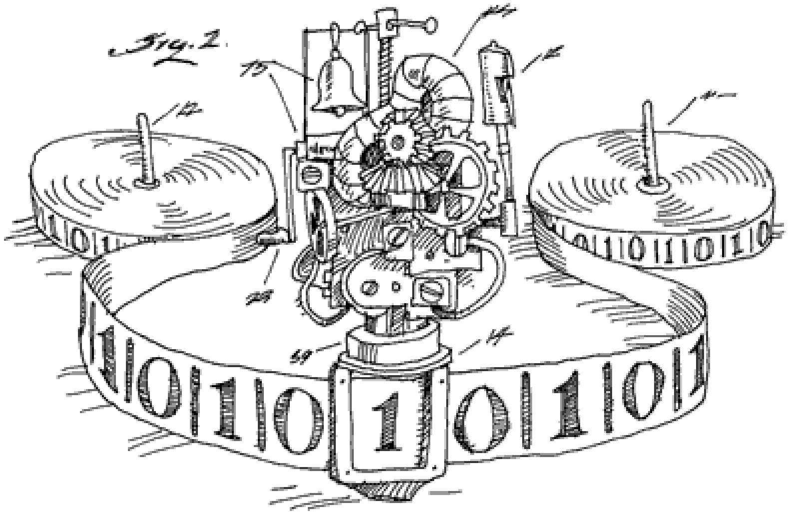
Lambda Calculus

Oded Padon & Mooly Sagiv

(original slides by Kathleen Fisher, John Mitchell, Shachar Itzhaky, S. Tanimoto)

Computation Models

- Turing Machines
- Wang Machines
- Counter Programs
- Lambda Calculus



Historical Context

Like Alan Turing, another mathematician, Alonzo Church, was very interested, during the 1930s, in the question “What is a computable function?”

He developed a formal system known as the pure lambda calculus, in order to describe programs in a simple and precise way.

Today the Lambda Calculus serves as a mathematical foundation for the study of functional programming languages, and especially for the study of “denotational semantics.”

Reference: http://en.wikipedia.org/wiki/Lambda_calculus

Basics

- Repetitive expressions can be compactly represented using functional abstraction
- Example:
    - (5 * 4* 3 * 2 * 1) + (7 * 6 * 5 * 4 * 3 * 2 *1) =
    - factorial(5) + factorial(7)
    - factorial(n) = if n = 0 then 1 else n * factorial(n-1)
    - factorial= λn. if n = 0 then 0 else n * factorial(n-1)
    - factorial= λn. if n = 0 then 0 else n * apply (factorial (n-1))

Untyped Lambda Calculus

$t ::=$	terms
x	variable
$\lambda x. t$	abstraction
$t\ t$	application

Terms can be represented as abstract syntax trees

Syntactic Conventions

- Applications associates to left

  $e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$

- The body of abstraction extends as far as possible
  - $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ (x\ y)\ x)$

Lambda Calculus in Python

$(\lambda x. x)\ y$ `(lambda x: x) (y)`

Substitution

- Replace a term by a term
  - $x + ((x + 2) * y)[x \mapsto 3, y \mapsto 7] = ?$
  - $x + ((x + 2) * y)[x \mapsto z + 2] = ?$
  - $x + ((x + 2) * y)[t \mapsto z + 2] = ?$
- More tricky in programming languages
  - Why?

Free vs. Bound Variables

- An occurrence of x is bound in t if it occurs in λx. t
  - otherwise it is free
  - λx is a binder
- Examples
  - Id= λx. x
  - λy. x (y z)
  - λz. λx. λy. x (y z)
  - (λx. x) x

FV: t $\rightarrow 2^{Var}$ is the set free variables of t

$FV(x) = \{x\}$

$FV(\lambda x. t) = FV(t) - \{x\}$

$FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$

Beta-Reduction

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

$[x \mapsto s]\ x = s$

$[x \mapsto s]\ y = y$ if $y \neq x$

$[x \mapsto s]\ (\lambda y.\ t_1) = \lambda y.\ [x \mapsto s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$

$[x \mapsto s]\ (t_1\ t_2) = ([x \mapsto s]\ t_1)\ ([x \mapsto s]\ t_2)$

Beta-Reduction

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

redex

$(\lambda\ x.\ x)\ y \Rightarrow_\beta\ y$

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \Rightarrow_\beta u\ r\ (\lambda\ x.\ x)$

$(\lambda\ x\ (\lambda w.\ x\ w))\ (y\ z) \Rightarrow_\beta\ \lambda w.\ y\ z\ w$

Alpha- Conversion

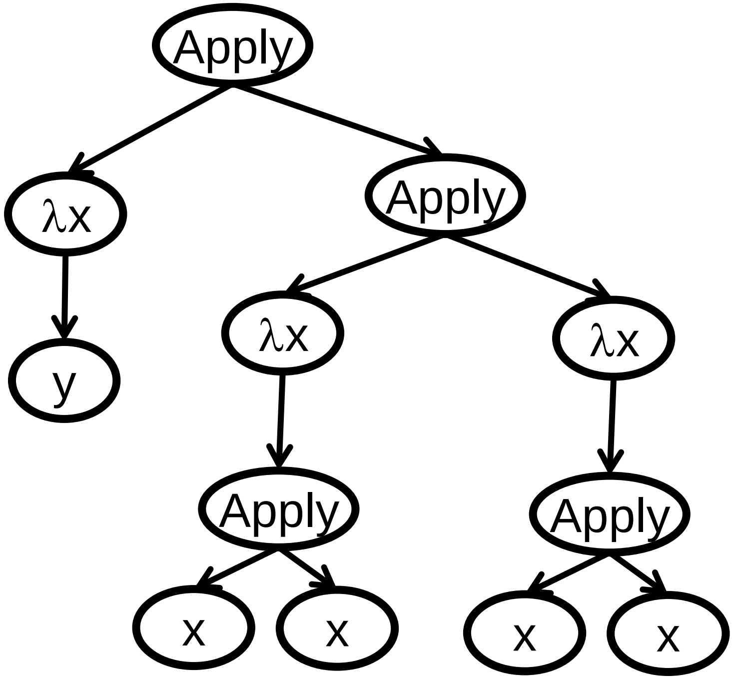
Alpha conversion:
Renaming of a bound variable and its bound occurrences

$\boldsymbol{\lambda x.\lambda y.y \quad \Rightarrow_\alpha \quad \lambda x.\lambda z.z}$

Divergence

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

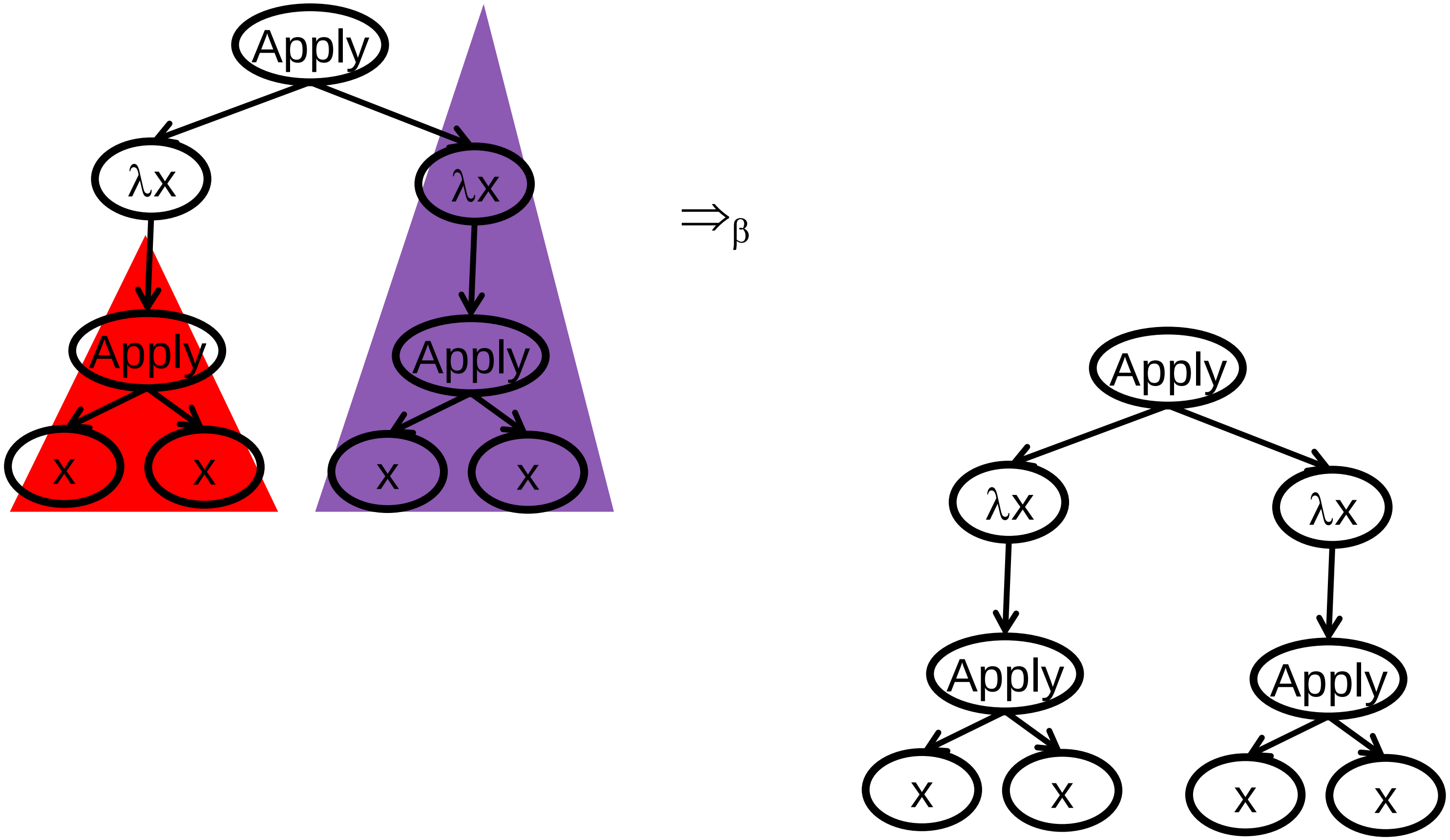
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Divergence

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

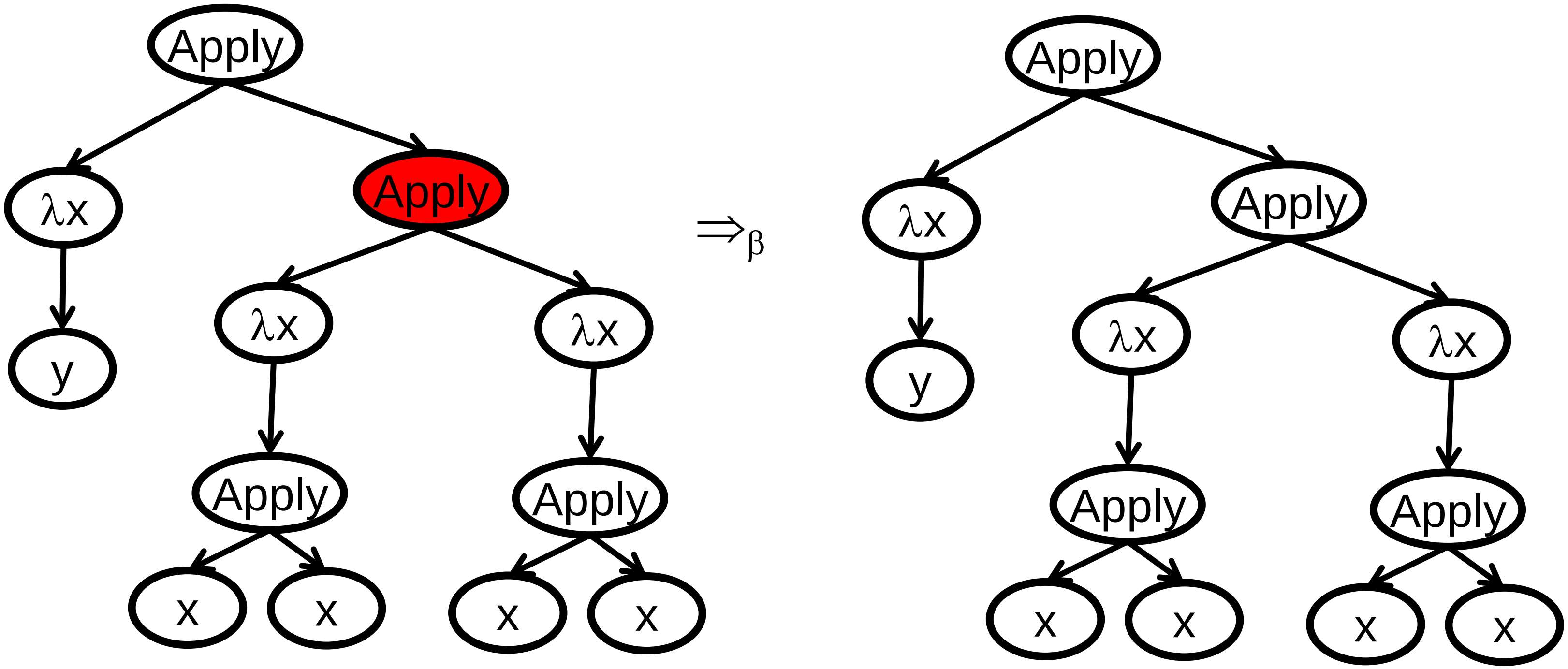
$(\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x))$



Different Evaluation Orders

$(\lambda\, x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

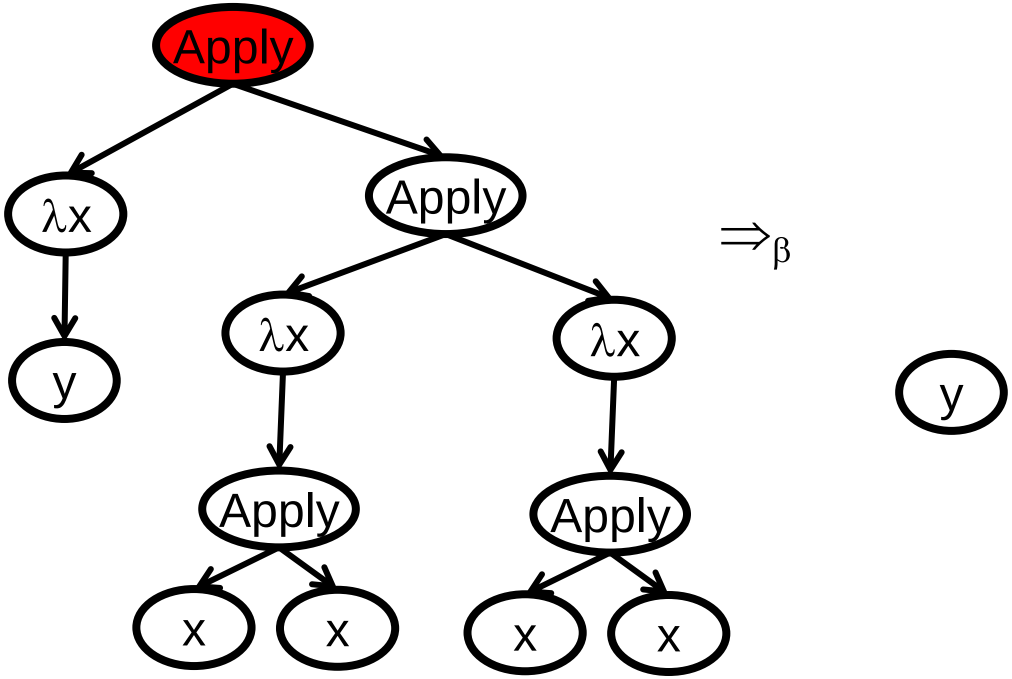
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

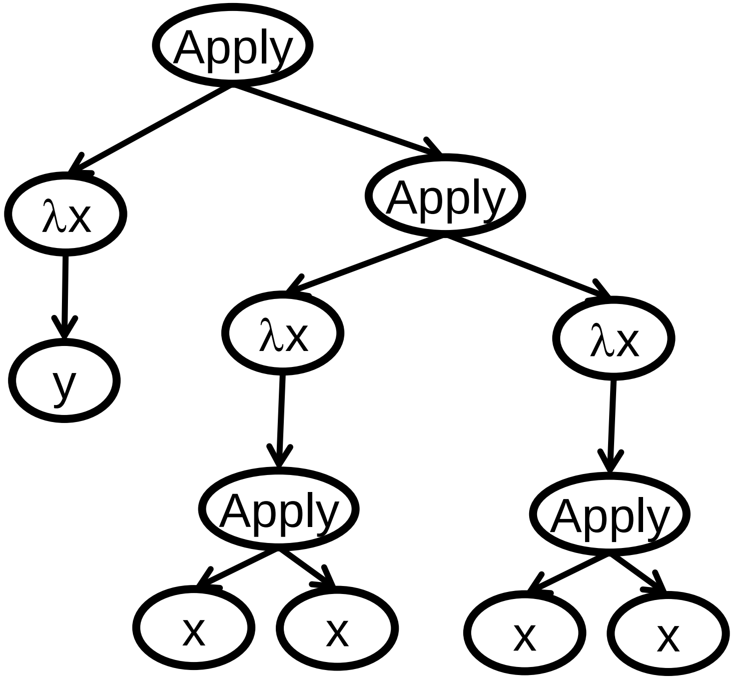
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$

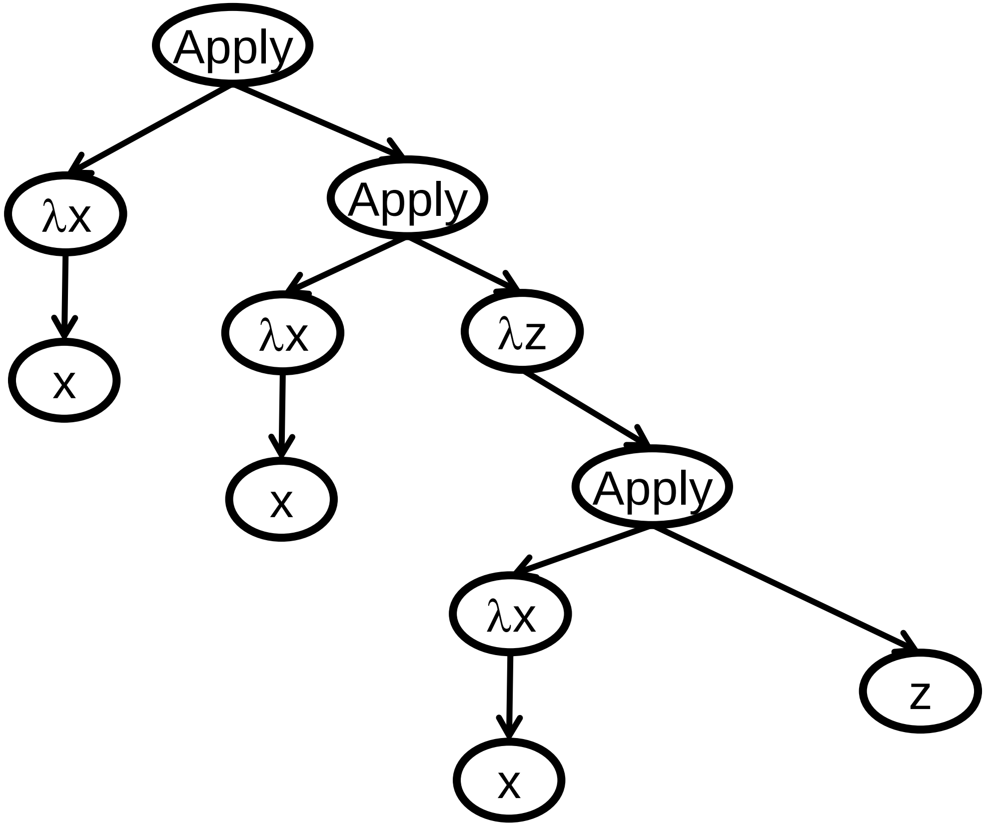


```
def f():
    while True: pass

def g(x):
    return 2

print g(f())
```

Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$ (β-reduction)

$(\lambda\ x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)) \equiv \text{id}\ (\text{id}\ (\lambda z.\ \text{id}\ z))$

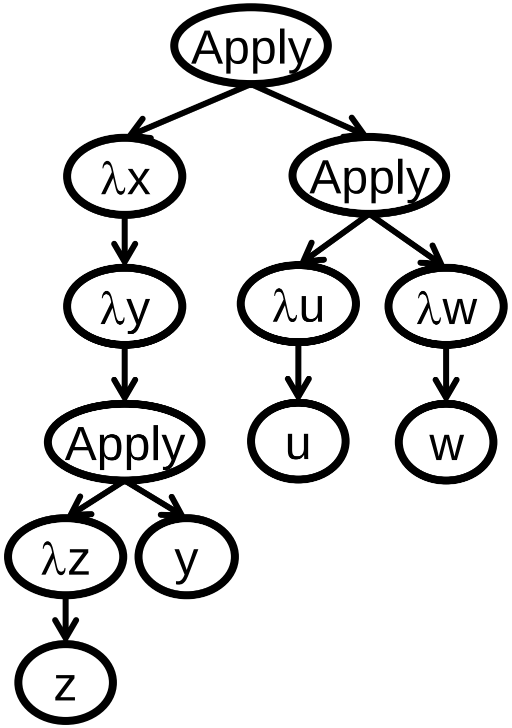


Order of Evaluation

- Full-beta-reduction
  - All possible orders
- Applicative order call by value (Eager)
  - Left to right
  - Fully evaluate arguments before function
- Normal order
  - The leftmost, outermost redex is always reduced first
- Call by name
  - Evaluate arguments as needed
- Call by need
  - Evaluate arguments as needed and store for subsequent usages
  - Implemented in Haskel

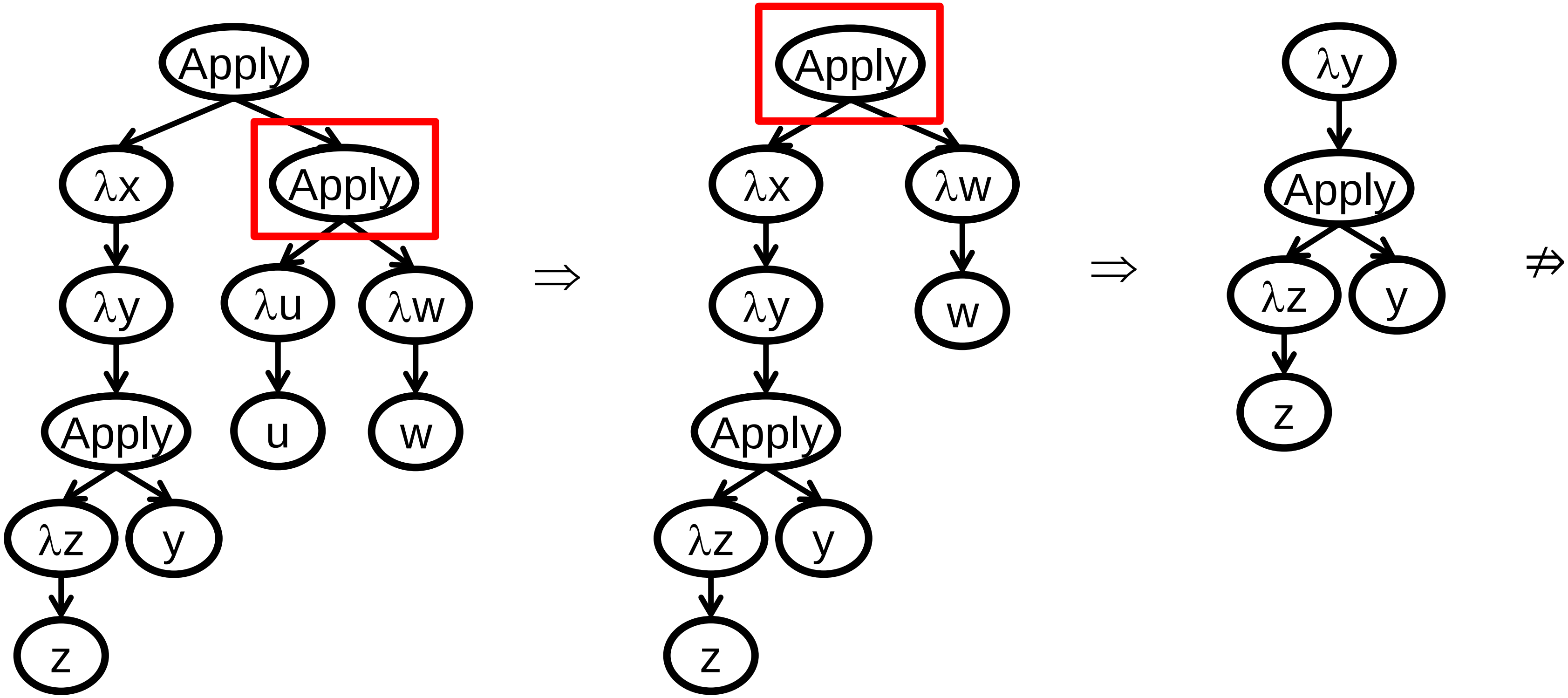
Different Evaluation Orders

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



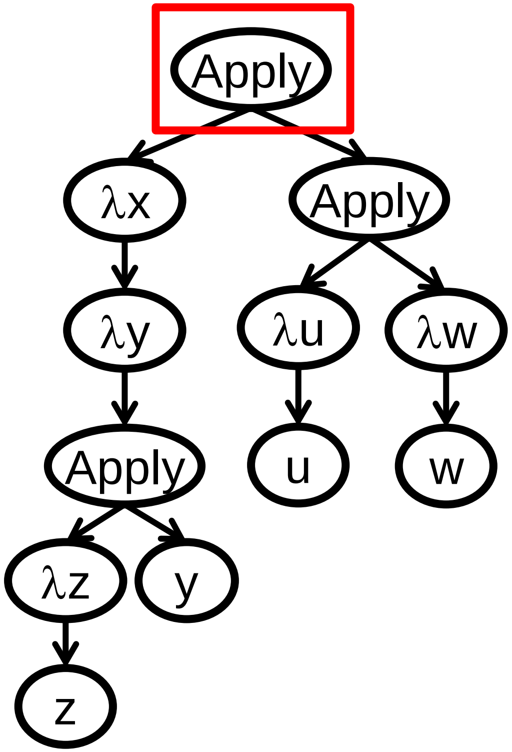
Call By Value

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))

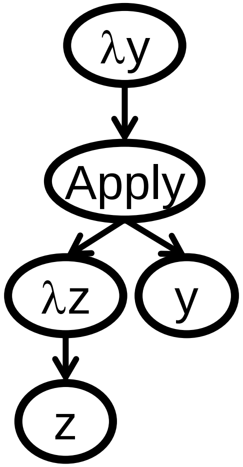


Call By Name (Lazy)

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



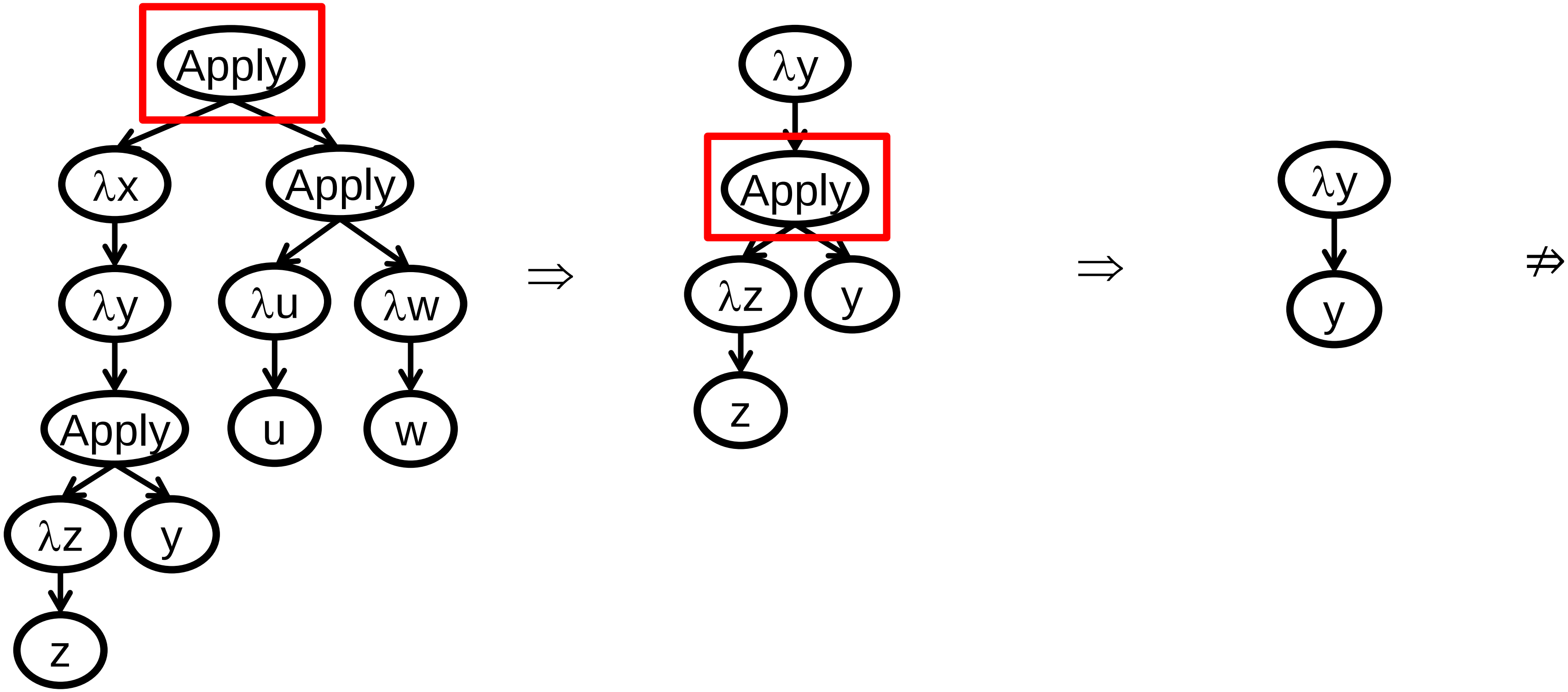
⇒



⇏

Normal Order

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Call-by-value Operational Semantics

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

v ::=	values
λ x. t	abstraction values
other values	

$$(\lambda\, x.\ t_1)\ v_2 \Rightarrow [x \mapsto v_2]\ t_1$$ (E-AppAbs)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ \ t_2\ \Rightarrow\ t'_1\ t_2}$$ (E-APPL1)

$$\frac{t_2 \Rightarrow t'_2}{v_1\ \ t_2\ \Rightarrow\ v_1\ t'_2}$$ (E-APPL2)

Programming in the Lambda Calculus

Multiple arguments

$f = \lambda(x, y).\ s$

-> Currying

$f = \lambda x.\ \lambda y.\ s$

$f\ v\ w =$

$(f\ v)\ w =$

$(\lambda x.\ \lambda y.s\ v)\ w \Rightarrow$

$\lambda y.[x \mapsto v]s)\ w) \Rightarrow$

$[x \mapsto v]\ [y \mapsto w]\ s$

Programming in the Lambda Calculus
Booleans

- tru = λt. λf. t
- fls = λt. λf. f
- test = λl. λm. λn. l m n
- test tru then else = (λl. λm. λn. l m n) (λt. λf. t)
- test fls then else = (λl. λm. λn. l m n) (λt. λf. f)
- and = λb. λc. b c fls
- or = ?

Programming in the Lambda Calculus
Numerals

- $c_0 = \lambda s.\ \lambda z.\ z$
- $c_1 = \lambda s.\ \lambda z.\ s\ z$
- $c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$
- $c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$
- $\text{succ} = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$
- $\text{plus} = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$
- $\text{times} = \lambda m.\ \lambda n.\ m\ (\text{plus}\ n)\ c_0$

> Turing Complete

Combinators

- A combinator is a function in the Lambda Calculus having no free variables
- Examples
  - λx. x is a combinator
  - λx. λy. (x y) is a combinator
  - λx. λy. (x z) is not a combinator
- Combinators can serve nicely as modular building blocks for more complex expressions
- The Church numerals and simulated booleans are examples of useful combinators

Loops in Lambda Calculus

- omega= $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$
- Recursion can be simulated
  - **Y = (λx . (λy. x (y y)) (λy. x (y y)))**
  - **Y f $\Rightarrow^*_\beta$ f (Y f)**

Factorial in the Lambda Calculus

Define H as follows, to represent 1 step of recursion.
Note that ISZERO, MULT, and PRED represent particular combinators that accomplish these functions

H = (λ f. λ n.(ISZERO n) 1 (MULT n (f (PRED n))))

Then we can create
FACTORIAL = Y H
= (λx . (λy. x (y y)) (λy. x (y y))) (λ f. λ n.(ISZERO n) 1 (MULT n (f (PRED n))))

Reference: http://en.wikipedia.org/wiki/Y_combinator

Consistency of Function Application

- Prevent runtime errors during evaluation
- Reject inconsistent terms
- What does ‘x x’ mean?
- Cannot be always enforced
  - if <tricky computation> then true else (λx. x)

Simple Typed Lambda Calculus

t ::=	terms
x	variable
λ x: T. t	abstraction
t t	application

T::=	types
T → T	types of functions

Summary: Lambda Calculus

- Powerful
- The ultimate assembly language
- Useful to illustrate ideas
- But can be counterintuitive
- Usually extended with useful syntactic sugars
- Other calculi exist
  - pi-calculus
  - object calculus
  - mobile ambients
  - …