Lambda Calculus

▶ Lambda calculus serves as a formal technique for defining the semantics of functional programming:
    ▶ Defines referentially transparent naming mechanism by formally specifying parameter passing mechanisms and scoping rules
    ▶ Represents basic data types in terms of functions
    ▶ Implements recursion without violating referential transparency
    ▶ Provides a model for interpreters for functional languages
Computation as Rewriting

- Computation transforms existing values into resulting values
- In \( \lambda \)-Calculus
  - Represent initial values as \( \lambda \)-Calculus expressions
  - Transform \( \lambda \)-Calculus expressions into new expressions
  - Interpret resulting \( \lambda \)-Calculus expression to get result
- Transformations \( \equiv \) rewriting expressions according to rules
- If rule \( \rho \) transforms \( \lambda \)-Calculus expression \( E_1 \) to \( E_2 \) write
  \[
  E_1 \xrightarrow{\rho} E_2
  \]

Preservation of Semantics

- Every transformed expression preserves semantics of expression
  - Represent: \( 2+0 \) as \( \lambda \)-calculus expression \( E_1 \)
  - Transform \( E_1 \xrightarrow{\rho} E_2 \)
    - Eg. transform \textquoteleft{}2+0\textquoteright{} into \textquoteleft{}2\textquoteright{}, or transform \textquoteleft{}4*(2+0)\textquoteright{} into \textquoteleft{}4*2\textquoteright{}
    - Now \( E_2 \) must still represent \( 2+0 \) (perhaps \textquoteleft{}2\textquoteright{})
Syntax

- Lambda Calculus expressions use
  - lower case letters: \{ a, b, c, d, ... \}
  - four symbols: \( \lambda \mid ( ) \)
- Each letter represents a function
- Three kinds of expressions
  - function constant
  - function definition
  - function application

Examples of Lambda Calculus Expressions

- \( f \)  
  a function identifier

- \( (f \ g) \)  
  an application of function \( f \) to \( g \)

- \( (\lambda x \mid (x \ y)) \)  
  definition of function with parameter \( x \) and body \( (x \ y) \) 
  Body is an application!

- \( (\lambda y \mid (\lambda x \mid (y \ (y \ x)))) \)  
  definition of function with parameter \( y \) and body 
  \( (\lambda x \mid (y \ (y \ x))) \)
The λ Definition

- Function definitions have the form: \((\lambda \langle \text{identifier} \rangle \mid \langle \text{expression} \rangle)\)
- \(\lambda\) is followed by a single identifier, called a formal parameter or variable
- When the \(\lambda\) is applied to an argument \(E\), the formal parameter will bind to \(E\).
  - Below the \(\langle \text{identifier} \rangle \) binds to value \(y\)
    \[ (\lambda x \mid \langle \text{body} \rangle) \ y \]
- Appearances of the identifier in the body of the \(\lambda\) are called instances
  - Every instance refers to the same expression — the one \(\lambda\) was called on. In the example below, each instance of \(x\) refers to \(y\).
    \[ (\lambda x \mid (x \ x)) \ y \]
Notational Conveniences

- Where order of operations is clear, can drop brackets
- Can use spacing arbitrarily to aid readability
  - In function definition
    
    $$\lambda x \, (x \, y) \equiv \lambda x \, y \equiv \lambda x \, xy$$
  - In function application
    
    $$(f \, g) \equiv f \, g \equiv fg$$

Associativity of $\lambda$-calculus operators

- Associative operators like integer addition can be composed in any order
  
  $$(1+2)+3 = 1 + (2+3)$$
- Non-associative operators like subtraction cannot be composed in any order
  
  $$(5-3)-2 \neq 5-(3-2)$$
- $\lambda$-application is not associative
  
  ($\lambda C$ must be able to represent non-associative functions!)
- By convention, $\lambda$-application is left-associative... terms group from the left

  $$f \, g \, h \equiv ((f \, g) \, h) \neq (f \, (g \, h))$$
More Left-associativity Examples

\[ f \, g \, h \equiv (f \, g) \, h \quad \text{YES} \]

\[ (\lambda a \mid (a \, (a \, b))) \equiv (\lambda a \mid a \, a \, b) \quad \text{NO} \]

\[ (\lambda z \mid (a \, (\lambda y \mid b))) \equiv (\lambda z \mid a \, (\lambda y \mid b)) \quad \text{YES} \]

\[ a \, b \, (c \, d) \equiv a \, b \, c \, d \quad \text{NO} \]

\[ (a \, b) \, c \, d \equiv a \, b \, c \, d \quad \text{YES} \]

Free and Bound Variables

▶ An instance of variable \( v \) is \textit{bound} in expression \( E \) when it is:
  ▶ a formal parameter of a \( \lambda \)
  ▶ it is enclosed by a \( \lambda \) with parameter \( v \) within the expression \( E \)

\[
\left( \lambda x \mid z \right) \quad \text{bound} \\
\left( \lambda x \mid x \right) \quad \text{bound} \\
\left( \lambda x \mid y \, x \, z \right) \quad \text{bound} \\
\left( \lambda x \mid (\lambda y \mid x \, y) \right) \quad \text{bound} \\
\left( \lambda x \mid (\lambda y \mid x \, y) \right) \quad \text{bound}
\]
Free and Bound Variables

- A variable that is not bound is free

\[
\begin{align*}
y & \quad \text{free} \\
(\lambda x \mid y) & \quad \text{bound free} \\
(y (\lambda y \mid y)) & \quad \text{free bound bound} \\
(\lambda x \mid (q y)) & \quad \text{bound free}
\end{align*}
\]

- Bound and free instances of the same variable within an expression:

\[
(\lambda x \mid x y (\lambda y \mid x y z) y)
\]

- Free variables in an expression can be later bound in an enclosing expression

\[
(\lambda x \mid x y)
\]

- Variables in λ-calculus derive their meaning from the argument the enclosing λ is applied to

  - They cannot be "assigned" a new "value"
More Convenience: Collapsing Enclosing $\lambda$’s

- The scope of a $\lambda$ is the expression to which its bindings apply

$$(\lambda x \mid (\lambda y \mid y) \ x) \ (\lambda w \mid w) \ v$$

- Scope of outer $\lambda$ includes $(\lambda y \mid y) \ x$

- If the scopes of nested $\lambda$’s coincide, the arguments can be coalesced into a multi-argument $\lambda$

$$(\lambda x \mid (\lambda y \mid x \ y)) \equiv (\lambda xy \mid xy)$$

- Just notational convenience!!

$\lambda$-calculus Computation

- $\lambda$-calculus computation is ...

Reducing complex expression to “simpler” form

- $(\lambda z \mid (z \ y) \ w) \rightarrow (w \ y)$

  [Every occurrence of $z \rightarrow w$]

- $(\lambda z \mid (z \ y)) \ (\lambda w \mid w) \rightarrow (\lambda w \mid w) \ y)$

  [Every occurrence of $z \rightarrow (\lambda w \mid w)$]

  $(\lambda w \mid w) \ y) \rightarrow y$

  [Every occurrence of $w \rightarrow y$]

- Need to

  “Replace every occurrence of $z$ in $(z \ y)$ with $(\lambda w \mid w)$”

  “Replace every occurrence of $<identifier>$ in $<expression>$ with $<expression’>$”
Substitution

- \(\lambda\)-calculus rules uses a special substitution

- Generally: write substitution of \(x\) for \(y\) in expression \(\langle E\rangle\) as: \([x/y] \langle E\rangle\)

\[
\begin{align*}
[x/y] (z \ y) &\rightarrow (z \ x) \\
[(a \ b)/y] (\lambda z \ | (z \ y)) &\rightarrow (\lambda z \ (a \ b))
\end{align*}
\]

- In \(\lambda\)-calculus, only free variables are replaced:

\[
\begin{align*}
[x/y] (z \ (\lambda y | z y)) &\rightarrow (z \ (\lambda y | z y))
\end{align*}
\]

Legal Substitution

- Legal substitutions do not change meaning of an expression
  - Legal: Substitute \(x\) for \(y\)
    \[
    [x/y] (\lambda y | z y) \rightarrow (\lambda y | xz)
    \]
  - \textit{Illegal substitutions} introduce bindings not present in original expressions
    \[
    [z/y] (\lambda z | yz) \not\rightarrow (\lambda z | zz)
    \]
    - Illegal because variable named \(y\) in \((\lambda z | yz)\) was free but now, as \(z\), is bound
    \[
    [(\lambda x | xz) / y] (\lambda z | yz) \not\rightarrow (\lambda z | (\lambda x | xz) z)
    \]
    - Illegal because \(z\) was free in \((\lambda x | xz)\) but now is bound
\( \beta \) (Beta Rule): Function Application

- A function application \((\lambda x \mid \langle E \rangle) \langle F \rangle)\) has function \((\lambda x \mid \langle E \rangle)\) and argument \(\langle F \rangle\)

- \(\beta\)-rule: apply \((\lambda x \mid \langle E \rangle)\) to \(\langle F \rangle\)
  \[ \equiv \]
  substitute \(\langle F \rangle\) for every free occurrence of \(x\) in body \(\langle E \rangle\)

  \[
eval[ (\lambda x \mid \langle E \rangle) \langle F \rangle ) ]
  \equiv [\langle F \rangle/x]_\lambda E \quad \text{if } [\langle F \rangle/x]_\lambda \text{ is legal}
  \]

- \(\beta\) defines a relationship between manipulation of symbols and a computation

Substitution Legality and the \(\beta\)-rule

- \(\beta\)-rule starts with application: \(( (\lambda x \mid \langle E \rangle) \langle F \rangle )\)

- Substitution is illegal only if
  \[ \exists \text{ free occurrences of variables in } \langle F \rangle \text{ that would become bound in } \langle E \rangle \]

- Later, a way to fix things when a substitution would be illegal
\( \beta \) example: constant argument

\[(\lambda f \mid f \ x) \ s\]

\[\beta \rightarrow [s/f] (f \ x)\]

Free variables in \( s \) that would get bound? No, go ahead and substitute

\[\equiv (s \ x)\]

Can we do more? No - normal form

\( \beta \) example: \( \lambda \) argument

\[(\lambda f \mid f \ x) \ (\lambda y \mid y)\]

\[\beta \rightarrow [(\lambda y \mid y)/f] (f \ x)\]

Free vars in \( (\lambda y \mid y) \) get bound? No.

\[\equiv (\lambda y \mid y) \ x\]

\[\beta \rightarrow [x/\ y] \ y\]

Free vars in \( x \) get bound? No.

\[\equiv x\]
\[ (\lambda f \mid (f \ (f \ x))) \ s) \]
\[ \xrightarrow{\beta} [s / f] \ (f \ (f \ x)) \]
Free vars in \( s \) get bound? No.
\[ \equiv (s \ (s \ x)) \]

Can we do more? No - in normal form

\[ (\lambda y \mid y) \ (\lambda y \mid y) \ x) \]
\[ \xrightarrow{\beta} \]
\[ (\lambda y \mid y) \ [x / y \ y] \]
\[ (\lambda y \mid y) \ x) \]
Now are we done? No
\[ (\lambda y \mid y) \ x) \xrightarrow{\beta} [x / y \ y] \]
\[ \rightarrow x \]
\( \beta \) example: complex multiple substitution

\[
( (\lambda f \mid (f (f x))) (\lambda y \mid (g (g (g y)))) ) \\
\equiv (\lambda f \mid (f (f x))) (\lambda y \mid (g (g (g y)))) \\
\beta \rightarrow [(\lambda y \mid (g (g (g y)))) / f] (f (f x))
\]

Free vars in \((\lambda y \mid (g (g (g y)))) \) get bound? No

\[
\equiv (\lambda y \mid (g (g (g y)))) ((\lambda y \mid (g (g (g y)))) x))
\]

\[
( (\lambda y \mid (g (g (g y)))) (\lambda y \mid (g (g (g y)))) x)) \\
\beta \rightarrow ( (\lambda y \mid (g (g (g y)))) [x/y] (g (g (g y)))) \\
\equiv ( (\lambda y \mid (g (g (g y)))) (g (g (g x))))
\]

\[
( (\lambda y \mid (g (g (g y)))) (g (g (g y)))) \\
\beta \rightarrow [(g (g (g x))) / y] (g (g (g y))) \\
\rightarrow (g (g (g (g (g (g x))))) )
\]

A formal definition of \( \beta \)-substitution

\( \rightarrow (\lambda y \mid [\langle E \rangle / x] \langle F \rangle) \)

\( \langle E \rangle, \langle F \rangle, \langle G \rangle \) be \( \lambda \)-calculus expressions; \( x \) and \( y \) be distinct \( \lambda \)-calculus identifiers (constants)

\( [\langle E \rangle / x] x \rightarrow \langle E \rangle \)
\( [\langle E \rangle / x] y \rightarrow y \)
\( [\langle E \rangle / x] (\langle F \rangle \langle G \rangle) \rightarrow ( [\langle E \rangle / x] \langle F \rangle [\langle E \rangle / x] \langle G \rangle ) \)
\( [\langle E \rangle / x] (\lambda x \mid \langle F \rangle) \rightarrow (\lambda x \mid \langle F \rangle) \)
\( [\langle E \rangle / x] (\lambda y \mid \langle F \rangle) \) where \( \langle E \rangle \) has no free instances of \( y \)

\( \rightarrow (\lambda y \mid [\langle E \rangle / x] \langle F \rangle) \)
Variable Names in $\lambda$-calculus

- The identifier used to represent a BOUND variable is irrelevant

- Meaning of variable based on the $\lambda$ that introduces it ... and how it is used in $\lambda$’s body

- If we change the identifier used in a formal parameter and all of its bound occurrences, the meaning of the expression is unaltered

\[(\lambda x \mid x) \equiv (\lambda y \mid y) \quad YES!\]
\[(\lambda x \mid x) \equiv (\lambda x \mid y) \quad NO!\]
\[(\lambda x \mid (\lambda y \mid x \ y)) \equiv (\lambda a \mid (\lambda b \mid a \ b)) \quad YES!\]
\[(\lambda x \mid (\lambda y \mid x \ y)) \equiv (\lambda a \mid (\lambda b \mid b \ a)) \quad NO!\]

Variables in $\lambda$-calculus

\[(\lambda x \mid (\lambda y \mid x \ y)) \equiv (\lambda y \mid (\lambda x \mid y \ x)) \quad YES!\]
\[(\lambda x \mid (\lambda y \mid x \ y)) \equiv (\lambda y \mid (\lambda x \mid x \ y)) \quad NO!\]
\[(\lambda x \mid (\lambda w \mid w) \ x) \equiv (\lambda y \mid (\lambda w \mid w) \ y) \quad YES!\]
\[(\lambda x \mid (\lambda y \mid y) \ x) \equiv (\lambda y \mid (\lambda y \mid y) \ y) \quad YES \ (same \ as \ above!) \ ... \ but \ confusing!\]

Think ... $(\lambda y \mid (\lambda y \mid y) \ y)$
\( \alpha \) (Alpha Rule): Motivation

- The \( \beta \)-rule cannot be applied in ...
  \[
  ( (\lambda y \mid (\lambda z \mid yz)) \ z )
  \]
  \[\beta \rightarrow [z / y] \ (\lambda z \mid yz)\]
  \[\not\equiv (\lambda z \mid zz) \quad \text{Why not?}\]
  \( z \) was free in \( z \) but \textit{bound} in the result \( \Rightarrow \) substitution is illegal!

- \[
  (\lambda y \ (\lambda z \mid yz)) \ (\lambda x \mid xz)
  \]
  \[\beta \rightarrow [(\lambda x \mid xz) / y] \ (\lambda z \mid yz)\]
  \[\not\equiv (\lambda z \mid (\lambda x \mid xz)z)\]

- But, variable identifiers in and of themselves are irrelevant

- The \( \alpha \)-rule changes variable identifiers without altering meaning

\( \alpha \) (Alpha Rule): Renaming

- \( \alpha \)-rule: substitute a new identifier for the instances of any bound variable... as long as the substitution is legal

- Just choose an identifier \textit{not} used in the current expression, ... and substitution guaranteed to be legal

  \[
  (\lambda z \mid yz) \xrightarrow{\alpha:q/z} (\lambda q \mid yq)
  \]

  - Note: Replace the formal parameter \textit{and} EVERY instance of \( z \) with \( q \)
  - Note: \( q \) is a NEW variable, never used ...

Dr. B. Price and Dr. R. Greiner  
CMPUT 325 : Lambda Calculus Basics  
29
\( \alpha \) (Alpha Rule): Examples

\[(\lambda a \ | \ b (\lambda c \ | \ c \ a) \ d) \xrightarrow{\alpha:z/a} (\lambda z \ | \ b (\lambda c \ | \ c \ z) \ d)\]

Legal

\[(\lambda x \ | \ (\lambda y \ | \ x \ y \ z)) \xrightarrow{\alpha:y/z} (\lambda x \ | \ (\lambda y \ | \ x \ y \ y))\]

Illegal

Formal definition of \( \alpha \)

- Let \( \langle E \rangle \) and \( \langle F \rangle \) be \( \lambda \)-calculus expressions;
  - \( x \) and \( y \) be distinct \( \lambda \)-calculus constants
- Let \( z \) be a newly generated \( \lambda \) calculus constant

\([\langle E \rangle / x] (\lambda y \ | \ \langle F \rangle) \rightarrow (\lambda z \ | \ [\langle E \rangle / x] [z/y] \langle F \rangle)\)
Using $\alpha$ and $\beta$ Together I

\[(\lambda y (\lambda z | yz)) (\lambda x | xz)\]

- We could use $\beta$ rule to simulate applying the function

\[\xrightarrow{\beta} \left[ (\lambda x | xz)/y \right] (\lambda z | yz)\]

- Legal substitution? No. Free $z$ in $(\lambda x | xz)$ becomes bound

- Use $\alpha$ rule to rename variable

\[\xrightarrow{\alpha} \left[ (\lambda x | xz)/y \right] [q/z] (\lambda z | yz) \equiv \left[ (\lambda x | xz)/y \right] (\lambda q | yq)\]

Using $\alpha$ and $\beta$ Together II

- Now we can apply $\beta$-substitution

\[\left[ (\lambda x | xz)/y \right] (\lambda q | yq) \equiv (\lambda q | (\lambda x | xz) \ q)\]

\[(\lambda q | (\lambda x | xz) \ q) \xrightarrow{\beta} (\lambda q | [q / x ] \ xz) \equiv (\lambda q | \ q \ z)\]
We can represent any calculation as a λ calculus expression!!

▶ Turing Equivalents!

▶ Computation ≡ Apply α, β rules (many times!) to reduce given expression to "unreducible" form

▶ Interpret value of resulting expression as result of computation

▶ Computation requires only two rules

\( \eta \) (Eta Rule): Null Application

▶ Special case of the \( \beta \)-rule: \((\lambda x | \langle E \rangle) v \xrightarrow{\beta} [v/x] \langle E \rangle\)

▶ Accelerates Rule 5 of \( \beta \) substitution

▶ If \( x \) does not appear as a free variable in \( \langle E \rangle \), then \( \langle E \rangle \) doesn't change

▶ \( \eta \)-rule:

\[ ((\lambda a | c d) q) \rightarrow (c d) \]
\[ ((\lambda x | (\lambda x | x y)) v) \rightarrow (\lambda x | x y) \]
To implement a λ-calculus interpreter

- Must determine if each variable is free or bound
  - to determine potential clashes with free variables
- Faster to determine the status of variable x in \langle E \rangle, than to “build” a new expression without any changes
  \( \Rightarrow \eta \)-rule

On Reductions

- λ-calculus reduces expressions to “simpler” expressions using β and η rules
  - Why scare quotes?
- β-rule and η-rule are called reductions (α is not a reduction)
- If we can obtain \langle N \rangle from \langle M \rangle using a sequence of β and η operations, then \langle M \rangle is reducible to \langle N \rangle
- An expression that can be reduced is called a redux
- Can only reduce applications that contain function definitions
  - Cannot reduce \( f, (f \ g), (\lambda f \ | \ (f \ g)) \)
  - Can reduce \( (\lambda x \ | \ (w \ x)) \ y \)
- An expression containing no reduxes is in normal form
  (i.e. a completed calculation)
Theoretical Questions

- So “interpretation” \( \equiv \) “reducing to normal form”

- Questions...
  - Is there more than one way to reduce an expression?
  - Is there one unique reduction for every expression?
  - Is every expression reducible?
  - If not, what are the implications?

- First topic: Order of reductions...

Order of Reductions: Normal I

- Normative Order: leftmost application first

- Which is leftmost function?

\[
\begin{align*}
(\lambda x | (\lambda y | x)) & \quad ((\lambda u | z) u) \\
(\lambda x | (\lambda y | x)) & \quad ((\lambda u | z) u) \\
\text{leftmost} & \quad (\lambda x | (\lambda y | x)) \quad ((\lambda u | z) u) \\
\beta & \quad [((\lambda u | z) u) / x] \quad (\lambda y | x) \\
\text{Free vars in } & \quad ((\lambda u | z) u) \text{ get bound? No} \\
\equiv & \quad (\lambda y \mid ((\lambda u | z) u))
\end{align*}
\]
Order of Reductions: Normal II

\((\lambda y \mid ((\lambda u \mid z) \; u))\) Left application?
\((\lambda y \mid ((\lambda u \mid z) \; u))\)

leftmost
\((\lambda y \mid ((\lambda u \mid z) \; u))\)

\[ \beta \rightarrow (\lambda y \mid [u/u] \; z) \]
Any free vars in \(u\) get bound? No.
\[ \rightarrow (\lambda y \mid z) \]
Done? Yes - normal form

Order of Reductions: Applicative I

▶ Applicative Order: innermost application first
▶ Like LISP: evaluate arguments first, then apply function

\((\lambda x \mid (\lambda y \mid x)) \; ((\lambda u \mid z) \; u)\)
\((\lambda x \mid (\lambda y \mid x)) \; (\lambda u \mid z) \; u)\)

innermost
\[(\lambda x \mid (\lambda y \mid x)) \; ((\lambda u \mid z) \; u)\]

\[ \beta \rightarrow (\lambda x \mid (\lambda y \mid x)) \; [u/u] \; (\lambda u \mid z) \]
any free vars in \(u\) get bound? No.
\[ \equiv (\lambda x \mid (\lambda y \mid x)) \; z \]
Done? Nope
Order of Reductions: Applicative II

\[(\lambda x | (\lambda y | x)) \ z \quad \text{Innermost?} \]
\[\ \ \\
\begin{align*}
(\lambda x | (\lambda y | x)) \ z \\
\text{innermost} \\
(\lambda x | (\lambda y | x)) \ z \\
\beta \\
[z / x] \ (\lambda y | x)
\end{align*}
\]
Any free vars in \(x\) get bound? No
\[\equiv (\lambda y | z)\]
Done? Yes - normal form

Order of Reductions: Comment

- You may choose
  - normative (left-most legal application) or
  - applicative order (innermost legal application) or
  - ...

- However, since \(\lambda\) calculus is left-associative,
  - at any given level within an expression, you must reduce the leftmost of a series of applications first

- So in: \(abc(cde)\)
  - May apply \(c\) to \(d\) (applicative) or \(a\) to \(b\) (normative)
  - CANNOT apply \(b\) to \(c\) nor \(c\) to \((cde)\) nor \(d\) to \(e\)
    (violation of left-associativity)
Let $\langle A \rangle, \langle B \rangle, \langle C \rangle, \langle D \rangle$ be λ-calculus expressions and $\xrightarrow{1}, \xrightarrow{2}, \xrightarrow{3}$ and $\xrightarrow{4}$ be reductions of zero or more steps.

Church and Rosser Theorem I

- If $\langle A \rangle \xrightarrow{1} \langle B \rangle$ and $\langle A \rangle \xrightarrow{2} \langle C \rangle$,
- Then $\exists \langle D \rangle \xrightarrow{3}$ and $\xrightarrow{4}$ s.t. $\langle B \rangle \xrightarrow{3} \langle D \rangle$ and $\langle C \rangle \xrightarrow{4} \langle D \rangle$

i.e., different reductions of $\langle A \rangle$ can always be reduced to the same expression (function)

Uniqueness Corollary

- Corollary: Given two reductions $\langle A \rangle \xrightarrow{1} \langle B \rangle$ and $\langle A \rangle \xrightarrow{2} \langle C \rangle$
  - If $\langle B \rangle$ and $\langle C \rangle$ are in normal form, neither can be reduced further
  - By Church and Rosser I, we can reduce $\langle C \rangle$ and $\langle B \rangle$ to an identical form in zero or more steps
  - Since both $\langle C \rangle$ and $\langle B \rangle$ are irreducible, the required reduction must be of length zero and $\langle C \rangle$ and $\langle B \rangle$ are identical
  - All reductions that result in a normal form, result in the same unique normal form!

- ... does every reduction result in normal form???
Existence Theorem

- Church and Rosser Theorem II
  - If $\langle A \rangle \rightarrow \langle B \rangle$ and $\langle B \rangle$ is in normal form
    then $\langle A \rangle \rightarrow \langle B \rangle$ by normative order reduction

- If $\langle A \rangle$ can be reduced to a normal form,
  it can be found by normal order reduction

- Not every expression has a normal form

  $$(\lambda x | x x) \ (\lambda x | x x)$$

  $$(\lambda x | x x) \ (\lambda x | x x) \rightarrow (\lambda x | x x) \ (\lambda x | x x)$$

- Because reductions are not guaranteed to terminate,
  the equivalence of $\lambda$-calculus expressions is undecidable

- This result predates the halting problem!

Reduction Orders as Parameter Types

- Applicative order reduction evaluates innermost applications first
  - $\approx$ evaluating arguments before passing them
  - Can be interpreted as "call by value"

- Normative order reduction evaluates leftmost applications first
  - $\approx$ passing unevaluated expressions to function
  - Can be interpreted as "call by name"
  - Passed-in expressions must still be evaluated in body of function
Completeness of Applicative vs. Normal Order

- The argument to \((\lambda x \mid y)\) does not matter
  - \((\ (\lambda x \mid y)\langle E\rangle) \rightarrow y\) for any \(\langle E\rangle\)
  - Here, expression \(\langle E\rangle\) is an *unneeded* argument
  - \(\eta\)-reductions

- Applicative order may evaluate *unneeded* arguments
  - If argument does not have a normal form, evaluation of arguments will not halt

- Normal order does not evaluate unneeded arguments
  - If only unneeded arguments lack a normal form, then Normal order will find a normal form

- \(\exists\) formulas that have a normal form that can be found by normal order reduction, but that cannot be found by applicative order reduction

Reducible by Normal Example

\[(\lambda z \ (\lambda y \mid y))\ (\ (\lambda x \mid x x) \ (\lambda x \mid x x) \ )\]

\[(\lambda z \ (\lambda y \mid y)) \ (\ (\lambda x \mid x x) \ (\lambda x \mid x x) \ )\]

leftmost application

\[(\lambda z \ (\lambda y \mid y))\ (\ (\lambda x \mid x x) \ (\lambda x \mid x x) \ )\]

\[\beta \rightarrow [\ (\ (\lambda x \mid x x) \ (\lambda x \mid x x) \ )/ z] \ (\lambda z \ (\lambda y \mid y))\]

Any free vars get bound? No.

\[\equiv (\lambda y \mid y)\]
Irreducible by Applicative Example

- Under Applicative order

\[(\lambda z \ (\lambda y \ | \ y)) \ (\ (\lambda x \ | \ x \ x) \ (\lambda x \ | \ x \ x) \ )\]

innermost application

\[(\lambda z \ (\lambda y \ | \ y)) \ (\ (\lambda x \ | \ x \ x) \ (\lambda x \ | \ x \ x) \ )\]

\[\beta \rightarrow (\lambda z \ (\lambda y \ | \ y)) \ [\ (\lambda x \ | \ x \ x) \ / \ x \ ] \ x \ x\]

Any free vars in \((\lambda x \ | \ x \ x)\) get bound? No.

\(\equiv (\lambda z \ (\lambda y \ | \ y)) \ (\ (\lambda x \ | \ x \ x) \ (\lambda x \ | \ x \ x) \ )\)

Notice anything fishy here?

We are back to what we started with!

Example 1: Normal Order

\[(\lambda x \ | \ (\lambda y \ x \ | \ x) \ z)) \ (\lambda x \ | \ x \ y)\]

First step? Identify leftmost applicable function

\[(\lambda x \ | \ (\lambda y \ x \ | \ x) \ z)) \ (\lambda x \ | \ x \ y)\]

Recall \((\lambda y \ x \ | \ x)\) means \((\lambda y \ | \ (\lambda x \ | \ x))\)

\[(\lambda x \ | \ (\lambda y \ | \ (\lambda x \ | \ x) \ z)) \ (\lambda x \ | \ x \ y)\]

\[\beta \rightarrow [\ (\lambda x \ | \ x \ y) \ / \ x \ ] \ (\lambda y \ | \ (\lambda x \ | \ x) \ z)\]

Free vars in \((\lambda x \ | \ x \ y)\)? get bound?

No free instances of \(x\) within \((\lambda y \ | \ (\lambda x \ | \ x) \ z)\)

\[\equiv (\lambda y \ | \ (\lambda x \ | \ x) \ z)\]

\[(\lambda y \ | \ (\lambda x \ | \ x) \ z) \ \eta \rightarrow (\lambda x \ | \ x)\]
Example 1: Applicative

\[(\lambda x \mid (\lambda y \ x \mid x) \ z) \ (\lambda x \mid x \ y)\]

First step? Identify *innermost* applicable function

\[(\lambda x \mid (\lambda y \ x \mid x) \ z) \ (\lambda x \mid x \ y)\]

Recall: \((\lambda y \ x \mid x)\) means \((\lambda y \mid (\lambda x \mid x))\)

\[(\lambda x \mid (\lambda y \mid x)) \ z \ (\lambda x \mid x \ y)\]

\[\beta \rightarrow (\lambda x \mid [z/y] (\lambda x \mid x) (\lambda x \mid x \ y))\]

No free instances of \(y\) in \((\lambda x \mid x)\)

\[\equiv (\lambda x \mid (\lambda x \mid x)) \ (\lambda x \mid x \ y)\]

\[\eta \rightarrow (\lambda x \mid x)\]

Example 2: Normal Order I

\[(\lambda x \mid (\lambda y \mid x)) \ ((\lambda x \mid y) \ x)\]

First task: find leftmost applicable function

\[(\lambda x \mid (\lambda y \mid x)) \ ((\lambda x \mid y) \ x)\]

\[\beta \rightarrow [(((\lambda x \mid y) \ x) \ / x] \ (\lambda y \mid x)\]

Free vars in \(((\lambda x \mid y) \ x)\) get bound? YES!

\[\alpha \rightarrow (\lambda y \mid ((\lambda x \mid y) \ x))\]

Use \(\alpha\) rule.

\[\alpha \rightarrow [((\lambda x \mid y) \ x) \ / x] \ [z/y] \ (\lambda y \mid x)\]

\[\equiv [((\lambda x \mid y) \ x) \ / x] \ (\lambda z \mid x)\]

\[(\lambda z \mid ((\lambda x \mid y) \ x))\]
Example 2: Normal Order II

\[(\lambda z \mid (\lambda x \mid y) \; x)\]  
\[(\lambda z \mid (\lambda x \mid y) \; x)\]  
\[\text{leftmost}\]  
\[(\lambda z \mid ((\lambda x \mid y) \; x))\]  
\[\eta\rightarrow (\lambda z \mid y)\]

Example 2: Applicative I

\[(\lambda x \mid (\lambda y \mid x)) \; ((\lambda x \mid y) \; x)\]  
First step? Find innermost application  
\[(\lambda x \mid (\lambda y \mid x)) \; ((\lambda x \mid y) \; x)\]  
\[\text{innermost}\]  
\[(\lambda x \mid (\lambda y \mid x)) \; ((\lambda x \mid y) \; x)\]  
\[\eta\rightarrow (\lambda x \mid (\lambda y \mid x)) \; y\]
Example 2: Applicative II

\((\lambda x \mid (\lambda y \mid x)) \ y\)
\(\underbrace{(\lambda x \mid (\lambda y \mid x)) \ y}_{\text{innermost}}\)
\[\beta\rightarrow [y/x] \ (\lambda y \mid x)\]

Free vars get bound? Yes
\[\alpha\rightarrow [y/x][z/y] (\lambda y \mid x)\]
\[\equiv [y/x] (\lambda z \mid x)\]
\[\equiv (\lambda z \mid y)\]

Example 3: Normal I

\(((\lambda x \ y \mid y) \ ((\lambda x \mid x \ x) \ (\lambda x \mid x \ x))) \ a\)

First step? Find leftmost application
\((\lambda x \ y \mid y) \underbrace{((\lambda x \mid x \ x) \ (\lambda x \mid x \ x)) \ a}_{\text{leftmost}}\)

Re-call: \((\lambda x \ y \mid y) \equiv (\lambda x \mid (\lambda y \mid y))\)
\((\lambda x \mid (\lambda y \mid y)) \underbrace{((\lambda x \mid x \ x) \ (\lambda x \mid x \ x))) \ a}_{\text{leftmost}}\)

\[\eta \rightarrow (\lambda y \mid y) \ a\]
\[\rightarrow [a/y] \ y \equiv a\]
Example 3: Applicative I

\((\lambda x \ y \ | \ y) \ ((\lambda x \ | \ x \ x) \ (\lambda x \ | \ x \ x))\) a

First step? Find innermost application.

\((\lambda x \ y \ | \ y) \ ((\lambda x \ | \ x \ x) \ (\lambda x \ | \ x \ x))\) a

innermost

\[ \beta \rightarrow (\lambda x \ y \ | \ y) \ [ (\lambda x \ | \ x \ x) \ / \ x] \ (x \ x) \] a

Will free vars in get \((\lambda x \ | \ x \ x)\) bound? No free vars!

\[ \equiv (\lambda x \ y \ | \ y) \ ((\lambda x \ | \ x \ x) (\lambda x \ | \ x \ x)) \] a

We get the original expression back again!

Shortcuts for Multi-argument \(\lambda\)'s

\((\lambda x \ y \ z \ | \ ⟨E⟩) \ ⟨A⟩ \ ⟨B⟩ \ ⟨C⟩\)

\[ \equiv (\lambda x \ | \ (\lambda y \ | \ (\lambda z \ | \ ⟨E⟩)) \ ⟨A⟩ \ ⟨B⟩ \ ⟨C⟩\]

\[ \beta \rightarrow [⟨A⟩/x] \ (\lambda y \ | \ (\lambda z \ | \ ⟨E⟩)) \ ⟨B⟩ \ ⟨C⟩\]

If \(⟨A⟩\) has free \(y\) or \(z\), must re-name \((\lambda y \ | \ (\lambda z \ | \ ⟨E⟩))\)

\[ \beta \rightarrow [⟨B⟩/y] \ (\lambda z \ | \ ⟨E⟩)\]

If \(⟨B⟩\) has free \(z\), must rename \((\lambda z \ | \ ⟨E⟩)\)

\[ \beta \rightarrow [⟨C⟩/z] \ ⟨E⟩\]

If \(⟨C⟩\) has free var bound in \(⟨E⟩\), must rename ...
Example of Multi-argument λ’s

△ Our basic solution method

\[(\lambda x y | x y) (\langle N \rangle y) (\langle M \rangle)\]
\[\equiv (\lambda x | (\lambda y | x y)) (\langle N \rangle y) (\langle M \rangle)\]
\[\beta \rightarrow [(\langle N \rangle y)/x] (\lambda y | x y) (\langle M \rangle)\]
Free vars in \((\langle N \rangle y)\) get bound? Yes!
Must rename \(y\) in \((\lambda y | x y)\). Say \(z\)
\[\alpha \rightarrow [(\langle N \rangle y)/x] [z/y] (\lambda y | x y) (\langle M \rangle)\]
\[\equiv [\langle N \rangle y]/x] (\lambda z | x z) (\langle M \rangle)\]
\[\equiv (\lambda z | (\langle N \rangle y) z) (\langle M \rangle)\]
\[\beta \rightarrow [\langle M \rangle/z] (\langle N \rangle y) z \equiv (\langle N \rangle y) (\langle M \rangle)\]

△ Note: we replaced \(y\) with \(z\),
but then immediately replace \(z\) with \(\langle M \rangle\)

Example of Multi-argument λ’s

△ In general, can perform multiple substitutions in parallel

△ If substituting in parallel,
given \((\lambda x | (\lambda y \ldots )) (\langle A \rangle) (\langle B \rangle)\),
we do not have to check for free \(y\)’s in \(\langle A \rangle\) as \(\langle B \rangle\) will be substituted for the "(\(\lambda y\)" and any free \(y\)’s in \(\langle A \rangle\) will remain free.

△ Example done with multiple substitution

\[(\lambda x y | x y) (\langle N \rangle y) (\langle M \rangle)\]
\[\beta \rightarrow [(\langle N \rangle y)/x, (\langle M \rangle)/y] (x y)\]
\[\equiv (\langle N \rangle y) (\langle M \rangle)\]

△ N.B: still need to check for free vars that get bound when considering substitution of \(\langle B \rangle\) in the body of the \((\lambda y \ldots )\) clause.
Curried functions

- Can represent $n$-ary functions as nested unary functions
  \[
  (\lambda x y \mid \langle E \rangle) a b \equiv (\lambda x (\lambda y \langle E \rangle)) a b
  \]

- Can treat an $n$-ary function as a unary function that returns an $n-1$-ary function

- Treating $n$-ary function as unary function that returns a function is called *currying*.