Functional Programming

Mooly Sagiv
Original slides by John Mitchell

Topics

- Lambda Calculus
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion

Model of Computations

- Turing Machines

- Wang Machines

- 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

What is λ calculus

- A complete computational model
- An assembly language for functional programming
  - Powerful
  - Concise
  - Counterintuitive
- Can explain many interesting PL features

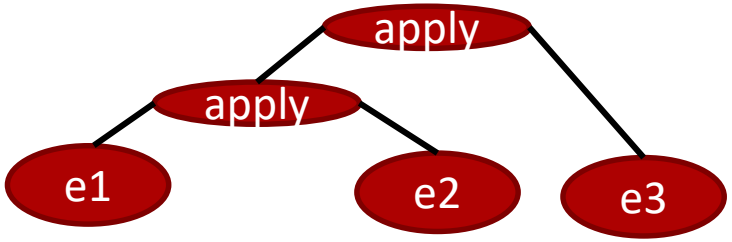
Basics

- 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
λ 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
  - $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ (x\ y)\ x)$

Untyped Lambda Calculus

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

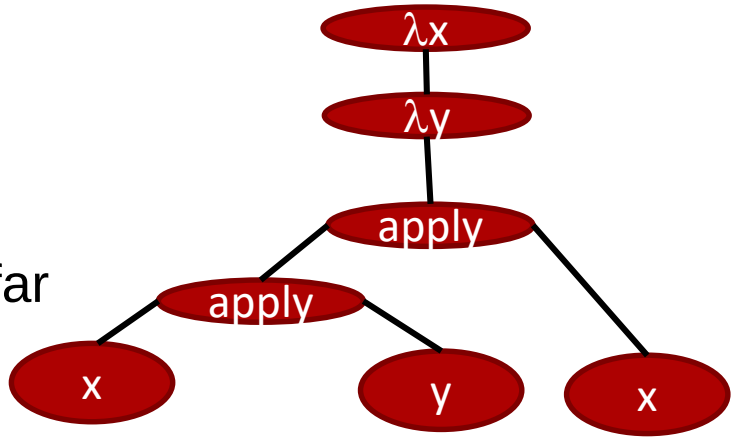
Terms can be represented as abstract syntax trees

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



Example Lambda Expressions

- λx.”1”
- λx.x
- λx.y
- λx.s x
- λx.s (s x)
- λf. λg. f g
- λb. “if” b “fls” “tru”
- λb. λb’. “if” b b’ “fls”

Lambda Calculus in Python

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

Substitution

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

Free vs. Bound Variables

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

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

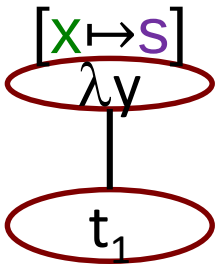
$(\lambda\, x.\, t_1)\; t_2 \Rightarrow_\beta [x \mapsto t_2]\; t_1$ (β-reduction)

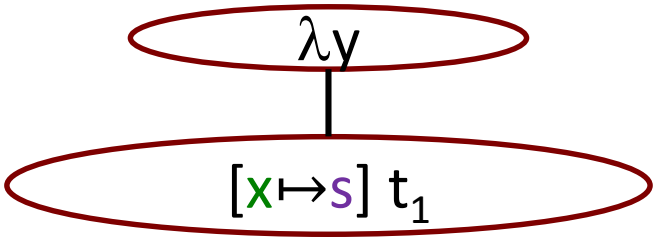
$[x \mapsto s]\; x = s$

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

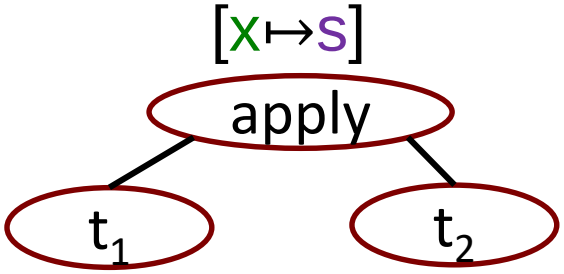
$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

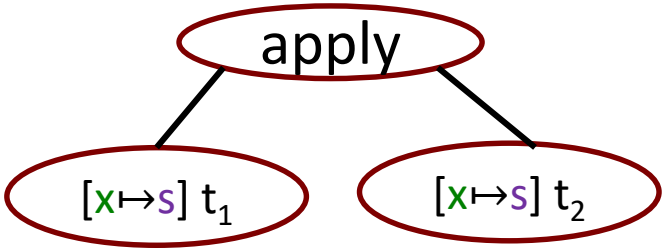
$[x \mapsto s]\ x = s$

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

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

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



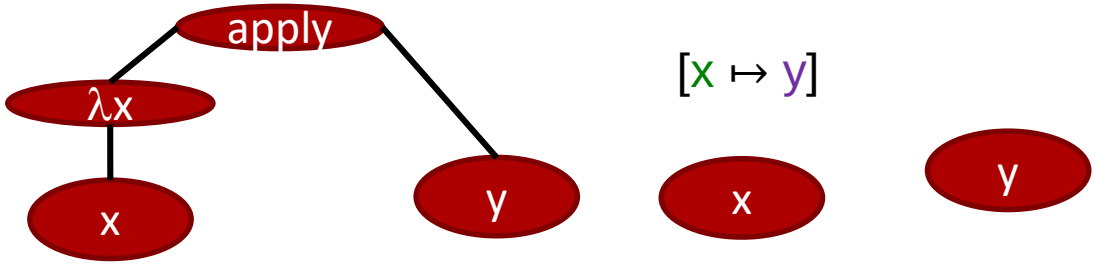


Example Beta-Reduction

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

redex

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

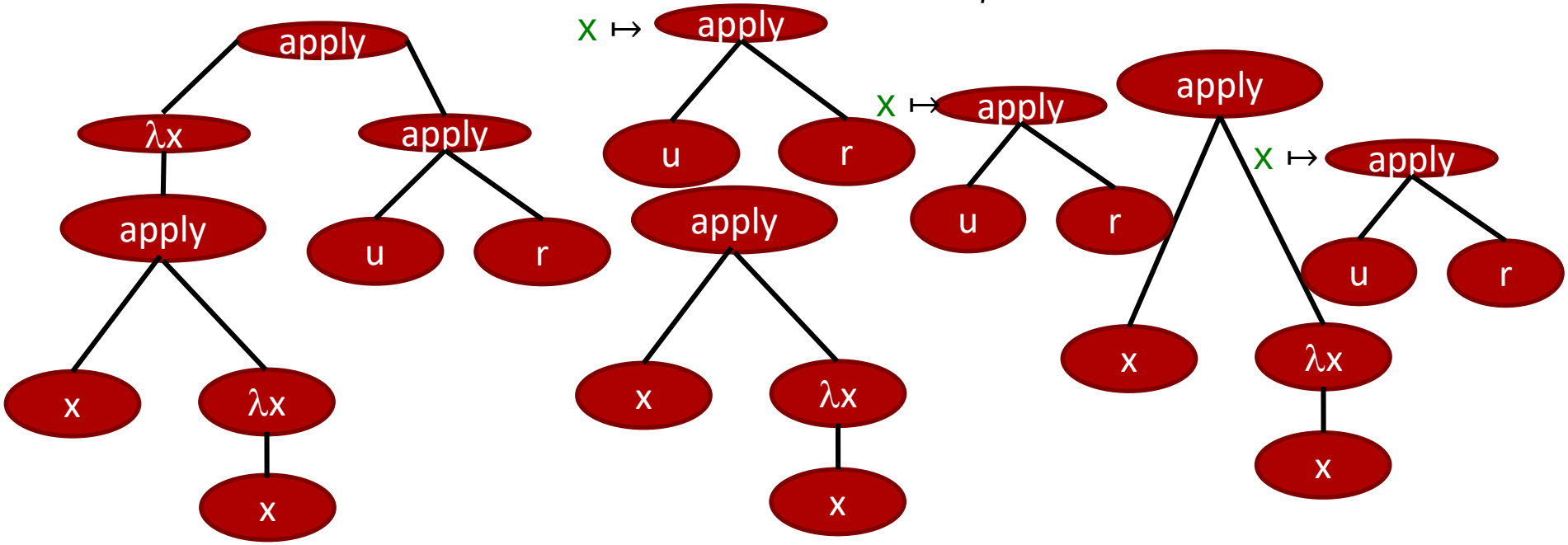


Example Beta-Reduction (ex 2)

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

redex

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

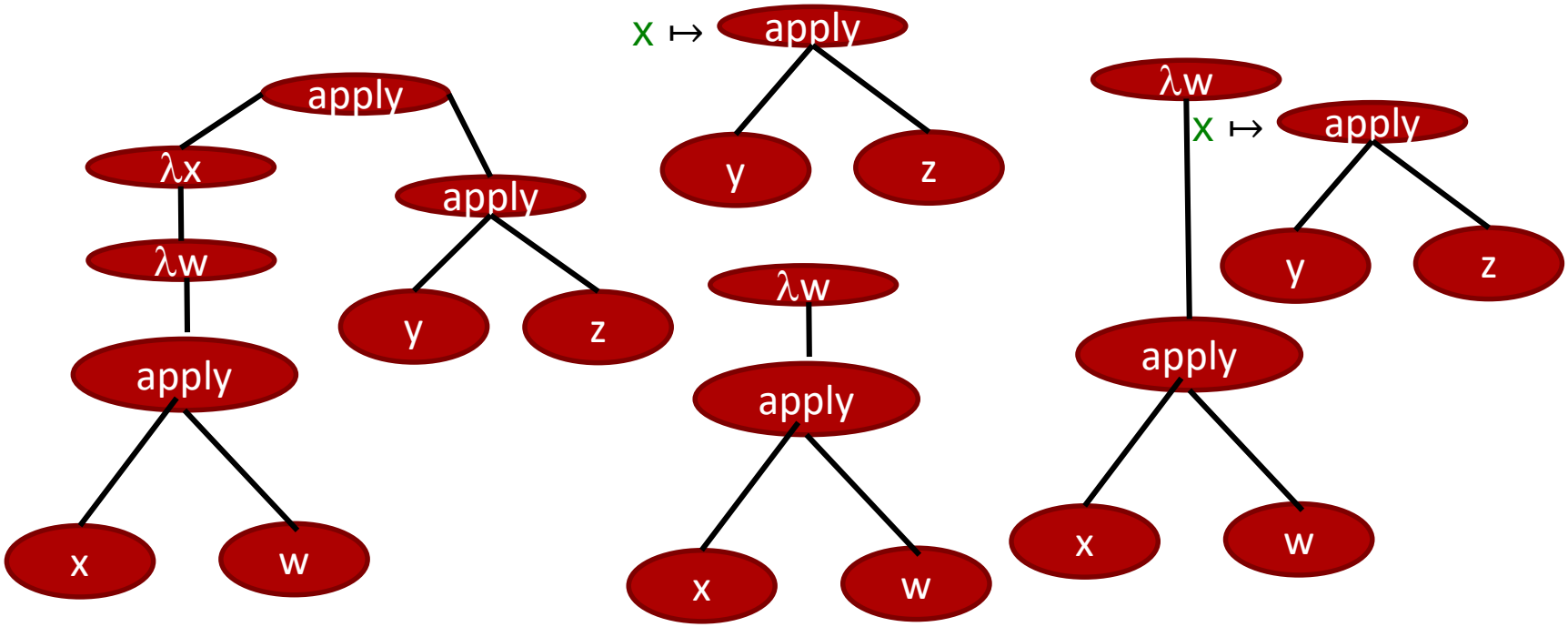


Example Beta-Reduction (ex 3)

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

redex

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



Alpha- Conversion

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

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

Simple Exercise

- Adding scopes to λ expressions
- Proposed syntax "let x = t_1 in t_2"
- Informal meaning:
  - all the occurrences of x in t2 are replaced by t1
- Example: let a = λ x. (λw. x w) in a a =
- How can we simulate let?

Another Exercise

- Simulate a function which composes two functions f : D1 → D2 and
  g: D2 →D3 in Lambda calculus

Back to Haskel

- Can be embedded in typed lambda calculus

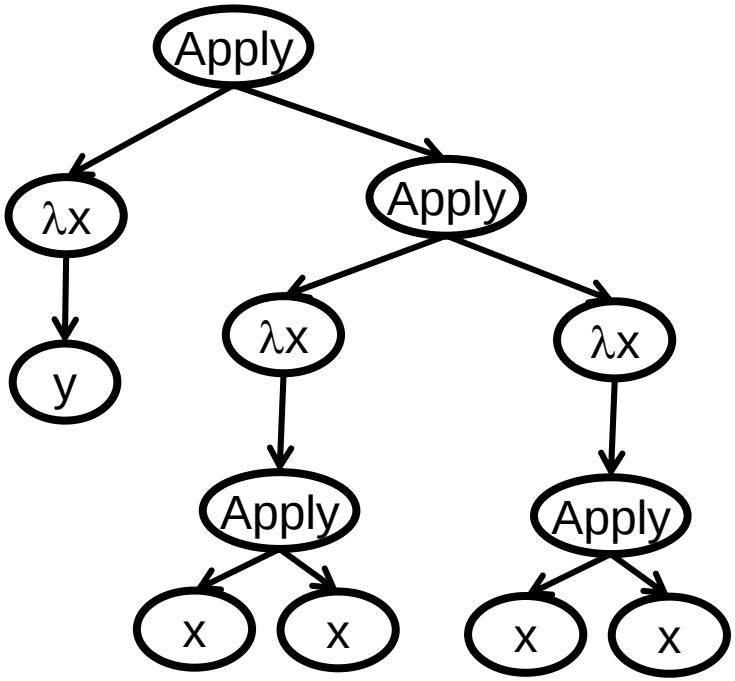
t ::=	terms
x	variable
λ x: T. t	abstraction
t t	application

T ::=	Types
Int, Bool, …	Primitive Types
α	Polymorphic Types
Id T*	Algebraic types
T → T	Function types

Divergence

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

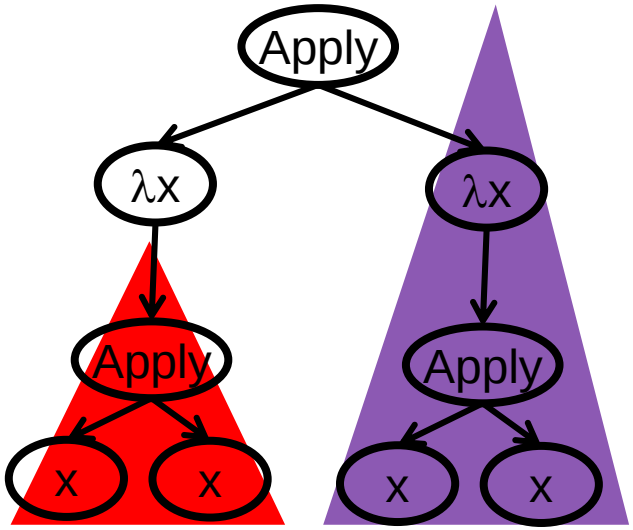
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



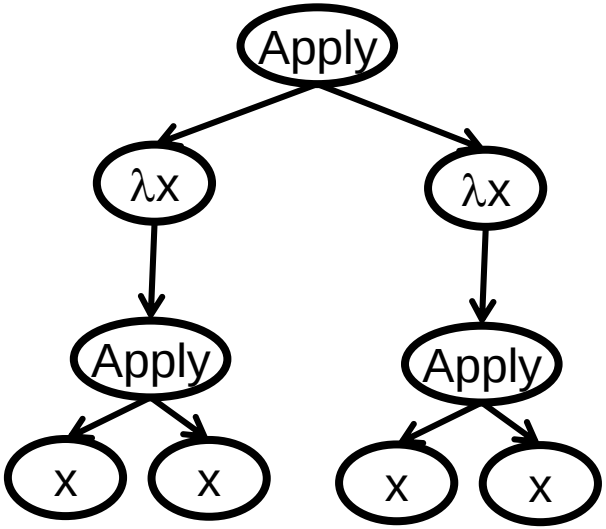
Divergence

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

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



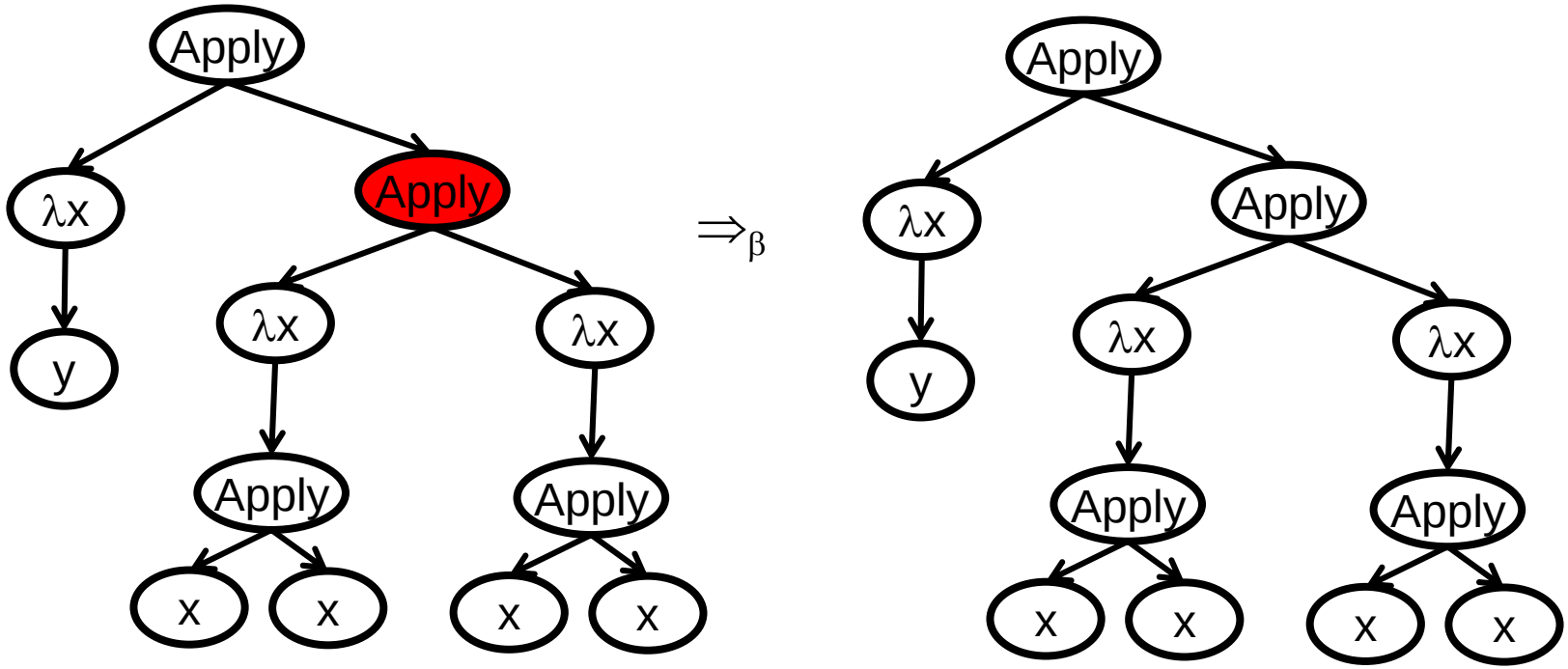
$\Rightarrow_\beta$



Different Evaluation Orders

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

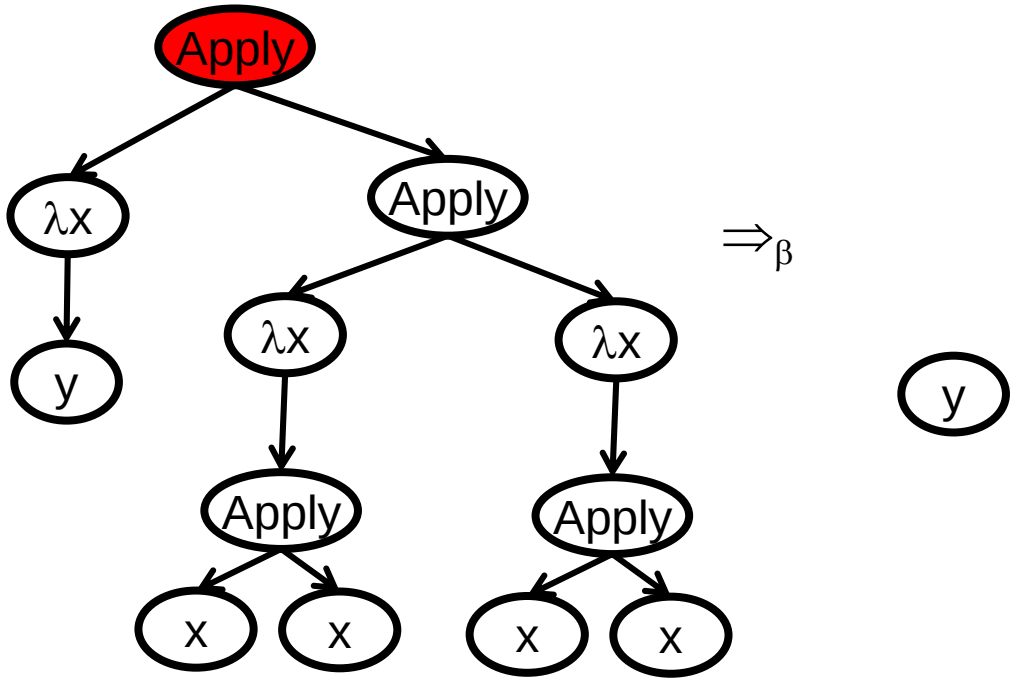
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

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

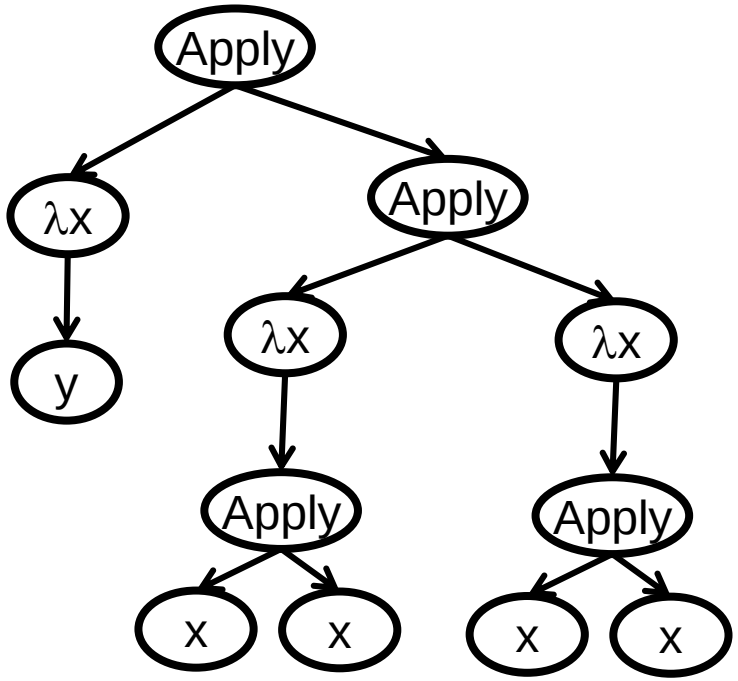
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

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

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

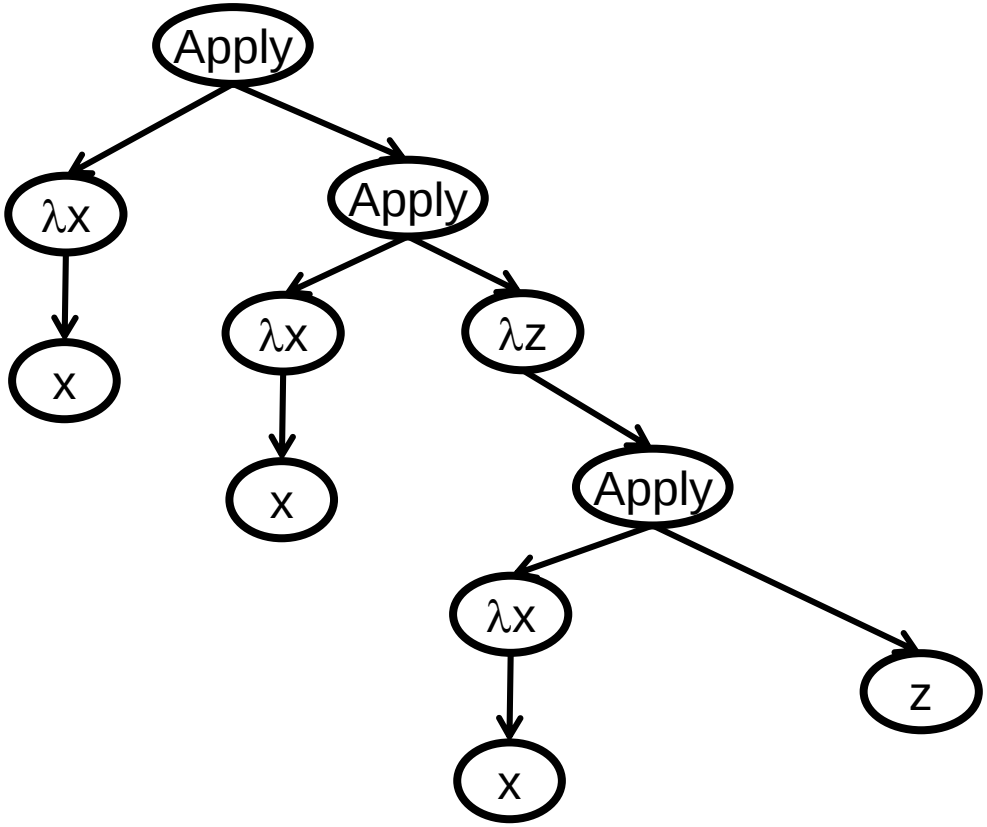



```
def f():
    while True: pass

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)

$(\lambda\ x.\ x)\ ((\lambda x.\ x)\ (\lambda z.\ (\lambda x.\ x)\ z)) \equiv id\ (id\ (\lambda z.\ id\ z))$

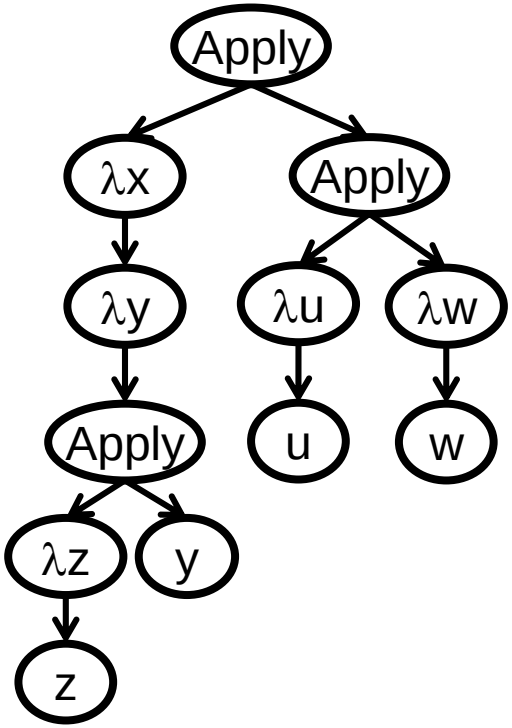


Order of Evaluation

- 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

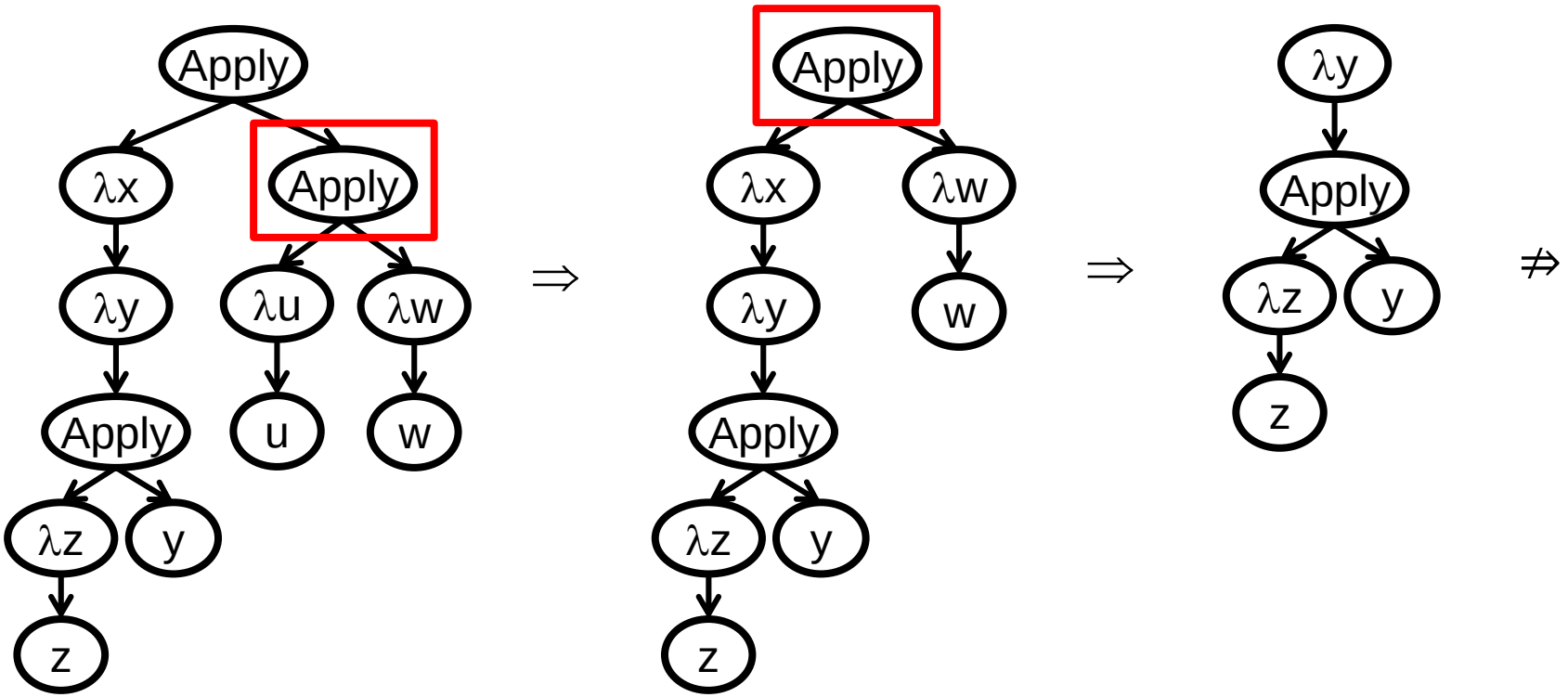
Different Evaluation Orders

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



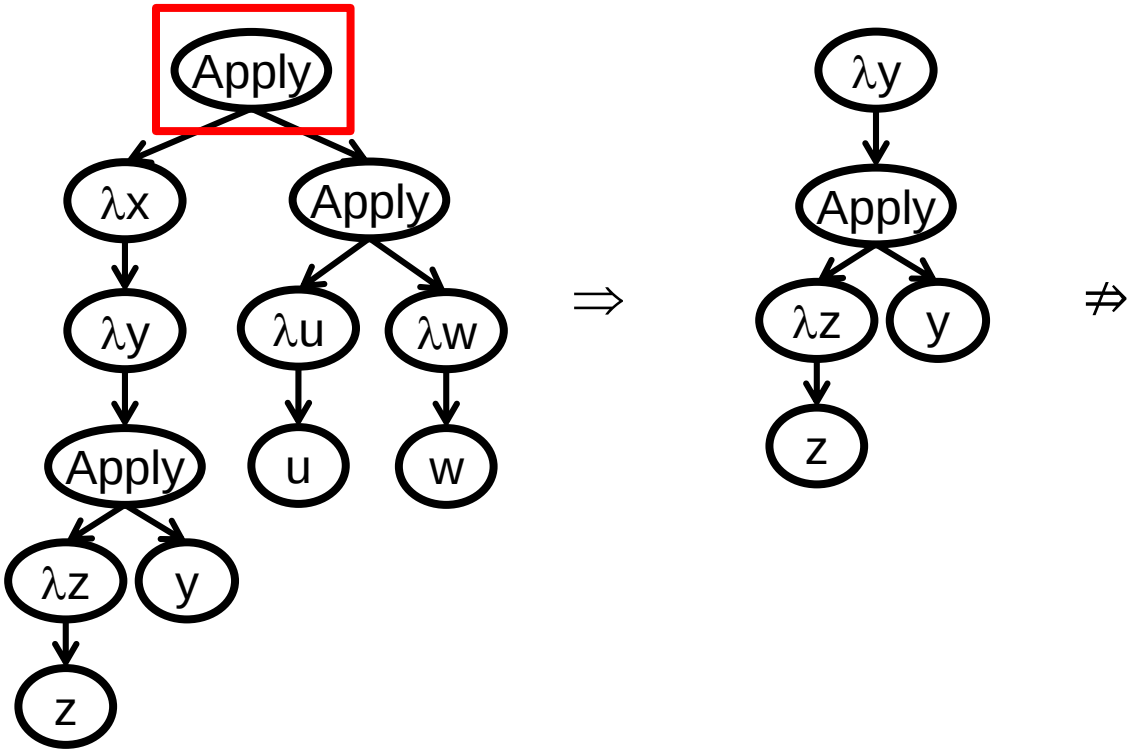
Call By Value

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



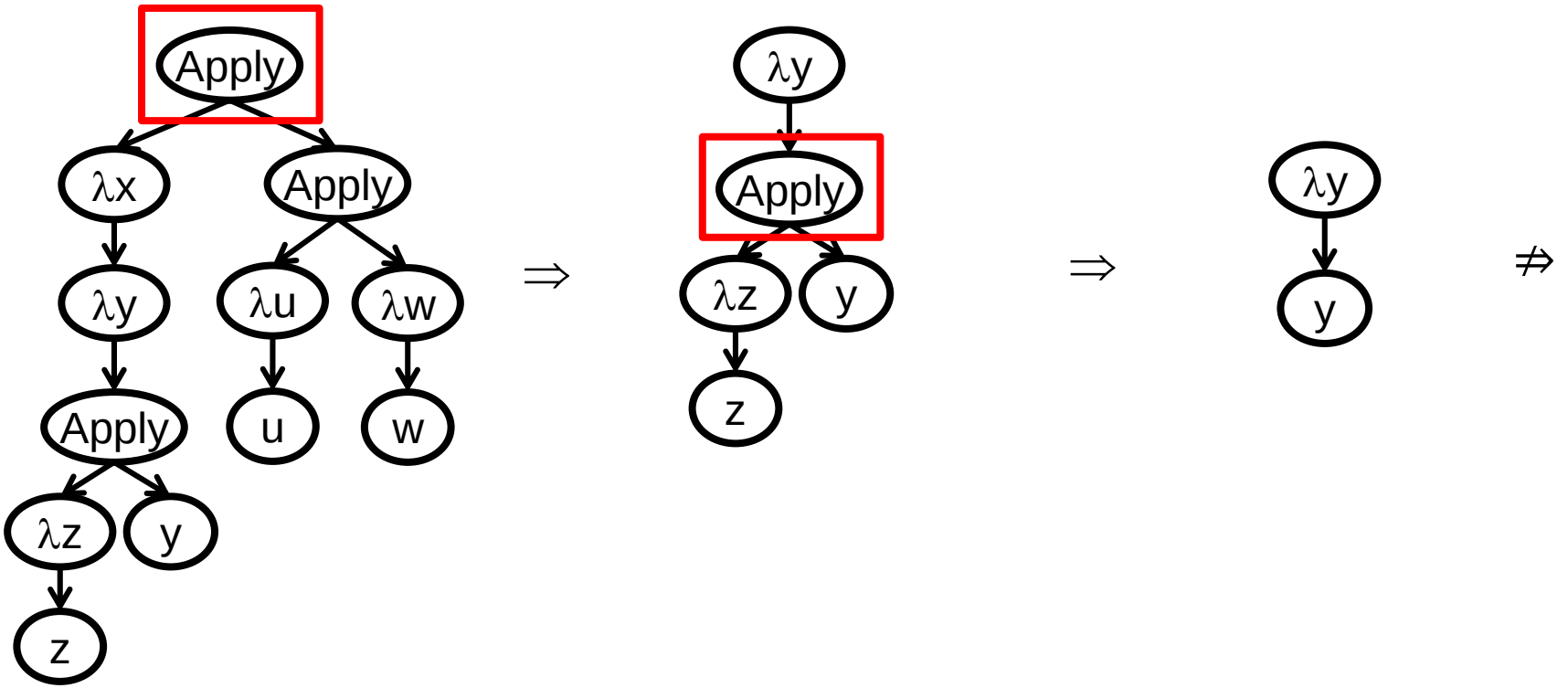
Call By Name (Lazy)

(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	

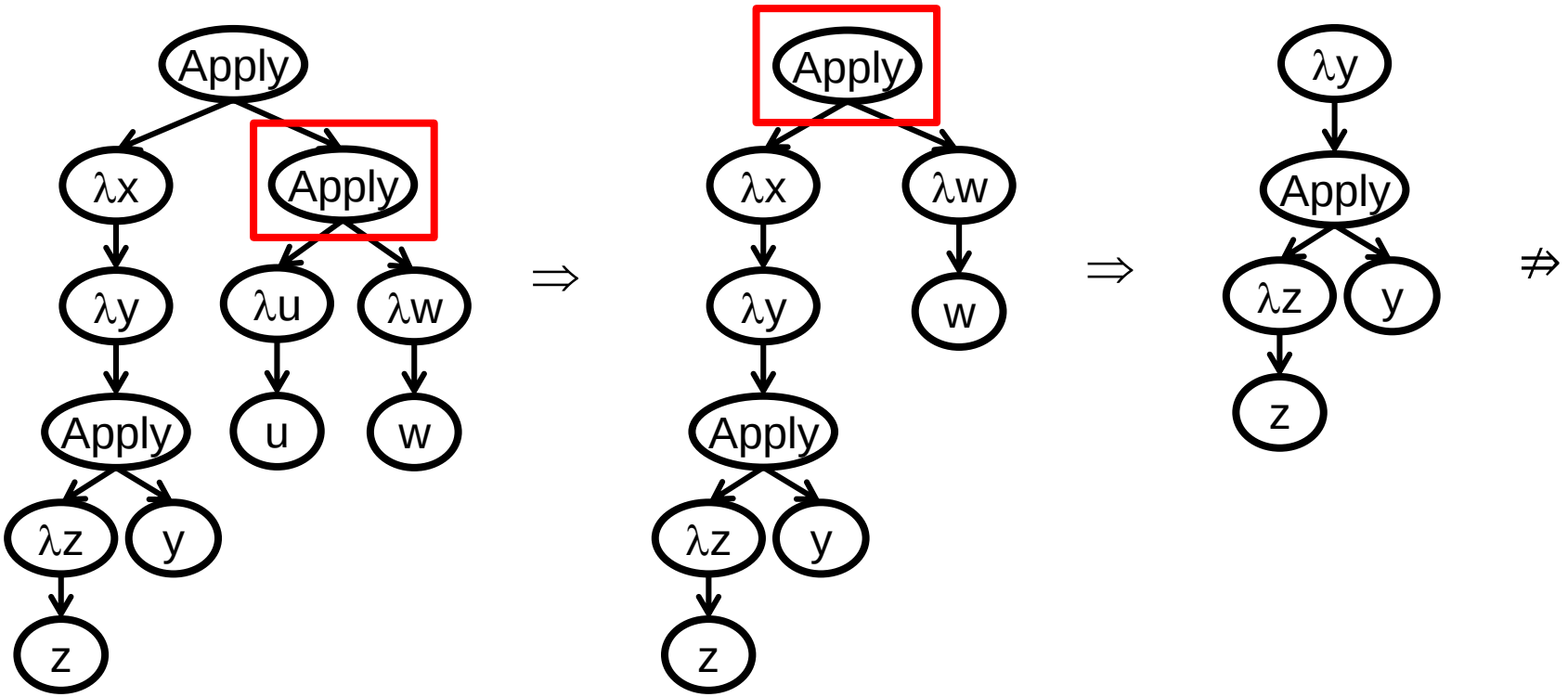
$$(\lambda\, x.\, t_1)\; v_2 \Rightarrow [x \mapsto v_2]\; t_1 \qquad \text{(E-AppAbs)}$$

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

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

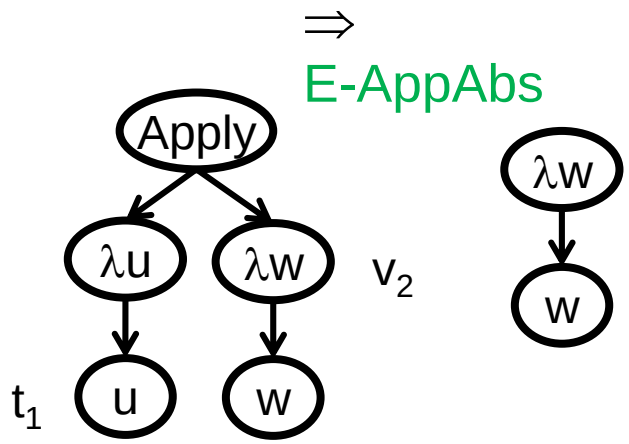
Call By Value

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



Call By Value OS

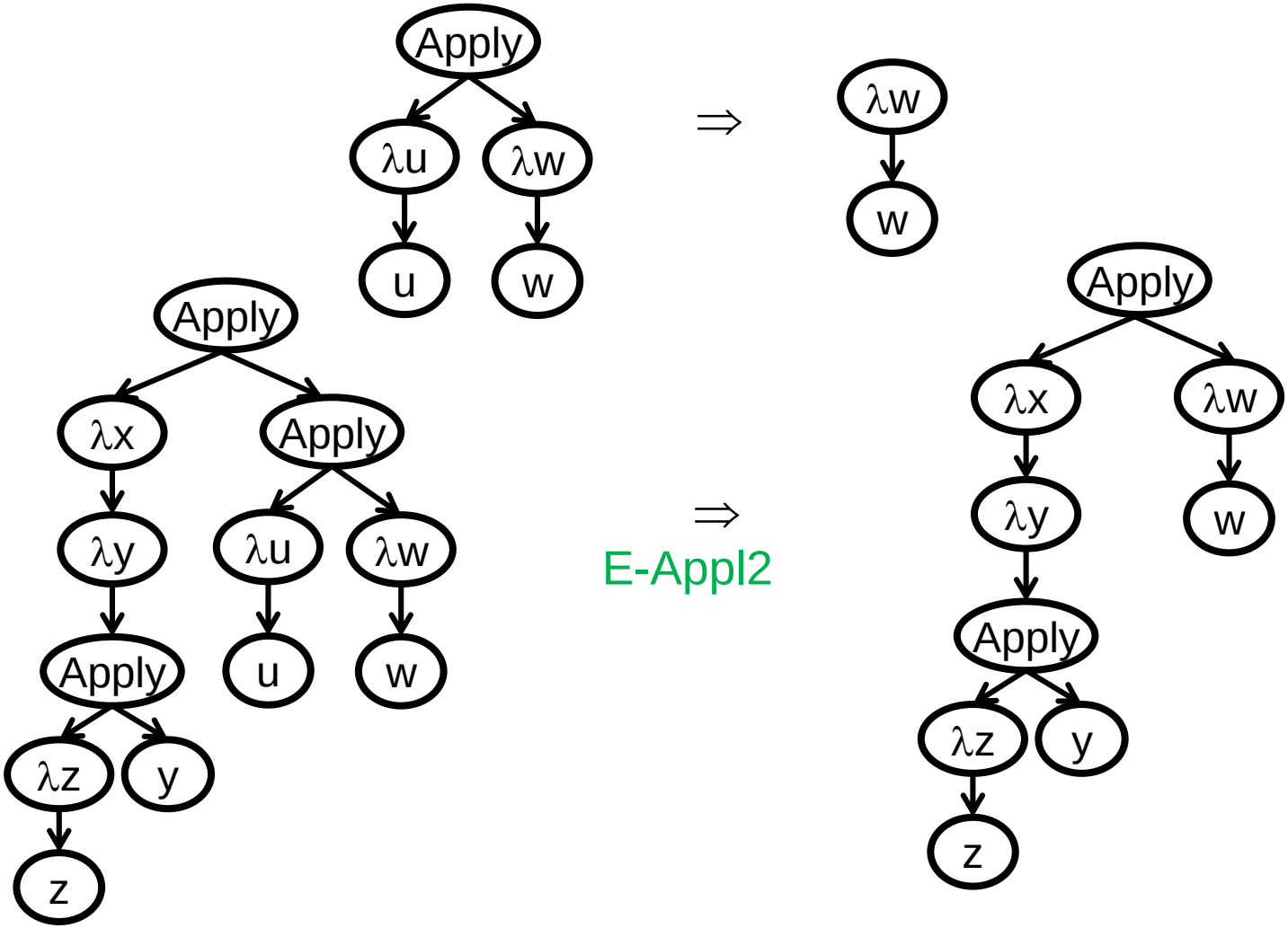
$(\lambda\ x.\ \lambda\ y.\ (\lambda\ z.\ z)\ y)\ \underline{((\lambda\ u.\ u)\ (\lambda\ w.\ w))}$



Call By Value OS (2)

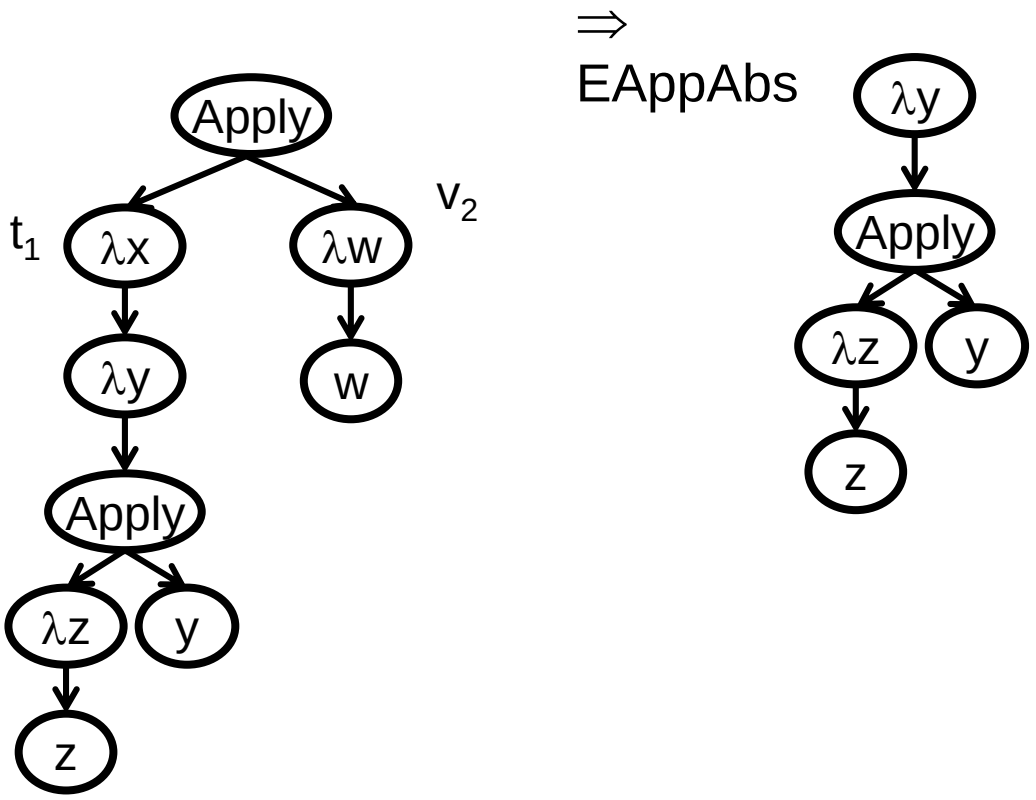
v_1 t_2

$(\lambda\, x.\, \lambda\, y.\, (\lambda\, z.\, z)\, y)\, ((\lambda\, u.\, u)\, (\lambda\, w.\, w))$



Call By Value OS (3)

$(\lambda\ x.\ \lambda\ y.\ (\lambda\ z.\ z)\ y)\ ((\lambda\ u.\ u)\ (\lambda\ w.\ w))$



Lazy Evaluation

- How to enforce lazy order of beta reductions?

Lazy operational semantics

$$\lambda x.e \rightarrow^{l} \lambda x.e \qquad \text{(L-Reflexive)}$$

$$\frac{e_1 \rightarrow^{l} \lambda x.e \qquad [x \mapsto e_2]e \rightarrow^{l} e'}{e_1\ e_2 \rightarrow^{l} e'} \qquad \text{(L-Apply)}$$

$(\lambda\ x.\ \lambda\ y.\ (\lambda\ z.\ z)\ y)\ ((\lambda\ u.\ u)\ (\lambda\ w.\ w))$

Eager operational semantics

$$\lambda x.e \rightarrow^{eg} \lambda x.e \qquad \text{(E-Reflexive)}$$

$$\frac{e_1 \rightarrow^{eg} \lambda x.e \quad e_2 \rightarrow^{eg} e' \quad [x \mapsto e']e \rightarrow^{eg} e''}{e_1\ e_2 \rightarrow^{eg} e''} \qquad \text{(E-Apply)}$$

$(\lambda\ x.\ \lambda\ y.\ (\lambda\ z.\ z)\ y)\ ((\lambda\ u.\ u)\ (\lambda\ w.\ w))$

Lazy vs eager in practice

- Lazy evaluation (call by name, call by need)
  - Has some nice theoretical properties
  - Terminates more often
  - Lets you play some tricks with “infinite” objects
  - Domain specific languages
  - Main example: Haskell
  - Support in Scala
- Eager evaluation (call by value)
  - Is generally easier to implement efficiently
  - Blends more easily with side-effects
  - Main examples: Most languages (C, Java, ML, . . .)

Continuation

Two Haskel Versions of Factorial

```
fac1 n = if n = 0 then 1 else n * fac n -1
```

```
fac2 n =
    let aux n acc = if n = 1 then acc else aux n-1 n * acc
in
   aux n 1
```

- Which version is more efficient?
- How does one go from fac1 to fac2

Continuations

- Idea:
  - The continuation of an expression is “the remaining work to be done after evaluating the expression”
  - Continuation of *e* is a function normally applied to *e*
- General programming technique
  - Capture the continuation at some point in a program
  - Use it later: “jump” or “exit” by function call
- Useful in
  - Denotational Semantics
  - Compiler optimization: make control flow explicit
  - Operating system scheduling, multiprogramming
  - Web site design, other applications

Example of Continuation Concept

- Expression
  - 2*x + 3*y + 1/x + 2/y
- What is continuation of 1/x?
  - Remaining computation after division

```
var before = 2*x + 3*y;
function cont(d) {return (before + d + 2/y)};
cont (1/x);
```

Example of Continuation Concept

- Expression
  - 2*x + 3*y + 1/x + 2/y
- What is continuation of 1/x?
  - Remaining computation after division

```
let val before = 2*x + 3*y
    fun continue(d) = before + d + 2/y
in
    continue (1/x)
end
```

Using Continuations for Error Handling

```
fun divide (numerator, denominator, normal_cont, error_con) =
   if denominator > 0.0001
      then normal_cont(numerator/denominator)
      else error_cont()


  fun f(x, y) let
        val before = 2.0 *x + 3.0 *y
        fun continue(quote) = before + quote + 2.0/y
        fun error_continu() = before/5.2
         in
          divide(1.0, x, continue, error_continu)
         end
```

Example: Tail Recursive Factorial

- Standard recursive function

  fact(n) = if n=0 then 1 else n*fact(n-1)

- Tail recursive

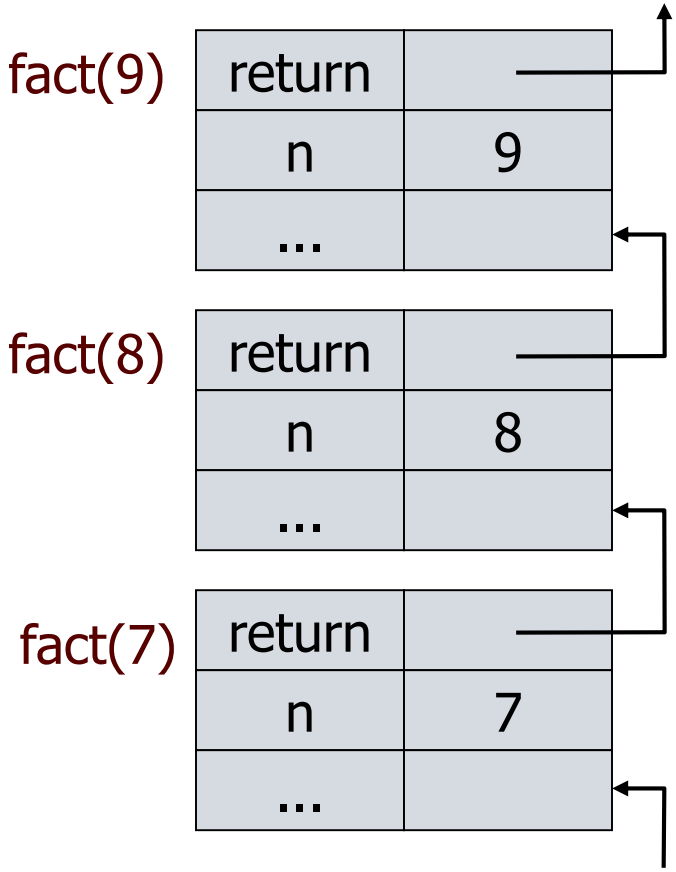
  f(n,k) = if n=0 then k else f(n-1, n*k)

  fact(n) = f(n,1)

- How could we derive this?
  - Transform to continuation-passing form (automatic)
  - Optimize continuation function to single integer

Continuation view of factorial

fact(n) = if n=0 then 1 else n*fact(n-1)



This invocation multiplies by 9 and returns
Continuation of fact(8) is
$\lambda x.\ 9*x$

Multiplies by 8 and returns
Continuation of fact(7) is
$\lambda y.\ (\lambda x.\ 9*x)\ (8*y)$

Multiplies by 7 and returns
Continuation of fact(6) is
$\lambda z.\ \ (\lambda y.\ (\lambda x.\ 9*x)\ (8*y))\ (7*z)$

continuation fact(n-1) = continuation fact(n) $\circ\ \lambda x.\ n*x$

Derivation of tail recursive form

- Standard function

  fact(n) = if n=0 then 1 else n*fact(n-1)

- Continuation form

  fact(n, k) = if n=0 then k(1)
  else fact(n-1, λx.k (n*x))

  continuation

  fact(n, λx.x) computes n!

- Example computation

  fact(3,λx.x) = fact(2, λy.((λx.x) (3*y)))

  = fact(1, λx.((λy.3*y)(2*x)))

  = λx.((λy.3*y)(2*x)) 1 = 6

Tail Recursive Form

- Optimization of continuations

  fact(n, k) = if n=0 then k(1)
  else fact(n-1, λx.k (n*x))

  ⇓

  fact(n,a) = if n=0 then a
  else fact(n-1, n*a)

  Each continuation is effectively λx.(a*x) for some a

- Example computation

  fact(3,1) = fact(2, 3) was fact(2, λy.3*y)

  = fact(1, 6) was fact(1, λx.6*x)

  = 6

Other uses for continuations

- Explicit control
  - Normal termination -- call continuation
  - Abnormal termination -- do something else
- Compilation techniques
  - Call to continuation is functional form of “go to”
  - Continuation-passing style makes control flow explicit

MacQueen: “Callcc is the closest thing to a ‘come-from’ statement I’ve ever seen.”

Continuations in Mach OS [SOSP'91]

- OS kernel schedules multiple threads
  - Each thread may have a separate stack
  - Stack of blocked thread is stored within the kernel
- Mach "continuation" approach
  - Blocked thread represented as
    - Pointer to a continuation function, list of arguments
    - Stack is discarded when thread blocks
  - Programming implications
    - Sys call such as msg_recv can block
    - Kernel code calls msg_recv with continuation passed as arg
  - Advantage/Disadvantage
    - Saves a lot of space, need to write "continuation" functions

Continuations in Web programming

- Use continuation-passing style to allow multiple returns

```
function doXHR(url, succeed, fail) {
    var xhr = new XMLHttpRequest(); // or ActiveX equivalent
    xhr.open("GET", url, true);
    xhr.send(null);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if (xhr.status == 200)
                succeed(xhr.responseText);
            else
                fail(xhr);
} }; }
```

- See http://marijn.haverbeke.nl/cps/

Continuations in compilation

- SML continuation-based compiler [Appel, Steele]
  1) Lexical analysis, parsing, type checking
  2) Translation to λ-calculus form
  3) Conversion to continuation-passing style (CPS)
  4) Optimization of CPS
  5) Closure conversion – eliminate free variables
  6) Elimination of nested scopes
  7) Register spilling – no expression with >n free vars
  8) Generation of target assembly language program
  9) Assembly to produce target-machine program

Theme Song: Charlie on the MTA

- Let me tell you the story
  Of a man named Charlie
  On a tragic and fateful day
  He put ten cents in his pocket,
  Kissed his wife and family
  Went to ride on the MTA
- Charlie handed in his dime
  At the Kendall Square Station
  And he changed for Jamaica Plain
  When he got there the conductor told him,
  "One more nickel."
  Charlie could not get off that train.
- Chorus:

  Did he ever return,
  No he never returned
  And his fate is still unlearn'd
  He may ride forever
  'neath the streets of Boston
  He's the man who never returned.

Summary

- Functional programming is a powerful concept
- Continuation enables control of executions
- Other useful concepts are Monads

Summary

- Structured Programming
  - Go to considered harmful
- Exceptions
  - “structured” jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
  - Used in Lisp/Scheme compilation, some OS projects, web application development, …