Lambda Calculus

- A formalism for describing the semantics of operations in functional programming languages
- Variables (free or bound), function definition (or abstraction), function application, currying
- Substitution rules

  β reduction, α reduction, η-reduction

  Normal form

Lambda Calculus

- Church-Rosser theorem
- Evaluation order
  - Call-by-name
  - Call-by-value
  - Call-by-need (lazy)

Lambda Calculus

- Universal *theory of functions*
- λ-calculus (Church), recursive function theory (Kleene), Turing machines (Turing) all were formal systems to describe computation, developed at the same time in the 1930's
  - Shown formally equivalent to each other
  - Results from one, apply to others

Lambda Calculus

- *Conjecture:* class of programs written in λ-calculus is equivalent to those which can be simulated on Turing machines.
- All partial recursive functions can be defined in λ-calculus.
- Pure λ-calculus involves functions with no side effects and no types.

Lambda Calculus

- Function: a map from a domain to a range
- Terms:
  - variable (X)
  - function abstraction or definition (λx.M)
  - function application (M N)

Function Definition (Abstraction)

- F(y) = 2 + y -- mathematics
- F ≡ λ y. *2+y* -- λ calculus
  - bound variable or argument
  - *function body*
- λ x.x (identity function)
- λ y. 2 (constant function whose value is 2)

Function Application

- *Process:* take the argument and substitute it everywhere in the function body for the parameter

  (F 3) is 2 + 3 = 5; ((λ x.x) λ y.2) is λ y.2;

  ((λ z. z+5) 3) is 3+5 = 8

- Functions are *first class citizens*
  1. Can be returned as a value
  2. Can be passed as an argument
  3. Can be put into a data structure as a value
  4. Can be the value of an expression

Relation to C Function Pointers

- Can simulate #1-4 with C function pointers, but this abstraction is closer to the machine than a function abstraction.
- Functions as values are defined more cleanly in Lisp or SML.
- No analogue in C for an unnamed function, (Lisp lambda expression)

Function Application

- Left associative operator- (f g h) is ((f g) h)
- λ x.M x is same as λ x.(M x)
- Function application has highest precedence
- *Currying* (cf. *Haskell Curry*)

  Area of triangle is λ b. λ h.(b*h)/2

  (Area 3) is a function, λ h.(3*h)/2, that describes the area of a family of triangles all with base 3

  ((Area 3) 7) = 3 * 7 / 2 = 10.5

  in *curried form*, a function takes its arguments one-by-one

Type Signatures

- Area: can write function in two ways
  - un-curried: $\alpha * \beta \rightarrow \gamma$, given b, h as a pair of values, the function returns area
  - curried: $\alpha \rightarrow (\beta \rightarrow \gamma)$, given b, returns a function to calculate area when given h(height)

Free and Bound Variables

- ***Bound*** **variable: x is *bound* when there is a corresponding λx in front of the λ expression:**

  ((λ y. y)   *y*)   is *y*
  bound   free

- **Free variable: x is not bound (analogous to a variable inherited from an encompassing imperative scope)**

binding occurrence of z **bound occurrence of z**

(λ y. z)   (λ z. z)

free occurrence of z

Free and Bound Variables

Sethi, p550

- **x is free in x, free(x) = x**
- **x is free (*bound*) in Y Z if x is free (*bound*) in Y or in Z, free(YZ)= free(Y) $\cup$ free(Z)**
- **x $\notin$ V, then x free *(bound)* in λV.Y iff it occurs free *(bound)* in Y. All occurrences of elements of V are *bound* in λ V.Y,**

  free(λx.M) = free(M) - {x}

- **x free *(bound)* in (Y), if x is free *(bound)* in Y**

Substitution

- *Idea*: function application is seen as a kind of substitution which simplifies a term
  - ((λx.M) N) as *substituting N for x in M* ; written as {N | x} M
- Rules - Sethi, p551
  1. If free variables of N have no bound occurrences in M, then {N | x} M formed by replacing all free occurrences of x in M by N.

Substitution

plus ≡ λa.λb. a+b

then (plus 2) ≡ λb. 2+b but if we evaluate (plus b 3) we get into trouble!

(plus b 3) = (λa.λb. a+b b 3)
= (λb. b+b 3)
= 3 + 3 = 6

(plus b 3) = (λa.λc. a+c b 3)
= (λc. b+c 3)
= b + 3, what we expected!

problem:
b is a bound variable; need to rename before substitute.

Substitution

2. If variable y free in N and bound in M, replace binding and bound occurrences of y by a new variable named z. Repeat until case 1. applies.

- Examples

$\{u \mid x\}\ x = u$ $\{u \mid x\}\ (x\ \ u) = (u\ \ u)$

$\{\lambda x.x \mid x\}\ x = \lambda x.x$ $\{u \mid x\}\ y = y$

$\{u \mid x\}\ \lambda x.x = \lambda x.x$

$\{u \mid x\}\ (\lambda u.x) = \{u \mid x\}\ (\lambda z.x) = \lambda z.u$

$\{u \mid x\}\ (\lambda u.u) = \{u \mid x\}\ (\lambda z.z) = \lambda z.z$

Examples of need for change of variables.

Reductions

- β–reduction $(\lambda x.M)\ N = \{N \mid x\}\ M$ with above rules
- α–reduction $(\lambda x.M) = \lambda z.\{z \mid x\}\ M$, if z not free in M (allows change of bound variable names)
- η-reduction $(\lambda\ x.(M\ \ x)) = M$, if x not free in M (allows stripping off of layers of indirection in function application)
- See Sethi, Figure 14.1, p 553 for rules about β–equality of terms

Example

Evaluate (λxyz . xz (yz)) (λx. x) (λy. y)

(λxyz . (xz (yz))) (λx. x) (λy. y), 2 α–reds + fully parenthesize

= [{ (λabz .(a z (b z))) (λx .x)} (λy .y)]

= [{ (λbz. ((λx.x) z (b z))) } (λy .y)], {λx.x | a}

= [{ λbz. (((λx.x) z) (b z))} (λy .y)], fully parenthesize

= [{λbz. (z (b z))} (λy .y)], {z | x}

= [{ λz. (z ((λy .y) z))}], {λy.y | b}

= { (λz. z z)}, {z | y}

- **Note: we picked the order of β–reductions here**

Substitution Rules cf Sethi p 555, GHH p 49

M	**{N | x} M**
x	N
y	M

if M a variable, then if M ≠ x get M, else get N (3.1 GHH)

PQ	{N | x} P {N | x} Q

result of substitution applied to function application is to apply that substitution to the function and its argument and then perform the resulting application(3.2 GHH)

Substitution Rules cf Sethi p 555, GHH p 49

	M	**{N	x} M**
3.3a)	**λx .P**	**λx .P**

never substitute for a bound variable within its scope

3.3b)	**λy .P**	**λy .P**

if there are no free occurrences of x in P

3.3c)	**λy .P**	**λy .{N	x} P**

when there are no free occurrences of y in N

3.3d)	**λy .P**	**λz .{N	x} { z	y} P**

when there is a free occurrence of y in N and z is not free in P or N, substitute z for y in P and continue.

Substitution Rules

- **All these checks are aimed at ensuring that we don't link variable occurrences that are independent!**
- **Our example ((λ a.λ b.a+b) b), would use 3.3d to change variables before doing the substitution**
- ***Normal form* of a term - a form which can allow no further β or η reductions**
  - **No remaining ((λx.M) N), called a *redex* or term which can be reduced**

Example GHH, p50

{y | x} λ y. x y *use 3.3d to change bound var*

λ z. {y | x} ({z | y} (x y)) *apply 3.2 for fcn appln*

λ z. {y | x} ({z | y} (x) {z | y} (y)) *apply 3.1 twice*

λ z. {y | x} (x z) *apply 3.2*

λ z. ({y | x} (x) {y | x} (z)) *apply 3.1 twice*

λ z. y z *final result;*

compare this to what we started with!

Church Rosser Property

- Fundamental result of λ-calculus:
  - Result of a computation is *independent* of the order in which β-reductions are applied
  - Leads to referential transparency in functional PL's
  - Another interpretation is that most terms in the λ -calculus have a *normal form*, a form that cannot be reduced any simpler; Church Rosser says <u>if a normal form exists</u>, <u>then all reduction sequences lead to it</u>

Normal Form

- **Does every λ-expression have a normal form? NO, because there are terms which cannot be simplified, yet they contain redices**
  - **(λx.x x) (λx.x x) = (λy. y y) (λx.x x) , α-reduction**
    = (λx.x x) (λx.x x), β-reduction

  this term has no normal form
  - **(λx.x x x) (λx.x x x) = (λy. y y y) (λx.x x x) , α-red**
    = (λx.x x x) (λx.x x x) (λx.x x x),β-red

  this term grows as we apply β-reductions!

Normal Form

- **If add6 ≡ λx. x+6, twice ≡ λfλx. f (f x), what is value of (twice add6)?**

(twice add6) = (λf.λz.f (f z)) (λx.x+6)
= λz. ((λx.x+6) ((λx.x+6) z))
= λz. ((λx.x+6) (z+6))
= λz. (z + 12), ***normal form***

- **normal form of {λx. ((λz.z x) (λx.x))} y?**

free

{λx. ((λz.z x) (λx.x))} y = {λx. ((λx.x) x)} y

bound

= {λx. x} y
= y

Equality of Terms

- **Reduce each term to its normal form and compare**
- **But whether or not a term has a normal form is *undecidable* (related to halting problem for Turing machies)**
- **Same term may have terminating and nonterminating β-reduction sequences; if at least one terminates, use its result as the normal form for that term**

Church Rosser Property

- **(GHH)Theorem 1: If a λ-expression reduces to a normal form, it is unique**
- **(GHH)Theorem 2: If we always reduce leftmost redex first, the reduction sequence will terminate in a normal form, if it exists.**
  - **….A….B… both A and B are redices. if first λ in A is to the left of first λ in B, then A is *to the left of* B**
  - **A redex to left of all other redices in a λ-expression is *leftmost***

Church Rosser Property

- (Sethi) Theorem: For λ-expressions M,P,Q, let ⇒ stand for a sequence of α and β-reductions. if M ⇒P and M ⇒Q then ∃ a term R such that P ⇒R and Q⇒ R
  - Says all reduction sequences progress towards the same end result if they all terminate

$$\begin{tikzcd} & M \arrow[dl] \arrow[dr] & \\ P \arrow[dr] & & Q \arrow[dl] \\ & R & \end{tikzcd}$$

"Proof of CR by Example"

(λx.λy.x-y) ((λz.z) 2) ((λr.r+2) 3) $\boxed{\sim f\ \ g\ \ h}$

first eval

second eval

substituting for x first:

= (λy.((λz.z) 2) - y) ((λr.r+2) 3)

= (λy.2-y) ((λr.r+2) 3)

= 2 - ((λr.r+2) 3)

= 2 - 5

= -3

"Proof of CR by Example"

(λx.λy.x-y) ((λz.z) 2) ((λr.r+2) 3)

substitute for y first:

= (λx.x- ((λr.r+2) 3)) ((λz.z) 2)

= (λx.x - 5) ((λz.z) 2)

= (((λz.z) 2) - 5)

= (2 - 5)

= -3, the same result!

substituting for x first:

= (λy.((λz.z) 2) - y) ((λr.r+2) 3)

= (λy.2-y) ((λr.r+2) 3)

= 2 - ((λr.r+2) 3)

= 2 - 5

= -3

Call by Name

- **Can result in some parameter being evaluated several times - inefficient**
- **Evaluates arguments only when they are needed (Algol60 thunks)**
- **Abandoned in modern PLs because of inefficiency**
- **However, guaranteed to reach a normal form if it exists**

Call by Value

- Efficient
- Potentially does a calculation that may not be used (if fcn is not *strict* in that parameter)
- Can lead to non-terminating computation
  - Used in C, Pascal, C++, functional languages
- Often obtains a normal form in real programs

Call by Need

- *Lazy evaluation* - once we evaluate an argument, then memoize its value to use again, if needed
- Inbetween two other methods: value and name
- Accomplished by embedding a pointer to a value instead of the argument itself in the expression. Then, when value is first calculated, it is saved so it will be available for other uses

Call by Need

- Allows use of unbounded streams of input as well
  - What if we need a function to generate list(n), a list of length n?
  - hd (tl (list(n))) needs only the first 2 elements to be generated; system will only evaluate this many elements which prefix the list.

Reduction Order

- Distinguishing order of applying β-reductions only matters when some reduction order leads to a non-terminating computation
- Sethi, p560:
  - Leftmost outermost redex first is call by name (normal order)
  - Leftmost, innermost redex first is call by value

Where inner and outer refer to nesting of terms

(λ yz. (λx.x) z (y z)) (λ x.x)

Reduction Order

- **Start with fully parenthesized expression:**
  - **(λv. e) (i) - always reduce e first**
  - **(c b) (ii) - if c is not of form (i), then reduce c until it is of that form. Then, we have a choice as to how to proceed:**
    - **call by name: reduce (c b) without further reducing inside c or b.**
    - **call by value: reduce any redices in c, then those in b, and then reduce (c b).**

Example 1

(Sethi, p560) {[λy.λz. ((λx.x) z) (y z))] (λx.x)} = (c b)

***call by value:* reduce c. [λy.λz.(z (y z))] (λx.x) = (c' b) where b already reduced. reduce (c' b) yielding**

λz.(z ((λx.x) z)) = λz.(z (c'' b'')). reduce (c'' b'') which yields

λz.(z z), the final term.

***call by name:* c is an abstraction (form i). so instantiate b directly into c yielding λz.(((λx.x) z) ((λx.x) z)) = λ z. (c* b*)**

now reduce c* so we get an abstraction (form i.), yielding z.   then can perform final reduction of λz.(z ((λx.x) z)), yielding

λz. z z, the final term, same as above.

Example 2

(((λx.λy.x) z) ((λr.r r) (λs. s s))) = (c b).

call by value: **reduce c to yield ((λy.z) ((λr.r r) (λs. s s))) which is ((λy.z) (c' b')). reduce (c' b') yielding**

((λy.z) ((λs.s s) (λs. s s))). we end up with a similar term b''. repeating this reduction will result in a non-terminating computation

call by name: **reduce c to yield ((λy.z) ((λr.r r) (λs. s s))). now substitute b into the reduced c, yielding z, because there is no bound y in λy.z. z is the normal form for the above term, by definition.**

Example 3

c'' b''

{(λz. (λx.x+6) ((λx.x+6) z)) 1} = { c b }

(c' , b')

call by value: **reduce redices in c = (c' b') where b' = (c'' b'').**

(c'' b'') evaluates to b' = z+6, yielding {(λz. (λx.x+6) (z+6)) 1}. now evaluating (c' b') yields {(λz. (z+6)+6) 1} = {(λz. z+12) 1} now evaluating {c b} yields 1 + 12 = 13.

call by name: **c is of correct form, an abstraction (form i.). so substitute b into c yielding ((λx.x+6) ((λx.x+6) 1)) = (c* b*). substitute b* into c* yielding ((λx.x+6) 1) + 6 = (c^ b^) + 6. substitute b^ into c^ yielding (1 + 6) + 6 = 7+6 = 13.**