Lambda Calculus

Variables and Functions

Lambda Calculus

- Mathematical system for functions
  - Computation with functions
  - Captures essence of variable binding
    - Function parameters and substitution
  - Can be extended with types, expressions, memory stores and side-effects
- Introduced by Church in 1930s
  - Notation for function expressions
  - Proof system for equality of expressions
  - Calculation rules for function application (invocation)

Pure Lambda Calculus

- Abstract syntax**:** **M ::= x | λx.M | M M**
  - x represents variable names
  - **λx.M** is equivalent to (lambda (x) M) in Lisp/Scheme
  - **M M** is equivalent to (M M) in Lisp/Scheme
  - Each expression is called a lambda term or a lambda expression
- Concrete syntax: add parentheses to resolve ambiguity
  - (**M M**) has higher precedence than **λx.M**;

    i.e. **λx.M N** => **λx. (M N)**
  - **M M** is left associative; i.e. x y z => (x y) z
- Compare: concrete syntax in Lisp/Scheme
  - **M ::= x | (lambda (x) M) | (M M)**

The Applied Lambda Calculus

- Can pure lambda calculi express all computation?
  - Yes, it is Turing complete. Other values/operations can be represented as function abstractions.
    - For example, boolean values can be expressed as

      True = λ t. (λ f. t)

      False = λ t. (λ f. f)
  - But we are not going to be extreme.
- The applied lambda calculus

  M ::= e | x | λ x.M | M M
  - e represents all regular arithmetic expressions
- Examples of applied lambda calculus
  - Expressions: x+y, x+2*y+z
  - Function abstraction/definition: λ x.(x+y), λz.(x+2*y+z)
  - Function application (invocation): (λ x.(x+y)) 3

Lambda Calculus In Real Languages

- Lisp
  - Many different dialects
    - Lisp 1.5, Maclisp, …, Scheme, ...CommonLisp,…
    - This class uses Scheme
  - Function abstraction (allow multiple parameters)
    - λ x. M => (lambda (x) M)
    - λx. λ y. λ z. M => (lambda (x y z) M)
  - Function application
    - M1 M2 => (M1 M2)
    - (M1 M2) M3 => (M1 M2 M3)
- C (each function must have a name)
  - λ x. λ y. λ z. M => int f(int x,int y,int z) { return M; }
  - (M1 M2) M3 => M1(M2, M3)

Example Lambda Terms

- Nested function abstractions (definitions)

  λ s. λ z. z

  λ s. λ z. s (s z)

  λ s. λ z. s (s (s z)))

- Nested function applications (invocations)

  x y z

  (λ s. λ z. z) y z

  (λ s. λ z. s (s z)) ((λ s. λ z. z) y z)

Semantics of Lambda Calculus

- The lambda calculus language
  - Pure lambda calculus supports only a single type: function
    - Applied lambda calculus supports additional types of values such as int, char, float etc.
  - Evaluation of lambda calculus involves a single operation: function application (invocation)
  - Provide theoretical foundation for reasoning about semantics of functions in Programming Languages
    - Functions are used both as parameters and return values
    - Support higher-order functions; functions are first-class objects.
- Semantic definitions
  - How to bind variables to values (substitute parameters with values)?
  - How do we know whether two lambda terms are equal? (evaluation)

Evaluating Lambda Calculus

- What happens in evaluation

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

  $(\lambda\ f.\ \lambda\ x.\ f\ (f\ x))\ g = \lambda\ x.\ g\ (g\ x)$

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

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

- Lambda term evaluation => substitute variables (parameters) with values
  - Each variable is a name (or memory store) that can be given different values
  - When variables are used in expressions, need find the binding location/declaration and get the value

Variable Binding

- Bound and Free variables
  - Each λ x.M declares a new local variable x
    - x is bound (local) in λ x.M
    - The binding scope of x is M => the occurrences of x in M refers to the λ x declaration
  - Each variable x in a expression M is free (global) if
    - there is no λx in the expression M, or x appears outside all λx declarations in M
    - The binding scope of x is somewhere outside of M
- Example: λ x. λ y. (z1*x+z2 *y)
  - Bound variables: x, y; free variables: z1, z2
  - Binding scopes

    λ x => λ y. (z1*x+z2 *y)

    λ y => (z1*x+z2 *y)
- Do variable names matter?

  λ x. (x+y) = λ z. (z+y)

  Bound (local) variables: no; Free (global) variables: yes
  - Example: y is both free and bound in λ x. ((λ y. y+2) x) + y

Equality of Lambda Terms

- α–axiom

  $\lambda x.\ M = \lambda y.\ [y/x]M$

  - [y/x]M: substitutes y for free occurrences of x in M
  - y cannot already appear in M
  - Example
    - $\lambda\, x.\ (x + y) = \lambda\, z.\ (z + y)$
    - But $\lambda\, x.\ (x + y) \neq \lambda\, y.\ (y + y)$

- β-axiom

  $(\lambda\, x.\ M)\ N = [N/x]\ M$

  - [N/x]M: substitutes N for free occurrences of x in M
  - Free variables in N cannot be bound in M
  - Example
    - $(\lambda\, x.\ \lambda\, y.\ (x + y))\ z1 = \lambda\, y.\ (z1+y)$
    - But $(\lambda\, x.\ \lambda\, y.\ (x + y))\ y \neq \lambda\, y.\ (y + y)$

Evaluation of Lambda-terms

- β-reduction

  $(\lambda\ x.\ t1)\ t2 \quad => \quad [t2/x]t1$

  - where [t2/x]t1 involves renaming as needed
  - Rename bound variables in t1 if they appear free in t2
    - α-conversion: $\lambda\ x.\ M => \lambda\ y.\ [y/x]M$ (y is not free in M)
  - Replaces all free occurrences of x in t1 with t2
- Reduction
  - Repeatedly apply β-reduction to each subexpression
  - Each reducible expression is called a redex
  - The order of applying β-reductions does not matter

Example: Variable Substitution

- $\underbrace{(\lambda\ f.\ \lambda\ x.\ f\ (f\ x))}_{\text{apply twice}}\ \ (\lambda\ \underbrace{y.\ y+x}_{\text{add x to argument}})$

- Substitute variables "blindly"

  $\lambda\ x.\ [(\lambda\ y.\ y+x)\ ((\lambda\ y.\ y+x)\ x)] \quad => \lambda\ x.\ x+x+x$

- Rename bound variables

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

  $=> \lambda\ z.\ [(\lambda\ y.\ y+x)\ ((\lambda\ y.\ y+x)\ z))]$

  $=> \lambda\ z.\ z+x+x$

Easy rule: always rename variables to be distinct

Examples
Reduce Lambda Terms

- (λ x. (x+y)) 3
- (λ f. λ x. f (f x)) (λ y. y+1)
- λ x. (λ y. y x) (λ z. x z)
- (λ x. (λ y. y x) (λ z. x z)) (λ y. y y)

Solutions
Reduce Lambda Terms

- (λ x. (x+y)) 3
  => 3 + y
- (λ f. λ x. f (f x)) (λ y. y+1)
  => λ x. (λ y. y+1) ((λ y. y+1) x)
  => λ x. (λ y. y+1) (x+1)
  => λ x. (x+1)+1
- λ x. (λ y. y x) (λ z. x z)
  => λ x. (λ z. x z) x
  => λ x. x x
- (λ x. (λ y. y x) (λ z. x z)) (λ y. y y)
  => (λ x. x x) (λ y. y y)
  => (λ y. y y) (λ y. y y)
  => (λ y. y y) (λ y. y y)

Confluence of Reduction

- Reduction
  - Repeatedly apply β-reduction to each subexpression
  - Each reducible expression is called a redex
- Normal form
  - A lambda expression that cannot be further reduced
  - The order of applying β-reductions does not matter
- Confluence
  - If a lambda expression can be reduced to a normal form, the final result is uniquely determined
  - Ordering of applying reductions does not matter

Termination of Reduction

- Can all lambda terms be reduced to normal form?
  - No. Some lambda terms do not have a normal form (i.e., their reduction does not terminate)
  - Example non-terminating reductions
    $(\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$
    $=> (\lambda\ y.\ y\ y)\ (\lambda\ x.\ x\ x) => (\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)\ \ldots$
- Combinators
  - Pure lambda terms without free variables
- Fixed-point combinator
  - A combinator Y such that given a function f, Y f => f (Y f)
  - Example: $Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
    - **$Yf = (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ f$**
      $=> (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
      $=> f\ ((\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))$
      $=> f\ (Yf)$

Recursion and Fixed Points

- Recursive functions
  - The body of a function invokes the function
    - Factorial: f(n) = if n=0 then 1 else n*f(n-1)
- Is it possible to write recursive functions in Lambda Calculus?
  - Yes, using fixed-point combinator
- More advanced topics (not required)