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
• $\lambda$-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 $\lambda$-calculus is equivalent to those which can be simulated on Turing machines.
• All partial recursive functions can be defined in $\lambda$-calculus.
• Pure $\lambda$-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 ($\lambda x.M$)
  – function application (M N)

Function Definition (Abstraction)

• $F(y) = 2 + y$ -- mathematics
• $F \equiv \lambda y. 2+y$ -- $\lambda$ calculus
  – bound variable or argument
  – function body
• $\lambda x.x$ (identity function)
• $\lambda y. 2$ (constant function whose value is 2)
Function Application

- **Process**: take the argument and substitute it everywhere in the function body for the parameter
  
  \[(F \ 3) \text{ is } 2 + 3 = 5; \quad ((\lambda \ x. x) \ \lambda. y. 2) \text{ is } \lambda \ y. 2; \]
  
  \[(\lambda \ z. z+5) \ 3 \text{ is } 3+5 = 8\]

- **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 \(((f \ g) \ h)\)
- \(\lambda \ x. M \ x\) is same as \(\lambda \ x. (M \ x)\)
- Function application has highest precedence
- **Currying** (cf. *Haskell Curry*)
  Area of triangle is \(\lambda \ b. \lambda \ h.(b*h)/2\)
  \((\text{Area}\ 3)\) is a function, \(\lambda \ h.(3*h)/2\), that describes the area of a family of triangles all with base 3
  ((\text{Area}\ 3)\ 7) = 3 \times 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: \(\alpha \times \beta \rightarrow \gamma\), 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 $\lambda x$ in front of the $\lambda$ expression:

  $$((\lambda y. y) \ y) \text{ is } y$$
  
  *bound* *free*

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

  free occurrence of $z$

  binding occurrence of $z$

  $(\lambda y. z)$

  $(\lambda z. z)$

  free occurrence of $z$

Free and Bound Variables

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

  $\text{free}(\lambda x. M) = \text{free}(M) - \{x\}$

- 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
  - \((\lambda x. M) \ N\) as *substituting* \(N\) for \(x\) in \(M\); written as \(\{N \mid x\} \ M\)

- **Rules** - Sethi, p551
  
  1. If free variables of \(N\) have no bound occurrences in \(M\), then \(\{N \mid x\} \ M\) formed by replacing all free occurrences of \(x\) in \(M\) by \(N\).

Substitution

\[ \text{plus} \equiv \lambda a. \lambda b. \ a + b \]

then \((\text{plus} \ 2) \equiv \lambda b. \ 2 + b\) but if we evaluate \((\text{plus} \ b \ 3)\) we get into trouble!

\[
\begin{align*}
(\text{plus} \ b \ 3) &= (\lambda a. \lambda b. \ a + b) \ b \ 3 \\
&= (\lambda b. \ b + b) \ 3 \\
&= 3 + 3 = 6
\end{align*}
\]

\[
\begin{align*}
(\text{plus} \ b \ 3) &= (\lambda a. \lambda c. \ a + c) \ b \ 3 \\
&= (\lambda c. \ b + c) \ 3 \\
&= b + 3, \text{ what we expected!}
\end{align*}
\]

**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
  \[ \{u \mid x\} \ x = u \quad \{u \mid x\} \ (x \ u) = (u \ u) \]
  \[ \{\lambda x.x \mid x\} \ x = \lambda x.x \quad \{u \mid x\} \ y = y \]
  \[ \{u \mid x\} \lambda x.x = \lambda x.x \]
  \[ \{u \mid x\} \ (\lambda u.x) = \{u \mid x\} \ (\lambda z.x) = \lambda z.u \]
  \[ \{u \mid x\} \ (\lambda u.u) = \{u \mid x\} \ (\lambda z.z) = \lambda z.z \]
  [Examples of need for change of variables.]

Reductions

- \(\beta\)-reduction \( (\lambda x.M) \ N = \{N \mid x\} \ M \) with above rules
- \(\alpha\)-reduction \( (\lambda x.M) = \lambda z.\{z \mid x\} \ M, \) if z not free in M (allows change of bound variable names)
- \(\eta\)-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 \(\beta\)-equality of terms

Lambda Calculus © BGR, Fall05
Example

Evaluate \((\lambda xyz \cdot xz (yz)) (\lambda x \cdot x) (\lambda y \cdot y)\)

\((\lambda xyz \cdot (xz (yz))) (\lambda x \cdot x) (\lambda y \cdot y)\), 2 \(\alpha\)-reductions + fully parenthesize

\[= \{ (\lambda abz \cdot (a z (b z))) (\lambda x \cdot x) \} (\lambda y \cdot y)\]

\[= \{ (\lambda bz \cdot ((\lambda x \cdot x) z (b z))) \} (\lambda y \cdot y), \{\lambda x. x \mid a\}\]

\[= \{ \lambda bz \cdot ((\lambda x \cdot x) z) (b z)) \} (\lambda y \cdot y), \text{ fully parenthesize}\]

\[= \{ \lambda z. (z (b z)) \} (\lambda y \cdot y), \{z \mid x\}\]

\[= \{ \lambda z. (z ((\lambda y \cdot y) z))\}, \{\lambda y. y \mid b\}\]

\[= \{ (\lambda z \cdot z z)\}, \{z \mid y\}\]

- Note: we picked the order of \(\beta\)-reductions here

Substitution Rules cf Sethi p 555, GHH p 49

\[
\begin{align*}
M & \quad \{N \mid x\} M \\
x & \quad N \\
y & \quad M \\
\text{if } M \text{ a variable, then if } M \not\equiv x \text{ get } M, \text{ else get } N \text{ (3.1 GHH)} \\
PQ & \quad \{N \mid x\} P \{N \mid x\} Q \\
\text{result of substitution applied to function application is} \\
\text{to apply that substitution to the function and its} \\
\text{argument and then perform the resulting} \\
\text{application(3.2 GHH)}
\end{align*}
\]
Substitution Rules

M \rightarrow \{N \mid x\} \ M
3.3a) \ \lambda \ x \ . \ P \rightarrow \lambda \ x \ . \ P

\textit{never substitute for a bound variable within its scope}

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

\textit{if there are no free occurrences of } x \ \text{in } P

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

\textit{when there are no free occurrences of } y \ \text{in } N

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

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

Substitution Rules

- All these checks are aimed at ensuring that we don’t link variable occurrences that are independent!
- Our example \((\lambda \ a . \lambda \ b . a + b) \ b\), would use 3.3d to change variables before doing the substitution
- Normal form of a term - a form which can allow no further \(\beta\) or \(\eta\) reductions
  - No remaining \((\lambda x. M) \ N\), called a \textit{redex} or term which can be reduced
Example GHH, p50

\{y \mid x\} \lambda y. x y \quad \text{use 3.3d to change bound var}
\lambda z. \{y \mid x\} (\{z \mid y\} (x \ y)) \ \text{apply 3.2 for fcn appln}
\lambda z. \{y \mid x\} (\{z \mid y\} (x) \ \{z \mid y\} (y)) \ \text{apply 3.1 twice}
\lambda z. \{y \mid x\} (x \ z) \ \text{apply 3.2}
\lambda z. (\{y \mid x\} (x) \ {y \mid x\} (z)) \ \text{apply 3.1 twice}
\lambda z. y z \quad \text{final result;}
\quad \text{compare this to what we started with!}

Church Rosser Property

- Fundamental result of \(\lambda\)-calculus:
  - Result of a computation is independent of the order in which \(\beta\)-reductions are applied
  - Leads to referential transparency in functional PL’s
  - Another interpretation is that most terms in the \(\lambda\)-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

- 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 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 \( \lambda \)-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 \( Q \Rightarrow R \)
  - Says all reduction sequences progress towards the same end result if they all terminate

\[ \begin{array}{c}
M \\
\downarrow \\
P \\
\downarrow \\
R \\
\uparrow \\
Q \end{array} \]

“Proof of CR by Example”

\[
(\lambda x.\lambda y. x-y) ((\lambda z. z) \ 2) ((\lambda r. r+2) \ 3) \quad \overset{\text{~ f g h}}{\text{first eval}}
\]

substituting for \( x \) first:

\[
= (\lambda y.((\lambda z. z) \ 2) - y) ((\lambda r. r+2) \ 3)
\]
\[
= (\lambda y.2-y) ((\lambda r. r+2) \ 3)
\]
\[
= 2 - ((\lambda r. r+2) \ 3)
\]
\[
= 2 - 5
\]
\[
= -3
\]
“Proof of CR by Example”

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

substitute for y first:
\(= (\lambda x.\ x - ((\lambda r.\ r+2)\ 3))\ ((\lambda z.\ 2)\)
\(= (\lambda x.\ x - 5)\ ((\lambda z.\ 2)\)
\(= (((\lambda z.\ 2)\ 2) - 5)\)
\(= (2 - 5)\)
\(= -3,\ the\ same\ result!\)

substituting for x first:
\(= (\lambda y.\ ((\lambda z.\ 2)\ - y)\ ((\lambda r.\ r+2)\ 3)\)
\(= (\lambda y.\ 2-y)\ ((\lambda r.\ r+2)\ 3)\)
\(= 2 - ((\lambda r.\ r+2)\ 3)\)
\(= 2 - 5\)
\(= -3\)

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?
  – \( \text{hd ( tl (list(n)) )} \) 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 \textbf{inner} and \textbf{outer} refer to nesting of terms

\[
(\lambda \, yz. \, (\lambda \, x. x) \, z \, y \, z) \, (\lambda \, x. x)
\]
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 reduces 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) (λs. s))) = (c  b).

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

{(%z. (%x.x+6) ((%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 {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.