Lambda Calculus-2

- Church-Rosser theorem
  - Supports referential transparency of function application
  - Says if it exists, normal form of a term is UNIQUE

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 if a normal form exists, then all reduction sequences lead to it

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!

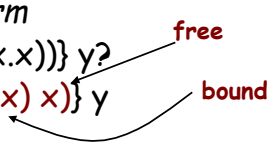
Normal Form

- 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?

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



Equality of Terms

- How check equality of 2 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

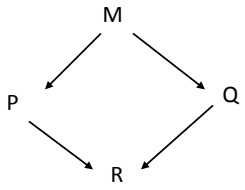
Church Rosser Property

- (GHT)Theorem 1: If a λ-expression reduces to a normal form, it is unique
- (GHT)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*

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

Church Rosser Property (Better Statement)

- (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
  - Theorem says all reduction sequences progress towards the same end result if they all terminate



Demonstration of CR by Example

(λx.λy.x-y) ((λz.z) 2) ((λr.r+2) 3) ~ f g h

first eval

second eval

substituting for x first:

= (λy.((λz.z) 2) - y) ((λr.r+2) 3), simplify

= (λy.2-y) ((λr.r+2) 3), apply lambda-expr

= 2 - ((λr.r+2) 3), apply lambda-expr

= 2 - 5

= -3

Demonstration 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), simplify
= (λx.x - 5) ((λz.z) 2), apply lambda-expr
= (((λz.z) 2) – 5), apply lambda-expr
= (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++, Scheme, 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
- In between 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.