Lambda Calculus

- Formalism to describe semantics of operations in functional PLs
  - Variables are free or bound
    - Function definition vs function abstraction
    - Substitution rules for evaluating functions
    - Normal form
  - Equivalent in descriptive power to Turing machines
- Substitution rules
    - β reduction, α reduction, η-reduction

Lambda Calculus

- A theory of functions
  - Equivalent in descriptive power to Turing machines
- Syntax

  Exp ::= Var | λ Var.Exp | (Exp1 Exp2)

  λx.(x z) corresponds to (lambda (x) (x z)) in Scheme
- How to express a function call?
  - Use notion of substitution
  - Substitute E for y in M, {E/y} M

  ((λx.λy.(+ x y)) 2 3)-->((λy.(+ 2 y)) 3)--> (+ 2 3)

Evaluating Functions

- Not so easy to do
  - if foo is (λx. λx.x) then (foo 2) is λx.x not 2 (x is a parameter of the function λx.x that is an argument and unrelated to x.
  - if area is (λx.λy. (* x y)) then (area 3 y) should be (* 3 y) not (* y y), but 2nd result happens by blindly following simple substitution
  - free/bound variables distinguish these cases

    λx.(x + y) y is *free*, x is *bound*

Rules of Lambda Calculus

- **α rule:** choice of parameter names doesn't matter, λx. M = λy. {y/x} M if y not free in M
  - e.g., λx. (* 2 x) is same as λz. (* 2 z)
- **β rule:** function application is substitution of argument for free parameter
  - e.g., (λx.(* 2 x) 4) is (* 2 4); more generally, ((λx. M) E) is {E/x} M
- **η rule:** these 2 functions always yield same results on equal arguments:
- ((λx. M) E) and E, if x is not free in E.

Evaluation Order

- More than one possible evaluation order
  - e.g., ((λx. (* x x)) (+ 2 3)) can be evaluated as
  ((λx.(* x x)) 5) -->(* 5 5) --> 25 *(by value)* OR
  ((λx.(* x x) (+ 2 3)) --> (* (+ 2 3) (+ 2 3)) -->
  (* 5 5) --> 25 *(by name)*
- ***by name* evaluation is *lazy*** in that the argument is not evaluated unless it is used,
- e.g., (define (foo v w) (if (> v 0) (* v v) (* w v))) evaluates as
  (foo 3 (fact 500)) will not need to evaluate (fact 500), an expensive calculation, unless it is necessary because of the code of foo
- ***by value* evaluation is *eager***

Normal Form

- Normal form implies there are no more function evaluations possible in the lambda term
- **Church-Rosser theorem**
  - 1936 Alonzo Church and J. Barkley Rosser – both mathematicians/ logicians
  - Normal forms are **unique**
  - If there is a normal form, (by name) substitution will find it
- Not every expression has a normal form
  - e.g., if bar is (λx.(x x)), evaluate (bar bar)

Lambda Calculus

- Universal *theory of functions*
- λ-calculus (Church, Rosser), 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 Teminology

- 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 ≡ λy. *2+y* -- λ calculus
  - bound variable or argument
  - *function body*
- λx.x (identity function)
- λy. 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; ((λx.x) λy.2) is λy.2;

((λz. 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 and its descendants.
- No analogue in C for an unnamed function, (Lisp lambda expression)

Function Application

- Is a left associative operator
  (f g h) is ((f g) h)
- λx.M x is same as λx.(M x)
- Function application has highest precedence
- *Currying* (cf. *Haskell Curry*)
  Area of triangle is λb. λh.(b*h)/2
  (Area 3) is a function, λh.(3*h)/2, that describes the area of a family of triangles all with base 3
  ((Area 3) 7) = 3 * 7 / 2 = 10.5
  Recall that in *curried form*, a function takes its arguments one-by-one

Type Signatures

- Can write function Area (λb. λh.(b*h)/2)

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:

  ((λy. 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(Y Z)= free(Y) ∪ free(Z)
- x ∉ 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 is 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 naively evaluate (plus b 3) we get into trouble!

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

(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!

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 | x} x = u //u not bound in M=x

  {u | x} (x u) = (u u) //u not bound in M=(x u)

  {λx.x | x} x = λx.x //λx.x not bound in M=x

  {u | x} y = y //no free occurrences of x in M=y so no sub

  {u | x} λx.x = λx.x //no free occurrences of x in M= λx.x so no sub

  {u | x} (λu.x) = {u | x} (λz.x) = λz.u

  {u | x} (λu.u) = {u | x} (λz.z) = λz.z

  Examples of need for change of variables.

Reductions

- β-reduction (λx.M) N = {N | x} M with above rules
- α-reduction (λx.M) = λz.{z | x} M, if z not free in M (allows change of bound variable names)
- η-reduction (λ 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)] change vars
= [{ (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 Sethi p 555

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 Sethi p 555

	M	{N | x} M
3.3a)	$\lambda x\ .P$	$\lambda x\ .P$

never substitute for a bound variable within its scope

3.3b)	$\lambda y\ .P$	$\lambda y\ .P$

if there are no free occurrences of x in P

3.3c)	$\lambda y\ .P$	$\lambda y\ .\{N \mid x\}\ P$

when there are no free occurrences of y in N

3.3d)	$\lambda y\ .P$	$\lambda z\ .\{N \mid x\}\ \{\ z \mid y\}\ P$

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

{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!

EG from Principles of Functional Programming,
H. Glaser, C. Hankin, D. Till, Prentice Hall, 1984