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 \(\lambda\)-expression have a normal form? NO, because there are terms which cannot be simplified, yet they contain redices
  
  - \((\lambda x. x) (\lambda x. x)\) = \((\lambda y. y y) (\lambda x. x)\), \(\alpha\)-reduction  
  
  = \((\lambda x. x) (\lambda x. x)\), \(\beta\)-reduction  
  
  this term has no normal form  
  
  - \((\lambda x. x x) (\lambda x. x x)\) = \((\lambda y. y y y) (\lambda x. x x)\) , \(\alpha\)-red  
  
  = \((\lambda x. x x) (\lambda x. x x) (\lambda x. x x)\), \(\beta\)-red  
  
  this term grows as we apply \(\beta\)-reductions!

---

- If \(add6 = \lambda x. x+6\), \(twice = \lambda f. \lambda x. f (f x)\), what is value of \((twice \ add6)\)?  
  
  \((twice \ add6)\) = \((\lambda f. \lambda z. f (f z)) (\lambda x. x+6)\)  
  
  = \(\lambda z. ((\lambda x. x+6) ((\lambda x. x+6) z))\)  
  
  = \(\lambda z. ((\lambda x. x+6) (z+6))\)  
  
  = \(\lambda z. (z+12)\), normal form  
  
  - normal form of \{\(\lambda x. ((\lambda z. z x) (\lambda x. x))\)\} \(y\)?  
  
  \{\(\lambda x. ((\lambda z. z x) (\lambda x. x))\)\} \(y\) = \{\(\lambda x. ((\lambda x. x) x)\)\} \(y\)  
  
  = \{\(\lambda x. x\)\} \(y\)  
  
  = \(y\)
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 machines)
• Same term may have terminating and nonterminating $\beta$-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 $\lambda$-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 $\lambda.$ in A is to the left of first $\lambda.$ in B, then A is to the left of B
  – A redex to left of all other redices in a $\lambda$-expression is leftmost

Principles of Functional Programming,
H. Glaser, C. Hankin, D. Till, Prentice Hall, 1984
Church Rosser Property (Better Statement)

• (Sethi) Theorem: For \( \lambda \)-expressions \( M, P, Q \), let \( \Rightarrow \) stand for a sequence of \( \alpha \) and \( \beta \)-reductions. If \( M \Rightarrow P \) and \( M \Rightarrow Q \) then \( \exists \) a term \( R \) such that \( P \Rightarrow R \) and \( Q \Rightarrow R \)

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

\[
\begin{aligned}
M & \quad P \\
\quad & \quad R \\
\quad & Q
\end{aligned}
\]

Demonstration of CR by Example

\((\lambda.x.\lambda.y.x-y)((\lambda.z.z)\ 2)((\lambda.r.r+2)\ 3)\)  

\[
\begin{array}{c}
\text{first eval} \\
\text{second eval}
\end{array}
\]

substituting for \( x \) first:

= \((\lambda.y.((\lambda.z.z)\ 2) - \ y) \ ((\lambda.r.r+2)\ 3)\), simplify

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

= 2 - ((\lambda.r.r+2)\ 3), apply lambda-expr

= 2 - 5

= -3
Demonstration of CR by Example

\[(\lambda x. \lambda y. x - y) ((\lambda z. z) \ 2) ((\lambda r. r + 2) \ 3)\]

substitute for \( y \) first:

\[= (\lambda x. x - ((\lambda r. r + 2) \ 3)) ((\lambda z. z) \ 2), \text{simplify}\]

\[= (\lambda x. x \ - \ 5) ((\lambda z. z) \ 2), \text{apply lambda-expr}\]

\[= (((\lambda z. z) \ 2) \ - \ 5), \text{apply lambda-expr}\]

\[= ( 2 - 5)\]

\[= -3, \text{the same result!}\]

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 $\beta$-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
$(\lambda y z. (\lambda x. x) z (y z)) (\lambda x. x)$
Reduction Order

- Start with fully parenthesized expression:
  - \((\lambda v. e) (i)\) - 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 reduces in c, then those in b, and then reduce \((c b)\).

Example 1

(Sethi, p560) \[\{[\lambda y.\lambda z. ((\lambda x. x) z) (y z)]\} (\lambda x. x)\] = \((c b)\)

- call by value: reduce c. \([\lambda y.\lambda z. (z (y z))]\) = \((c' b)\) where b already reduced. reduce \((c' b)\) yielding
  \(\lambda z. (z ((\lambda x. x) z)) = \lambda z. (z (c'' b''))\). reduce \((c'' b'')\) which yields \(\lambda z. (z z)\), the final term.

- call by name: c is an abstraction (form i), so instantiate b directly into c yielding \(\lambda z. ((\lambda x. x) z) (\lambda x. x) z)\) = \(\lambda z. (c* b*)\)
  now reduce c* so we get an abstraction (form i.), yielding \(z\).
  then can perform final reduction of \(\lambda z. (z (\lambda x. x) z)\), yielding \(\lambda z. z z\), the final term, same as above.
Example 2

\(((\lambda x.\lambda y.x)\, z)\, (\lambda r.r\, r)\, (\lambda s.s\, s)) = (c\, b).

**call by value:** reduce \(c\) to yield \(((\lambda y.z)\, ((\lambda r.r\, r)\, (\lambda s.s\, s)))\) which is \(((\lambda y.z)\, (c'\, b'))\). reduce \(c'\, b'\) yielding \(((\lambda y.z)\, ((\lambda s.s\, s)\, (\lambda s.s\, s)))\). 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 \(((\lambda y.z)\, ((\lambda r.r\, r)\, (\lambda s.s\, s)))\). now substitute \(b\) into the reduced \(c\), yielding \(z\), because there is no bound \(y\) in \(\lambda y.z\). \(z\) is the normal form for the above term, by definition.

Example 3

\(\{ (\lambda z.\, (\lambda x.x+6)\, ((\lambda x.x+6)\, z))\, 1\} = \{ c\, b \} \)

\((c'\, b')\)

**call by value:** reduce reduces in \(c = (c'\, b')\) where \(b' = (c''\, b'')\).
\((c''\, b'')\) evaluates to \(b' = z+6\), yielding \(((\lambda z.\, (\lambda x.x+6)\, (z+6))\, 1\). now evaluating \((c'\, b')\) yields \(((\lambda z.\, (z+6)+6)\, 1\) = \(((\lambda z.\, z+12)\, 1\)
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 \(((\lambda x.x+6)\, ((\lambda x.x+6)\, 1)) = (c^*\, b^*)\). substitute \(b^*\) into \(c^*\) yielding \(((\lambda x.x+6)\, 1) + 6 = (c^\wedge\, b^\wedge) + 6\). substitute \(b^\wedge\) into \(c^\wedge\) yielding \((1 + 6) + 6 = 7 + 6 = 13\).