quad Q \rightarrow X \quad ? \end{array}$$

λ Functions

- Lambda function resolves ambiguity and defines difference between functions and forms:

$$(\lambda x. (\lambda y. a x^2 + x + y))$$

- function with two parameters, x & y, where first actual is to be bound to x and second to y.
- Curried interpretation: function of x which yields a function of y which...
- Two interesting characteristics of lambda calculus:
 - 1) Church defined argument substitution assuming *static scope*. (and actuals bound by λ were to be unique throughout form)

λ Functions

2) Form can only contain applications of other functions, not their definitions.

- instances of other formal parameters bound to other lambdas cannot exist in a given lambda function.
- functions cannot be used as arguments or function values because a function would appear where a form or object is expected.

McCarthy's LISP

- McCarthy's LISP (1958-1960)
 - First language to be based on Lambda Calculus
 - Two major differences:
- 1) LISP used dynamic scope
so:
(Define poly (λ (X Y) (+ (+ (* a (* X X)) X) Y)))
(Define p1 (λ (a) (poly 2 3)))
(Define p2 (λ (a) (poly 4 5)))

(p1 10)
(p2 20)

"a" has different bindings in poly when called by p1, p2.
- so LISP maintained "a-list"

(continue) McCarthy's LISP

- 2) LISP (many versions before Scheme, ML) allowed functions as arguments.
 - quoted lambda expressions were passed as "funargs." (pass-by-name definitions)
 - each time funarg was referenced it caused evaluation of actual parameter's lambda definition *in its defining scope*.
 - Note: Scheme, ML, Haskell allow functions as arguments
 - they evaluate to themselves.
- McCarthy has suggested that the reason LISP used dynamic scope was that he did not fully understand the Lambda Calculus of Church during the development of LISP...

λ Expressions

$\langle \text{exp} \rangle ::= \langle \text{constant} \rangle$	built-in constant
$\langle \text{variable} \rangle$	variable names
$\langle \text{exp} \rangle \langle \text{exp} \rangle$	applications
$\lambda \langle \text{variable} \rangle . \langle \text{exp} \rangle$	lambda abstractions

λ Abstractions

- Purpose is to denote new functions:

$(\lambda x . + x 1)$

$(\lambda$ x $.$ $+$ x $1)$
↑ ↑ ↑ ↑ ↑ ↑
That function of x which adds x to 1

Free and Bound Variables

$(\lambda x . + x y)$

-- x is bound (by the λ) but y is free

$\lambda x . + ((\lambda y . + y z) 7) x$

-- x & y are bound; z is free

$+ x ((\lambda x . + x 1) 4)$

-- first x is free; second is bound...

- Occurrence of variable is bound if an enclosing λ expression binds it, and it is free otherwise.

Conversions, BAH!

- Beta (β): (abstraction and reduction)
 - reduction: applying λ abstraction to an argument, making new instance of abstraction body, and substituting argument for free occurrence of formal
 - abstraction: going the opposite way
- Alpha (α): changing names
 - consistent formal parameter name change in λ expression.
- Eta (η): elimination of redundant λ abstractions

Substitution

$E[M/x]$

-- expression E with all free occurrences of x replaced by M

$$x [M/x] = M$$

$$c [M/x] = c, \text{ where } c \text{ is variable or constant other than } x$$

$$(E F)[M/x] = E [M/x] F[M/x]$$

$$(\lambda x.E)[M/x] = \lambda x.E \quad (\text{because no free occurrences of } x)$$

$$(\lambda y.E)[M/x] \text{ where } y \text{ is not } x$$

$$= \lambda y.E[M/x] \text{ if } x \text{ does not occur free in } E \\ \text{or } y \text{ does not occur free in } M$$

$$= \lambda z.(E[z/y]) [M/x] \text{ otherwise}$$

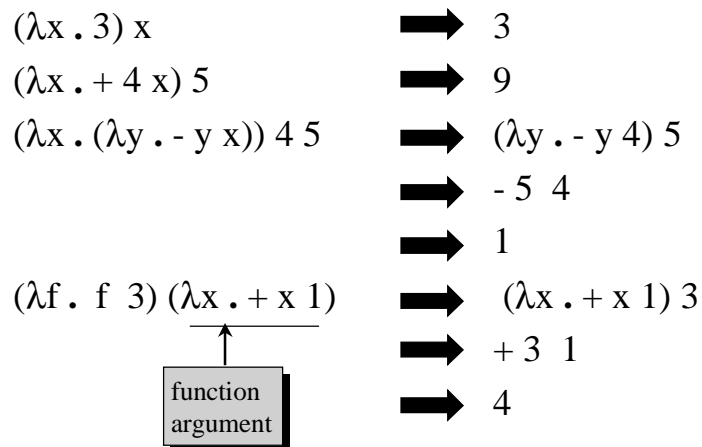
$$\text{where } z \text{ is new variable not free in } E \text{ or } M$$

Conversions Summary

- β : $(\lambda x . E) M \xleftrightarrow{\beta} E[M/x]$
- α : if y is not free in E then
 $(\lambda x . E) \xleftrightarrow{\alpha} (\lambda y . E[y/x])$
- η : if x is not free in E
 and E denotes a function then
 $(\lambda x . E x) \xleftrightarrow{\eta} E$
- when applied left to right ($\xrightarrow{\quad}$), β and η rules are called reductions

Beta Reduction

- Reducing an expression:



Alpha Conversion

- Ought to be equivalent...

$(\lambda x . + 1 x)$ & $(\lambda y . + 1 y)$

and, indeed...

$(\lambda x . + 1 x)$ $\longleftrightarrow_{\alpha}$ $(\lambda y . + 1 y)$

...as long as newly introduced name does not occur freely in body of original lambda expression.

Eta Conversion

- Ought to be equivalent...

$(\lambda x . + 1 x)$ & $(+ 1)$

and, indeed...

$(\lambda x . + 1 x)$ $\longleftrightarrow_{\eta}$ $(+ 1)$

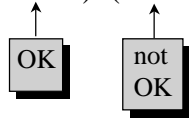
- In general...

$(\lambda x . F x)$ $\longleftrightarrow_{\eta}$ F

...provided x is not free in F and F is a function

Normal Form

- **Def:** When an expression contains no reducible expressions (redexes).
- There may be more than one route to normal form for an expression E
 - e.g. $(+ (* 3 4) (* 7 8))$
- Not every expression has a normal form
 - e.g. $(D D)$ where D is $(\lambda x . x x)$
 - produces $(D D)$
- Some reductions may reach normal form while others do not
 - e.g. $(\lambda x . 3) (D D)$



Normal Form (CRT)

- (CRT-I):
 - If $E_1 \iff E_2$ then there exists an E such that $E_1 \implies E$ and $E_2 \implies E$
 - in words: two expressions that can be converted to each other share a common normal form.
- Corollary: No expression can be converted to two distinct normal forms
- (CRT-II):
 - If $E_1 \implies E_2$ and E_2 is in normal form, then there exists a normal order reduction sequence from E_1 to E_2 .
 - Normal order reduction: reduce leftmost, outermost redex first

λ Calculus Utility

- Supports expression of recursion!
 $\mathbf{Y H = H (Y H)}$
 - Y: a fixed point combinator: takes a function H and produces a fixed point of H
 - See Peyton-Jones, section 2.4.1
- Supports typed, untyped and polymorphic systems
- Underlies denotational semantics...