

# Lambda Calculus

Oded Padon & Mooly Sagiv

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

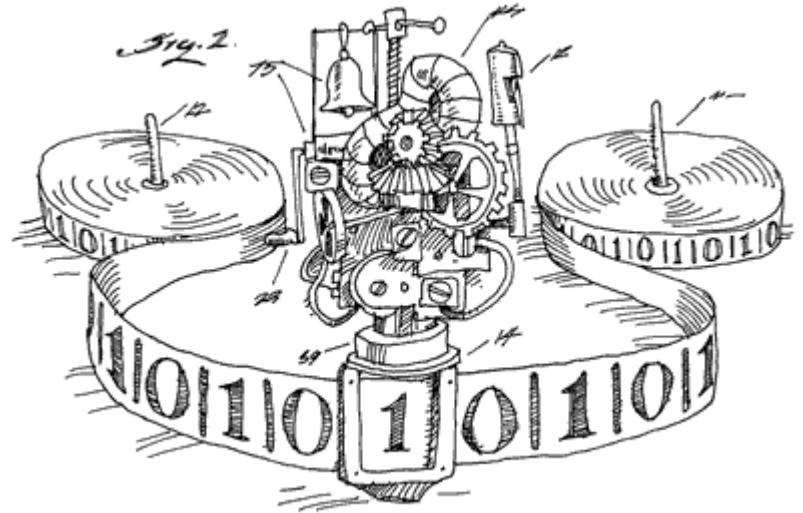
Benjamin Pierce

Types and Programming Languages

<http://www.cs.cornell.edu/courses/cs3110/2008fa/recitations/rec26.html>

# 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](http://en.wikipedia.org/wiki/Lambda_calculus)

# What is $\lambda$ 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:
  - $(5 * 4 * 3 * 2 * 1) + (7 * 6 * 5 * 4 * 3 * 2 * 1) =$
  - $\text{factorial}(5) + \text{factorial}(7)$
  - $\text{factorial}(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n * \text{factorial}(n-1)$
  - $\text{factorial} = \lambda n. \text{if } n = 0 \text{ then } 0 \text{ else } n * \text{factorial}(n-1)$
  - $\text{factorial} = \lambda n. \text{if } n = 0 \text{ then } 0 \text{ else } n * \text{apply}(\text{factorial}, (n-1))$

# Untyped Lambda Calculus

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

Terms can be represented as abstract syntax trees

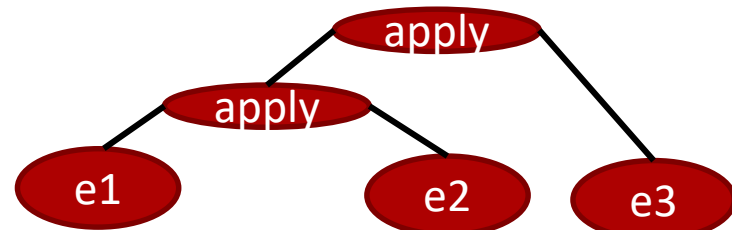
## Syntactic Conventions

- 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

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



# Untyped Lambda Calculus

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

Terms can be represented as abstract syntax trees

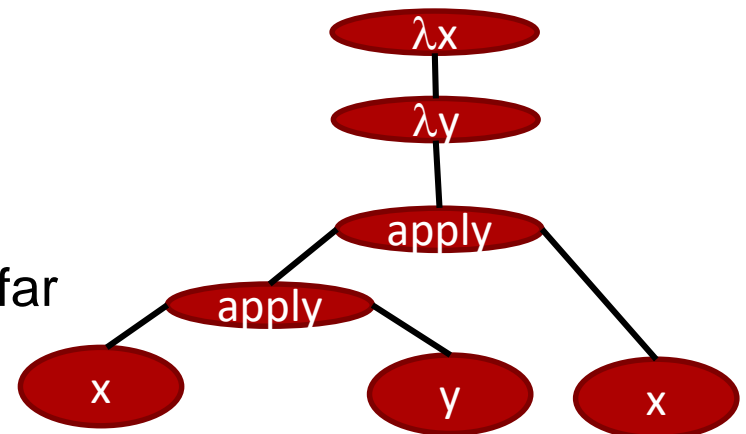
## Syntactic Conventions

- 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

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



# Example Lambda Expressions

- $\lambda x. "1"$
- $\lambda x. x$
- $\lambda x. y$
- $\lambda x. s\ x$
- $\lambda x. s\ (s\ x)$
- $\lambda f. \lambda g. f\ g$
- $\lambda b. "if" b "fls" "tru"$
- $\lambda b. \lambda b'. "if" b b' "fls"$



# 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**
  - $\lambda x$  is a **binder**
- **Examples**
  - $\text{Id} = \lambda x. x$
  - $\lambda y. x (y z)$
  - $\lambda z. \lambda x. \lambda y. x (y z)$
  - $(\lambda x. x) x$

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

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

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

$$\text{FV}(t_1 t_2) = \text{FV}(t_1) \cup \text{FV}(t_2)$$

# Beta-Reduction

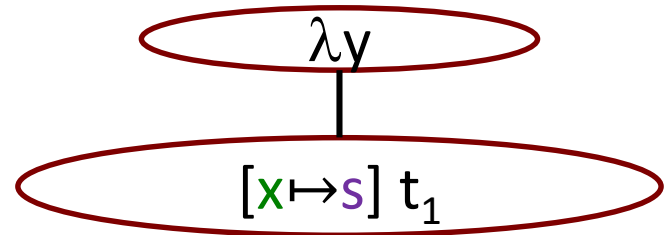
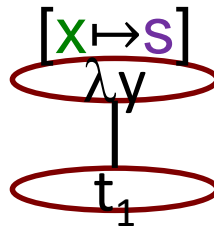
$$(\lambda x. t_1) t_2 \Rightarrow_{\beta} [x \mapsto t_2] t_1 \quad (\beta\text{-reduction})$$

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

$$[x \mapsto s] y = y \quad \text{if } y \neq x$$

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

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



# Beta-Reduction

$$(\lambda x. t_1) t_2 \Rightarrow_{\beta} [x \mapsto t_2] t_1 \quad (\beta\text{-reduction})$$

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

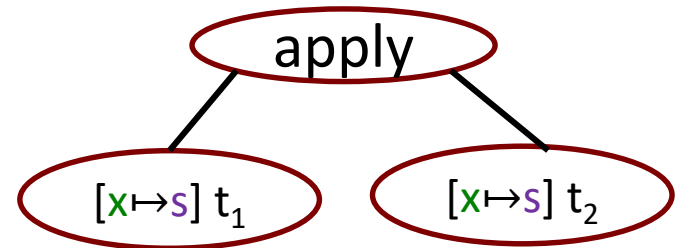
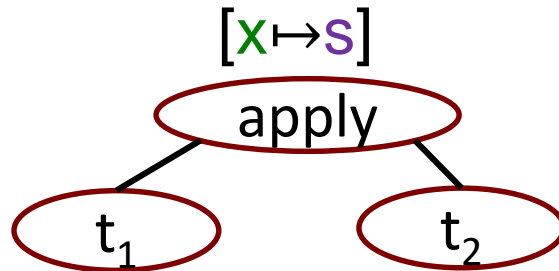
$$[x \mapsto s] y = y$$

if  $y \neq x$

$$[x \mapsto s] (\lambda y. t_1) = \lambda y. [x \mapsto s] t_1$$

if  $y \neq x$  and  $y \notin \text{FV}(s)$

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

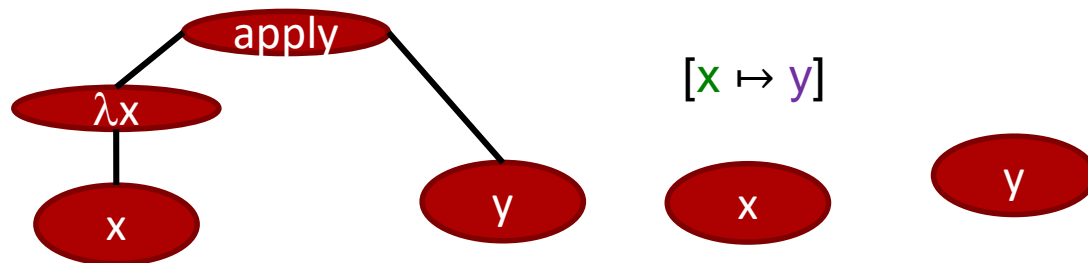


# Example Beta-Reduction

$$(\lambda x. t_1) t_2 \Rightarrow_{\beta} [x \mapsto t_2] t_1 \quad (\beta\text{-reduction})$$

redex

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



# Example Beta-Reduction (ex 2)

$$(\lambda x. t_1) t_2 \Rightarrow_{\beta} [x \mapsto t_2] t_1 \quad (\beta\text{-reduction})$$

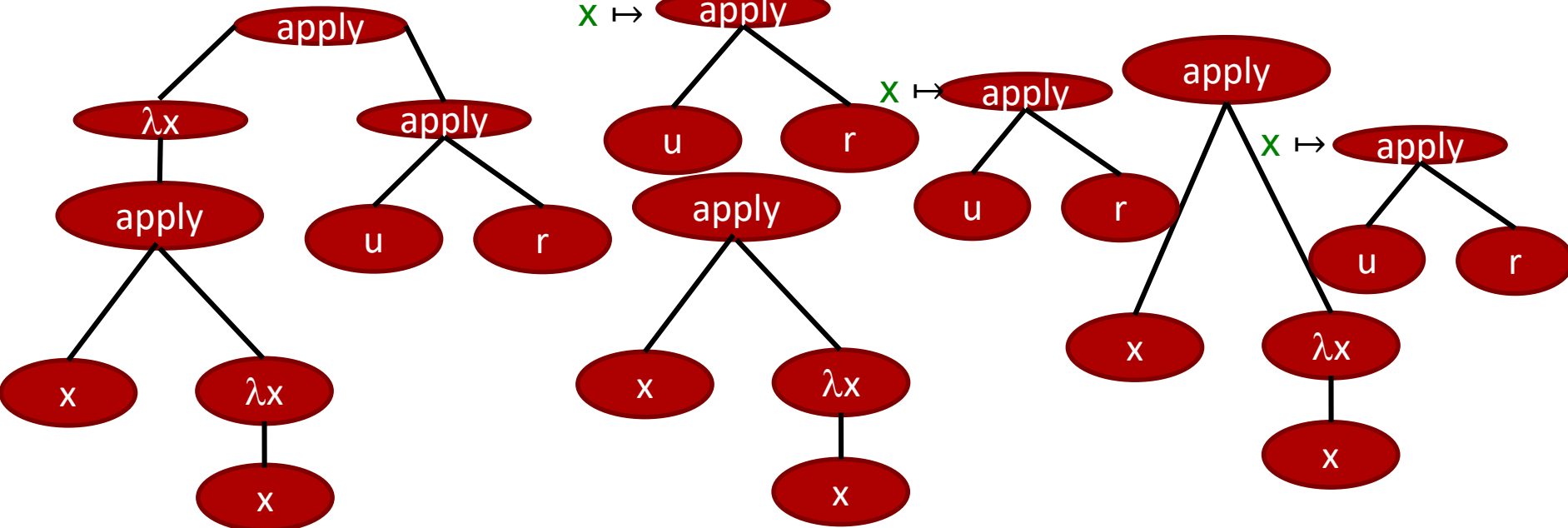
redex

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

$x \mapsto$  apply

$x \mapsto$  apply

$x \mapsto$  apply

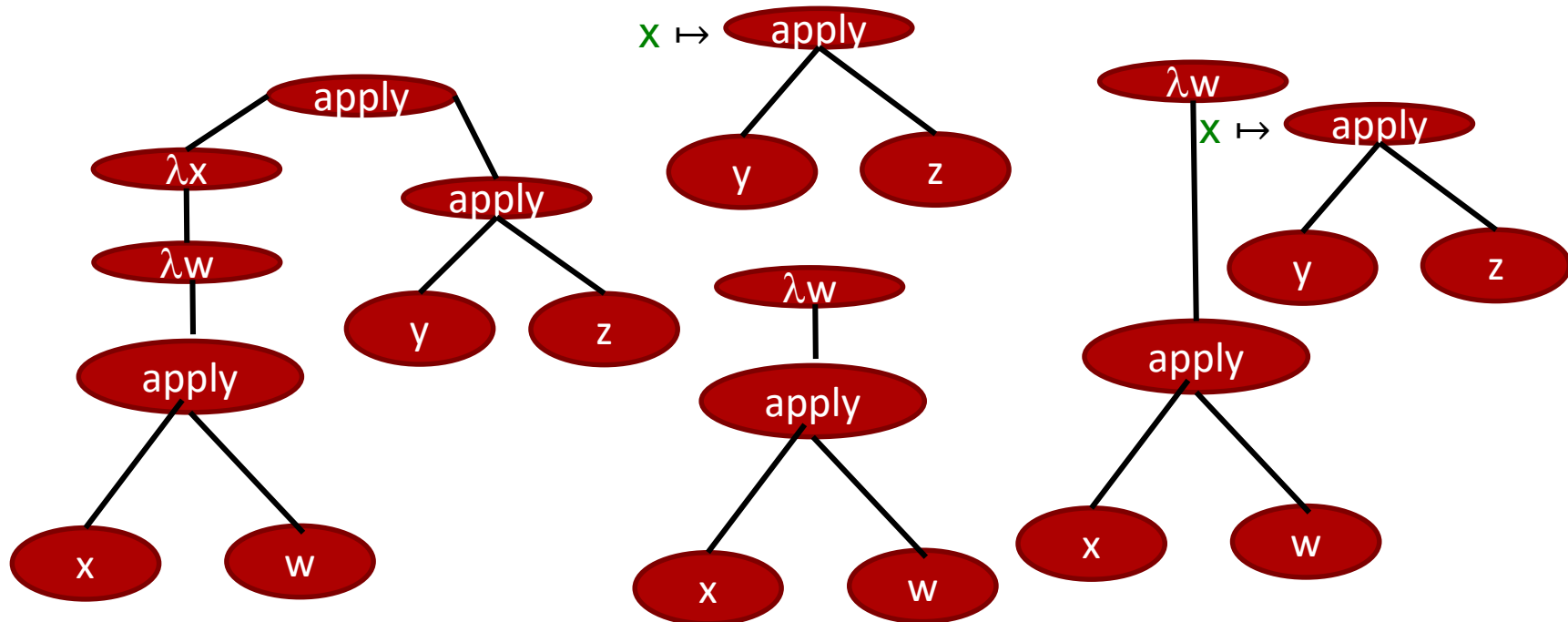


# Example Beta-Reduction (ex 3)

$$(\lambda x. t_1) t_2 \Rightarrow_{\beta} [x \mapsto t_2] t_1 \quad (\beta\text{-reduction})$$

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

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

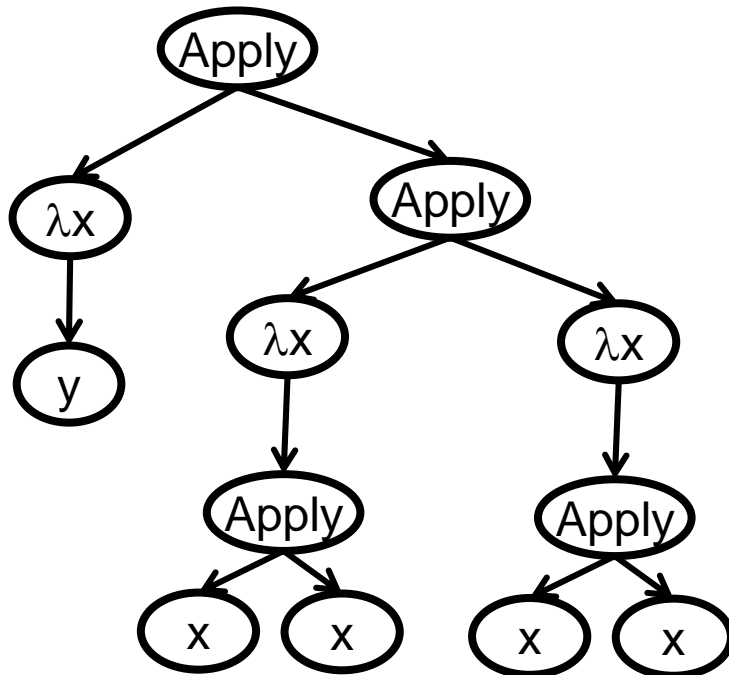
# Simple Exercise

- Adding scopes to  $\lambda$  expressions
- Proposed syntax “**let**  $x = t_1$  **in**  $t_2$ ”
- Informal meaning:
  - all the occurrences of  $x$  in  $t_2$  are replaced by  $t_1$
- Example: let  $a = \lambda x. (\lambda w. x w)$  in  $a a =$
- How can we simulate **let**?

# Divergence

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

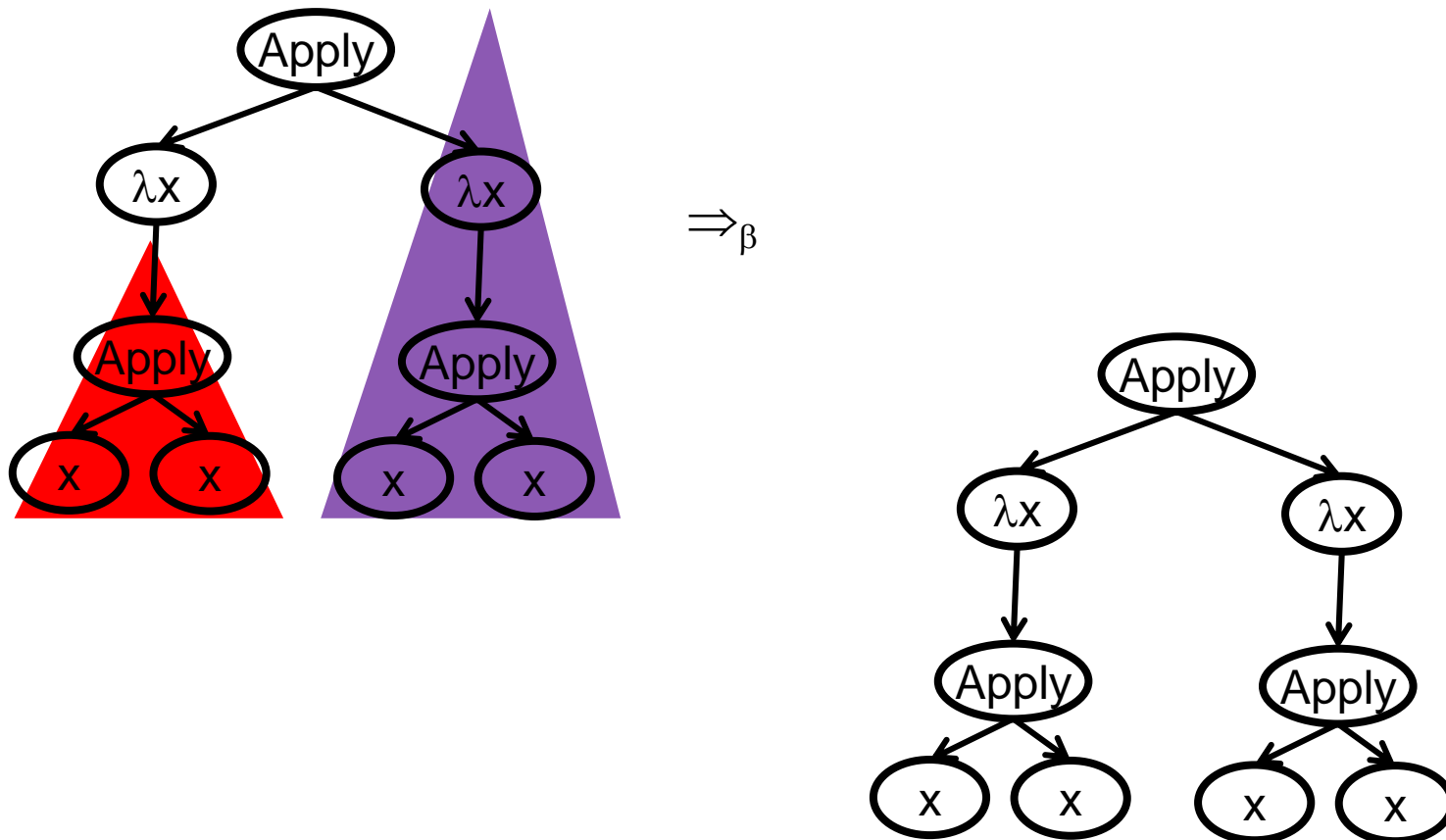
$(\lambda x. y) ((\lambda x. (x x)) (\lambda x. (x x)))$



# Divergence

$$(\lambda x. t_1) t_2 \Rightarrow_{\beta} [x \mapsto t_2] t_1 \quad (\beta\text{-reduction})$$

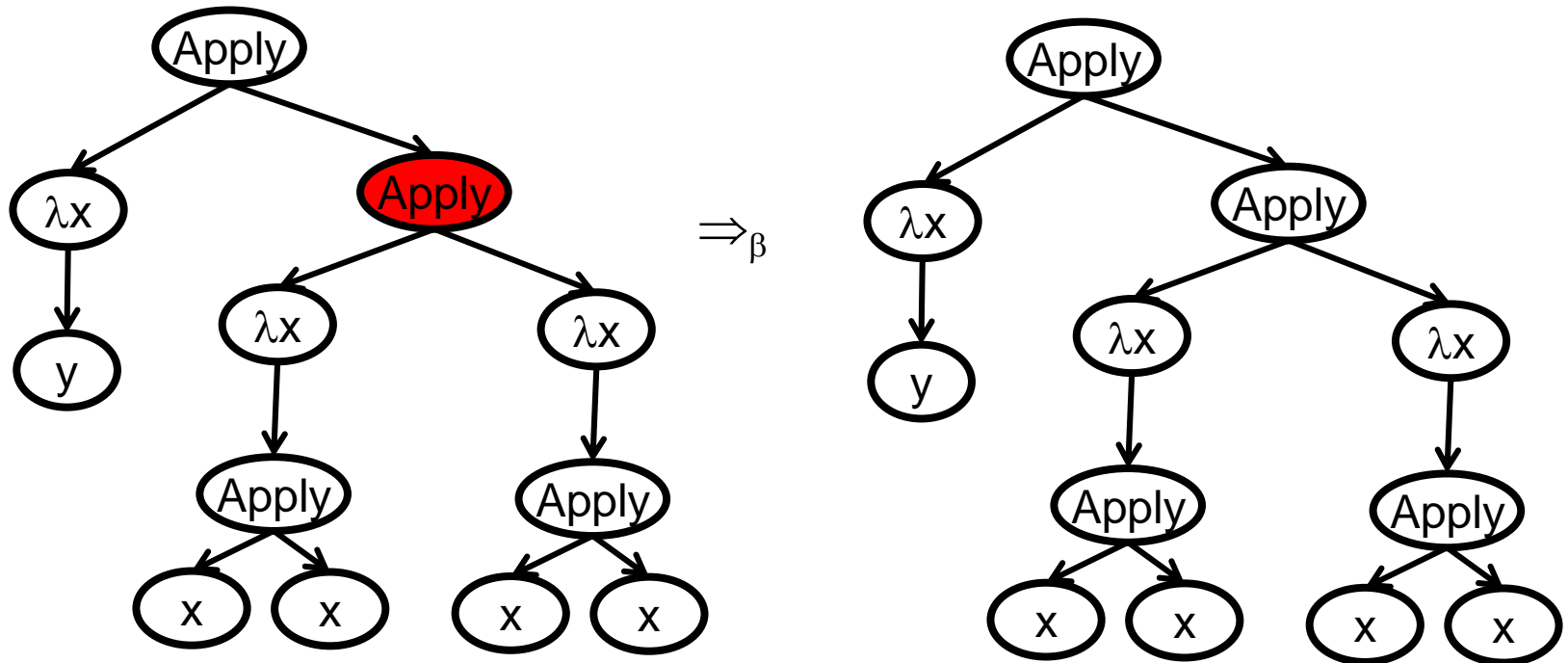
$$(\lambda x.(x x)) (\lambda x.(x x))$$



# Different Evaluation Orders

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

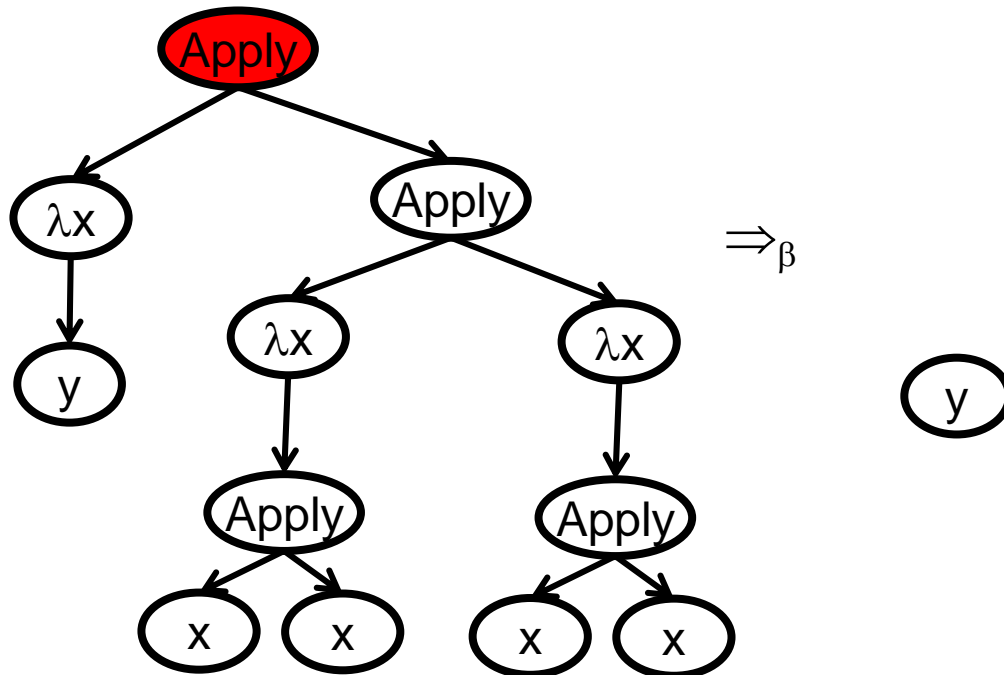
$(\lambda x.y) ((\lambda x.(x x)) (\lambda x.(x x)))$



# Different Evaluation Orders

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

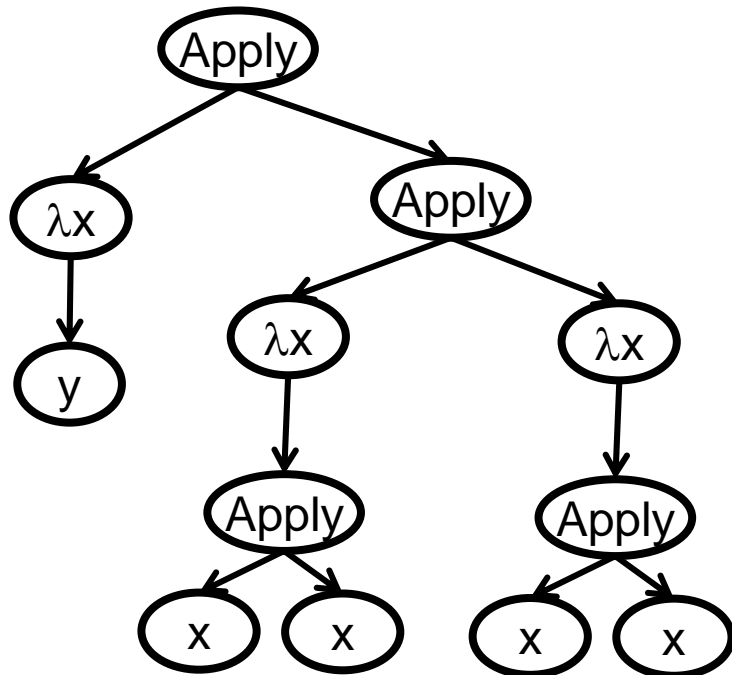
$(\lambda x.y) ((\lambda x.(x x)) (\lambda x.(x x)))$



# Different Evaluation Orders

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

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



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

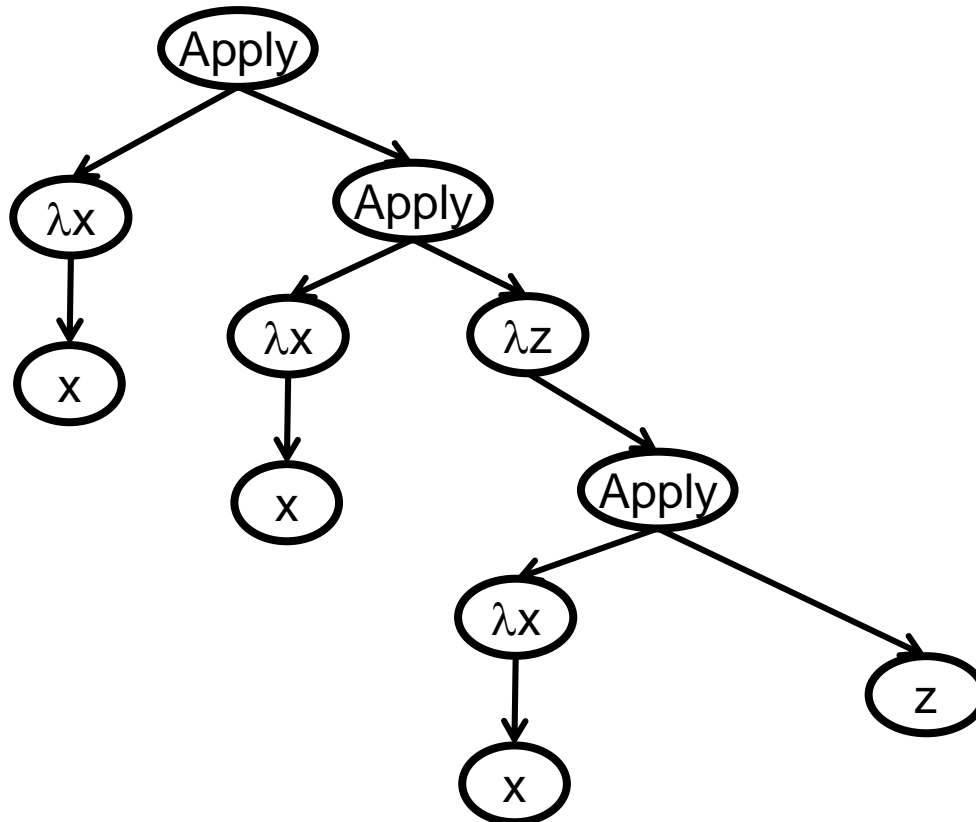
```
def g(x):  
    return 2
```

```
print g(f())
```

# Different Evaluation Orders

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

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



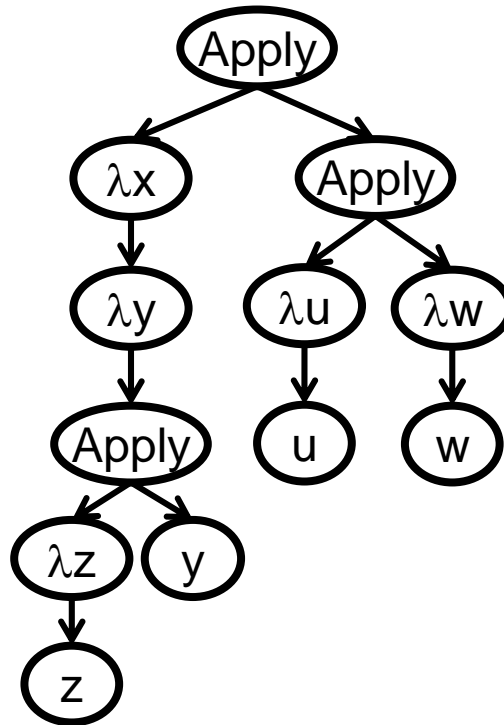


# Order of Evaluation

- 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 Haskell

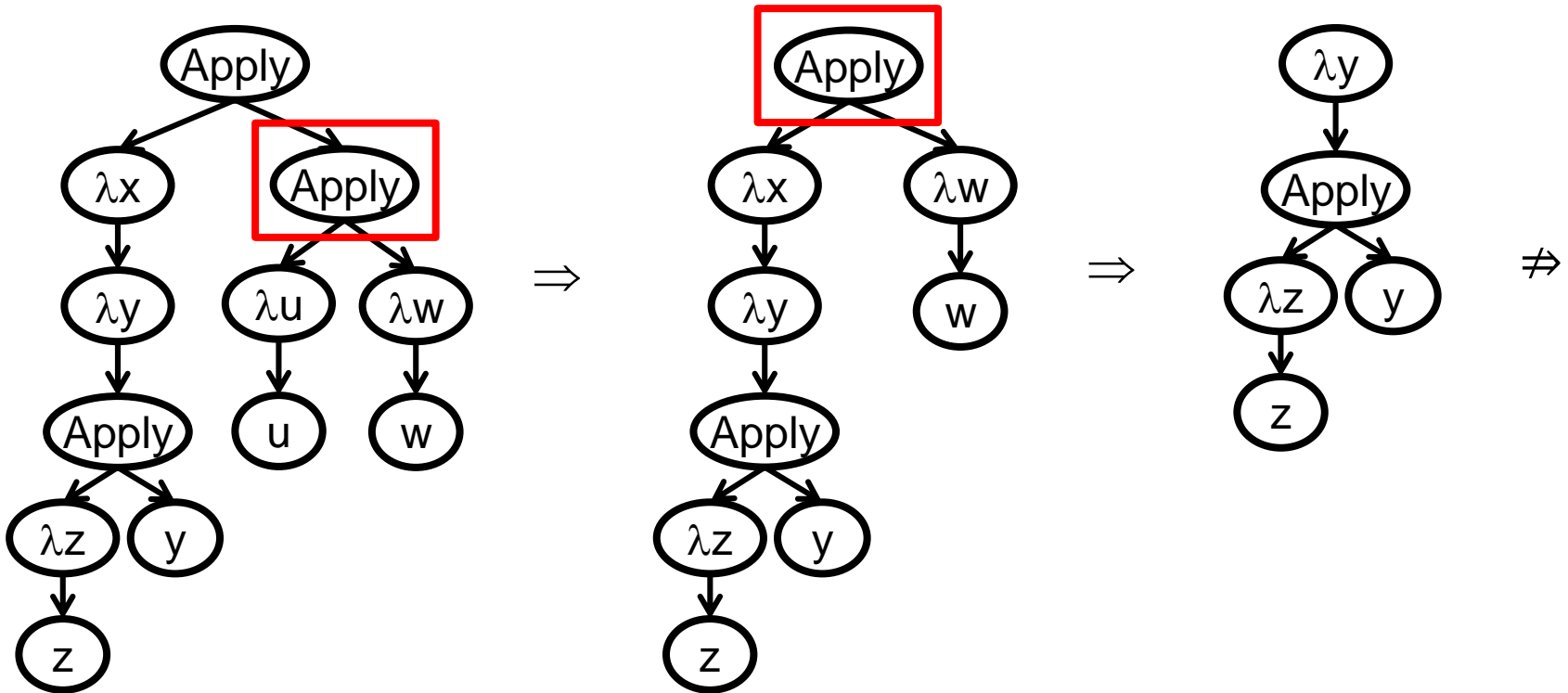
# Different Evaluation Orders

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



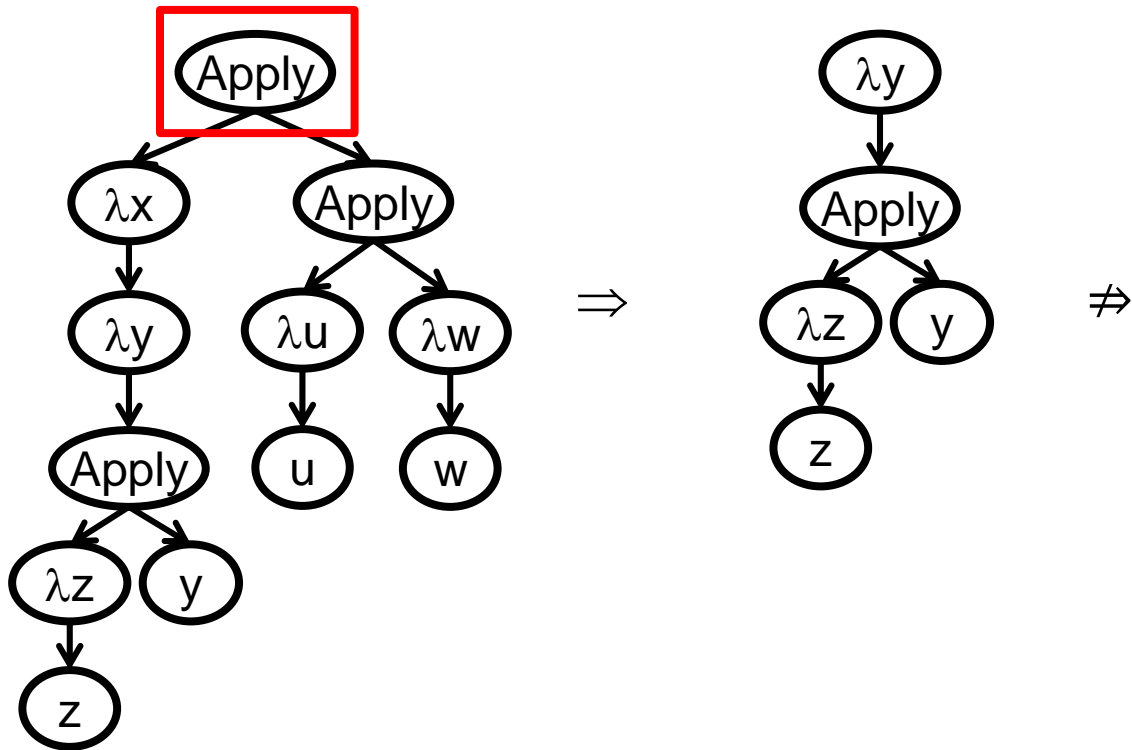
# Call By Value

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



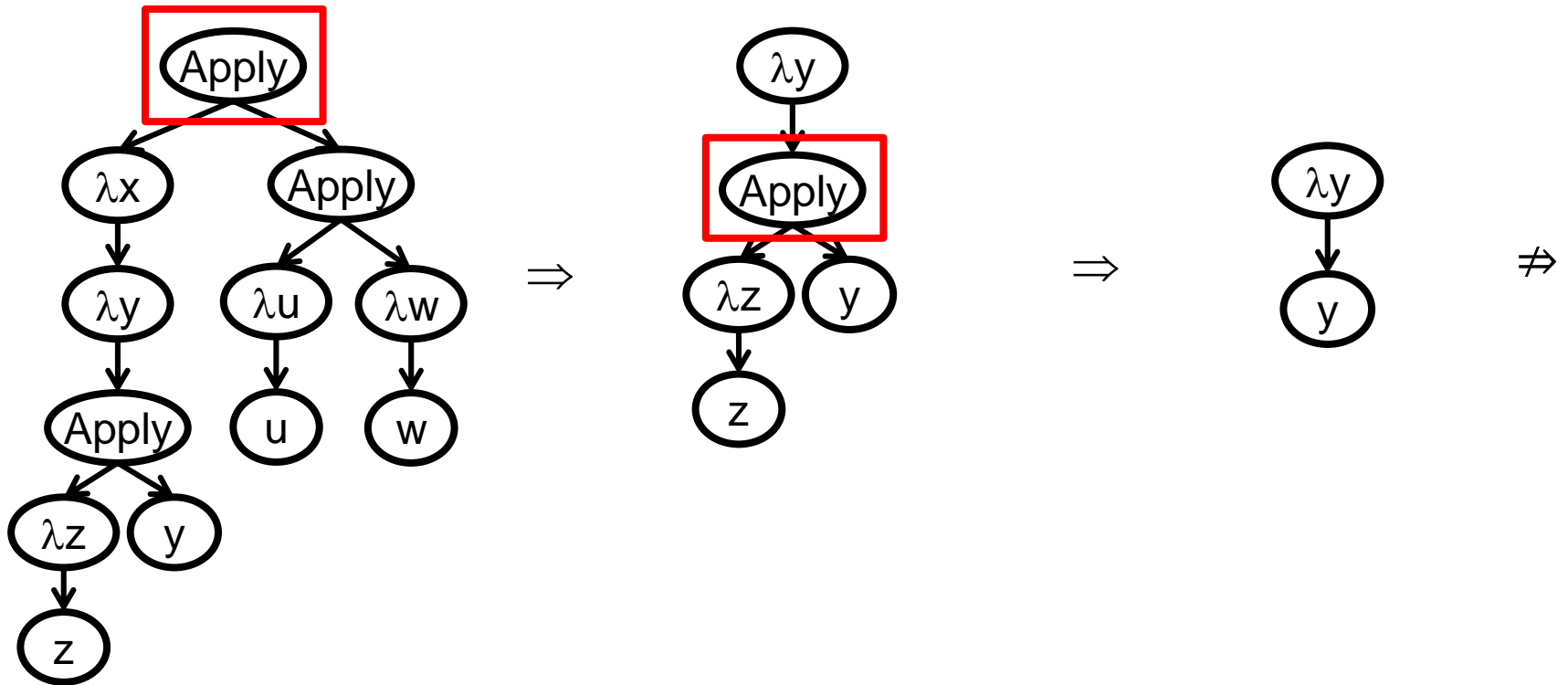
# Call By Name (Lazy)

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



# Normal Order

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



# Call-by-value Operational Semantics

|                |             |                |                    |
|----------------|-------------|----------------|--------------------|
| $t ::=$        | terms       | $v ::=$        | values             |
| $x$            | variable    | $\lambda x. t$ | abstraction values |
| $\lambda x. t$ | abstraction |                | other values       |
| $t t$          | application |                |                    |

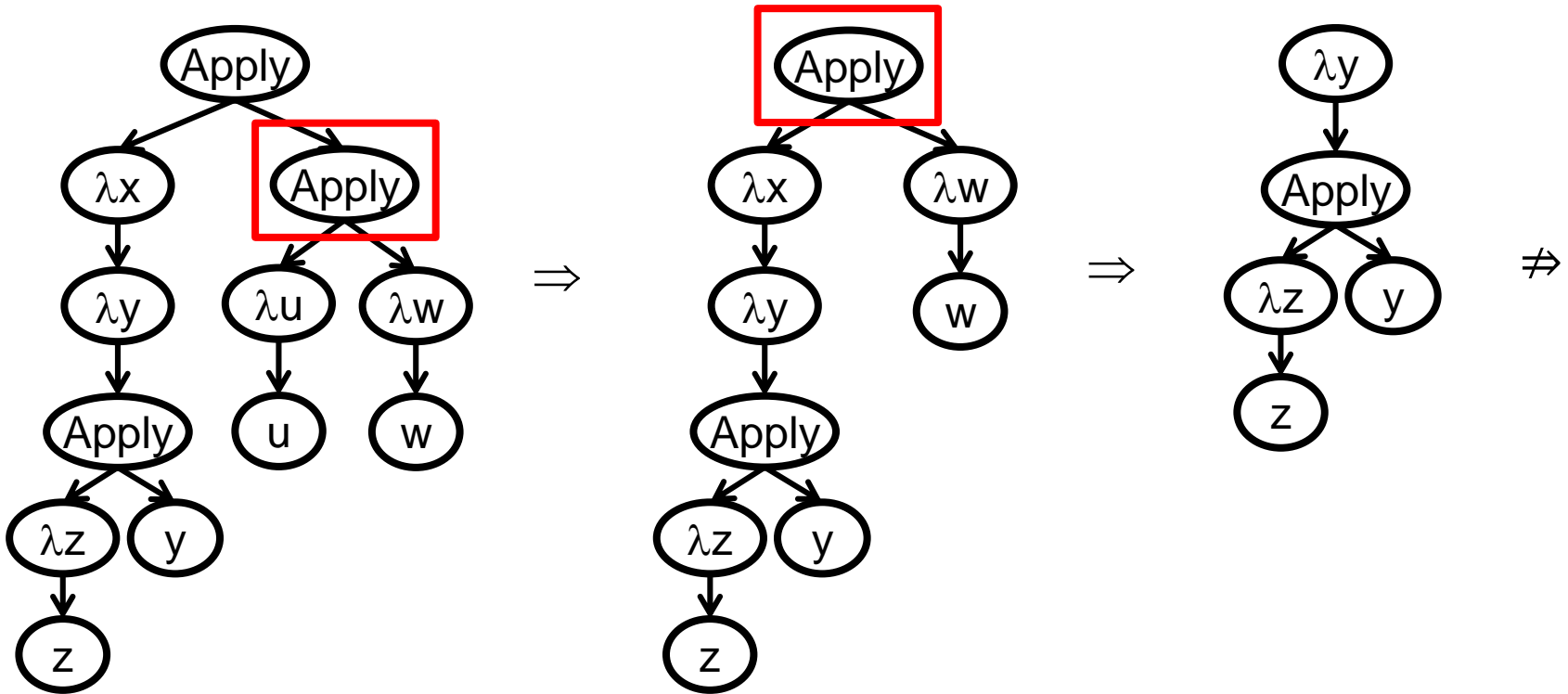
$$(\lambda x. t_1) v_2 \Rightarrow [x \mapsto v_2] t_1 \quad (\text{E-AppAbs})$$

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

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

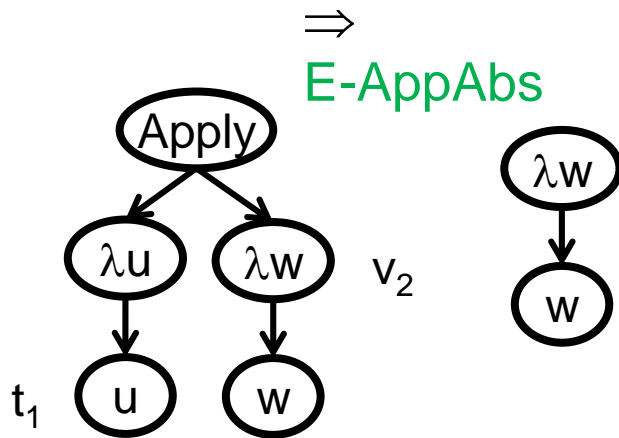
# Call By Value

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



# Call By Value OS

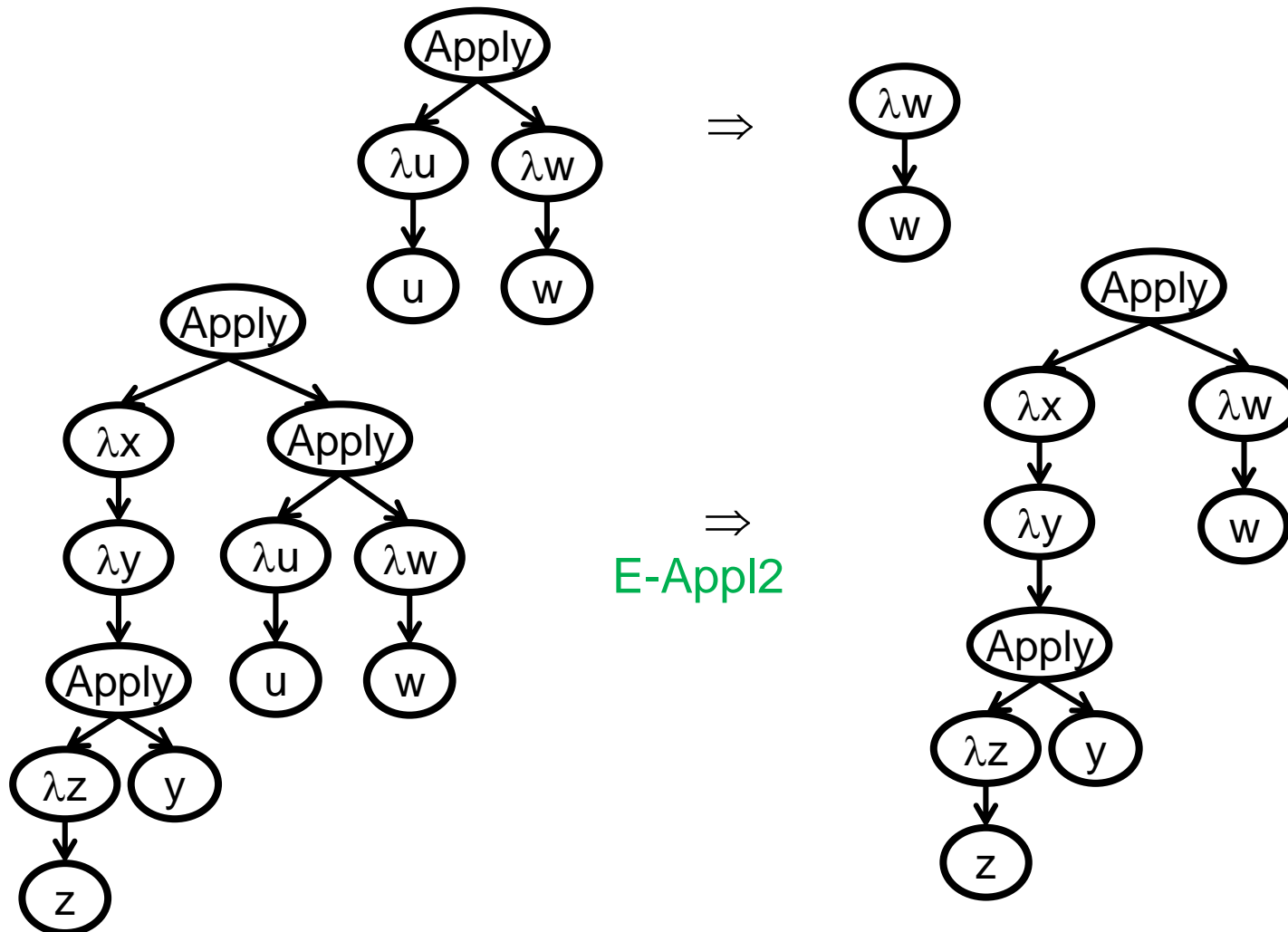
$(\lambda x. \lambda y. (\lambda z. z) y) \underline{((\lambda u. u) (\lambda w. w))}$





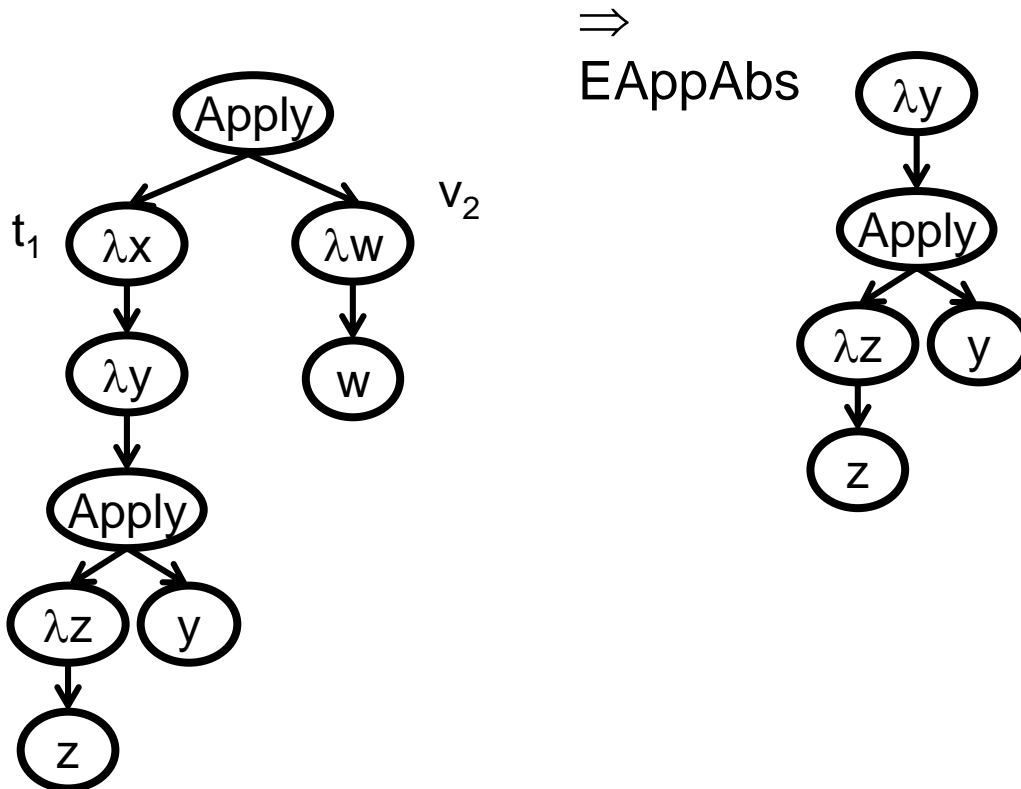
# Call By Value OS (2)

$v_1$   $t_2$   
 $(\lambda x. \lambda y. (\lambda z. z) y) ((\lambda u. u) (\lambda w. w))$



# Call By Value OS (3)

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



# Programming in $\lambda$ Calculus

- Functions with multiple arguments
- Simulating values
  - Tuples
  - Booleans
  - Numerics

# Programming in the $\lambda$ Multiple arguments

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

-> Currying

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

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

$$((\lambda x. \lambda y. x^*x + y^*y) \ 3) \ 4 \Rightarrow (\lambda y. [x \mapsto 3] x^*x + y^*y) \ 4 = (\lambda y. 3^*3 + y^*y) \ 4 \Rightarrow$$

$$[y \mapsto 4] 3^*3 + y^*y = 3^*3 + 4^*4$$

# Adding Values

- Can be explicitly added

|   |             |
|---|-------------|
| $t ::=$   | terms       |
| $x$   | variable    |
| $\lambda x. t$                                      | abstraction |
| $t t$   | application |
| $\top \top \mid \text{FF} \mid 1 \mid 2 \mid \dots$ |             |

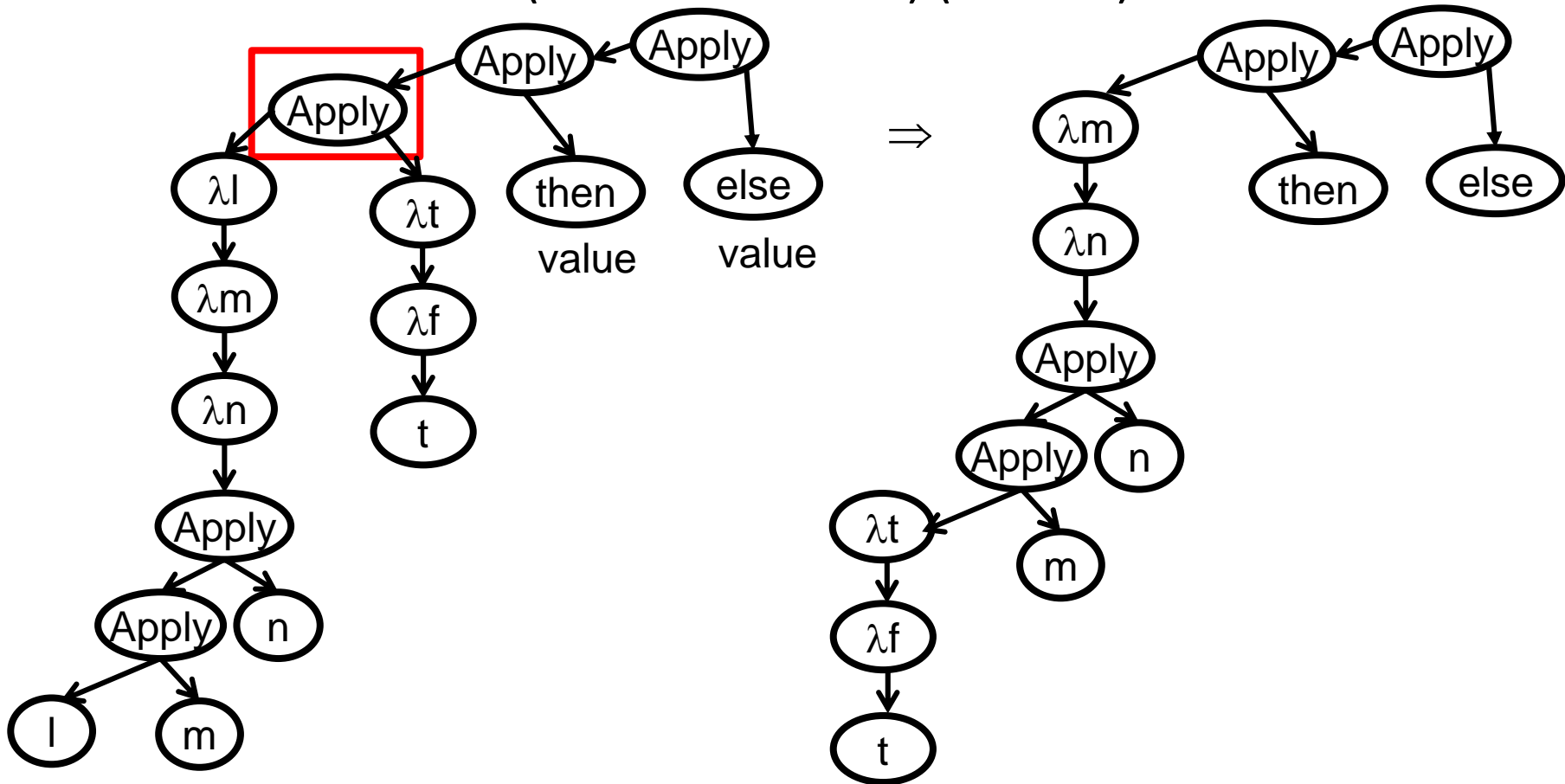
- Can be simulated

# Simulating Booleans

- $\text{tru} = \lambda t. \lambda f. t$
- $\text{fls} = \lambda t. \lambda f. f$

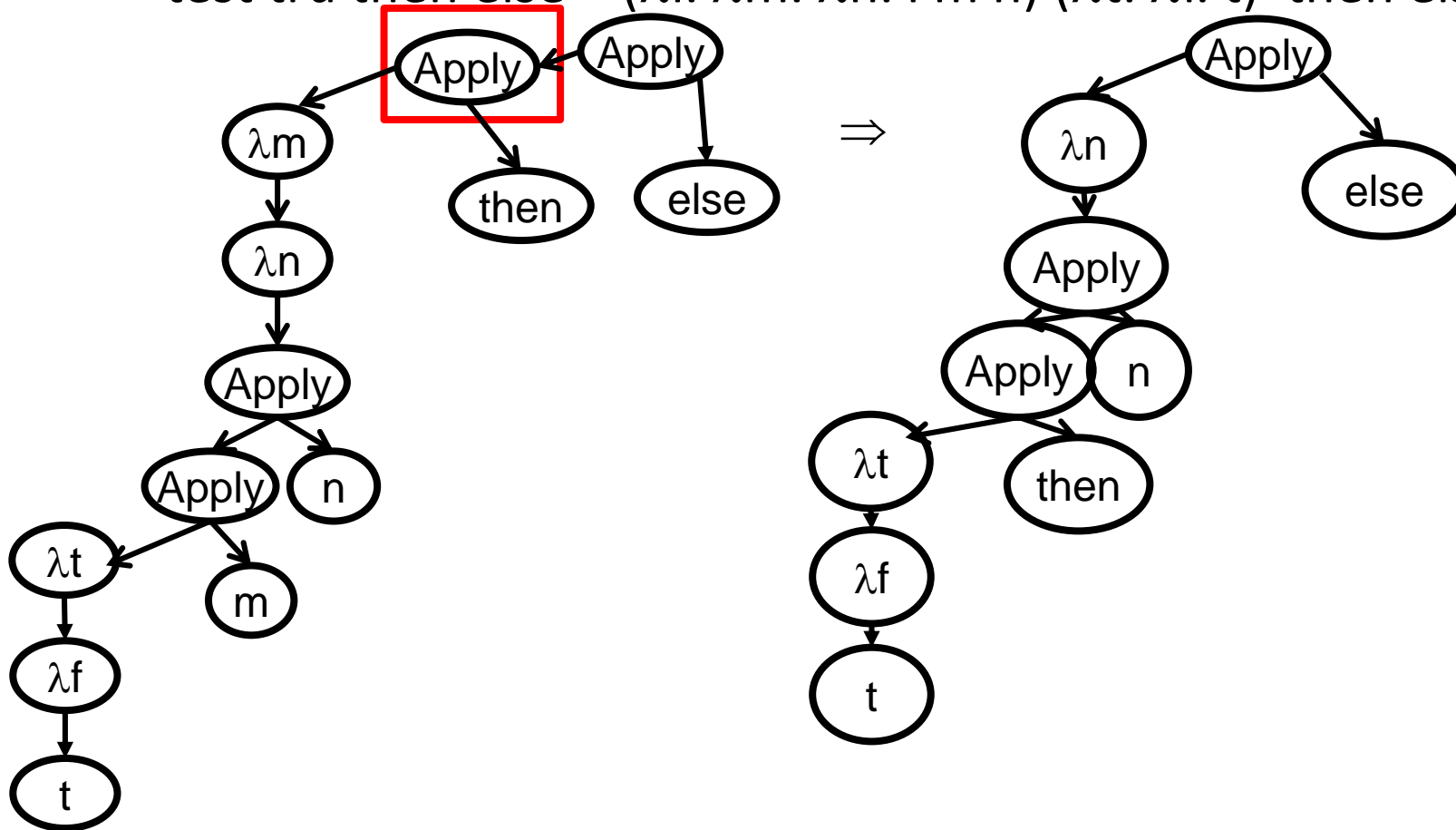
# Simulating Tests

- $\text{tru} = \lambda t. \lambda f. t$        $\text{fls} = \lambda t. \lambda f. f$
- $\text{test} = \lambda l. \lambda m. \lambda n. l\ m\ n$
- $\text{test tru then else} = (\lambda l. \lambda m. \lambda n. l\ m\ n) (\lambda t. \lambda f. t)$  then else



## Simulating Tests(2)

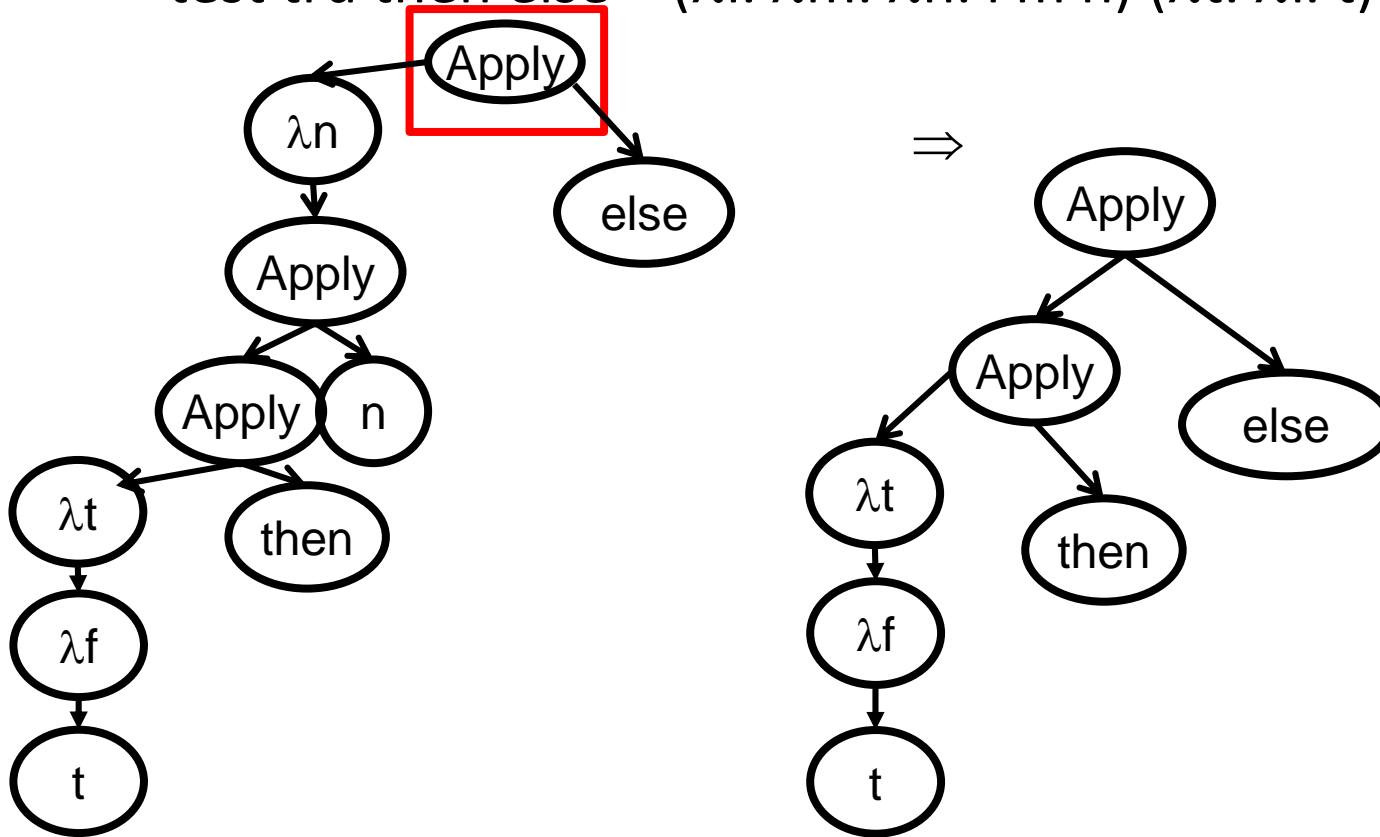
- $\text{tru} = \lambda t. \lambda f. t$        $\text{fls} = \lambda t. \lambda f. f$
- $\text{test} = \lambda l. \lambda m. \lambda n. l\ m\ n$
- $\text{test tru then else} = (\lambda l. \lambda m. \lambda n. l\ m\ n) (\lambda t. \lambda f. t)$  then else





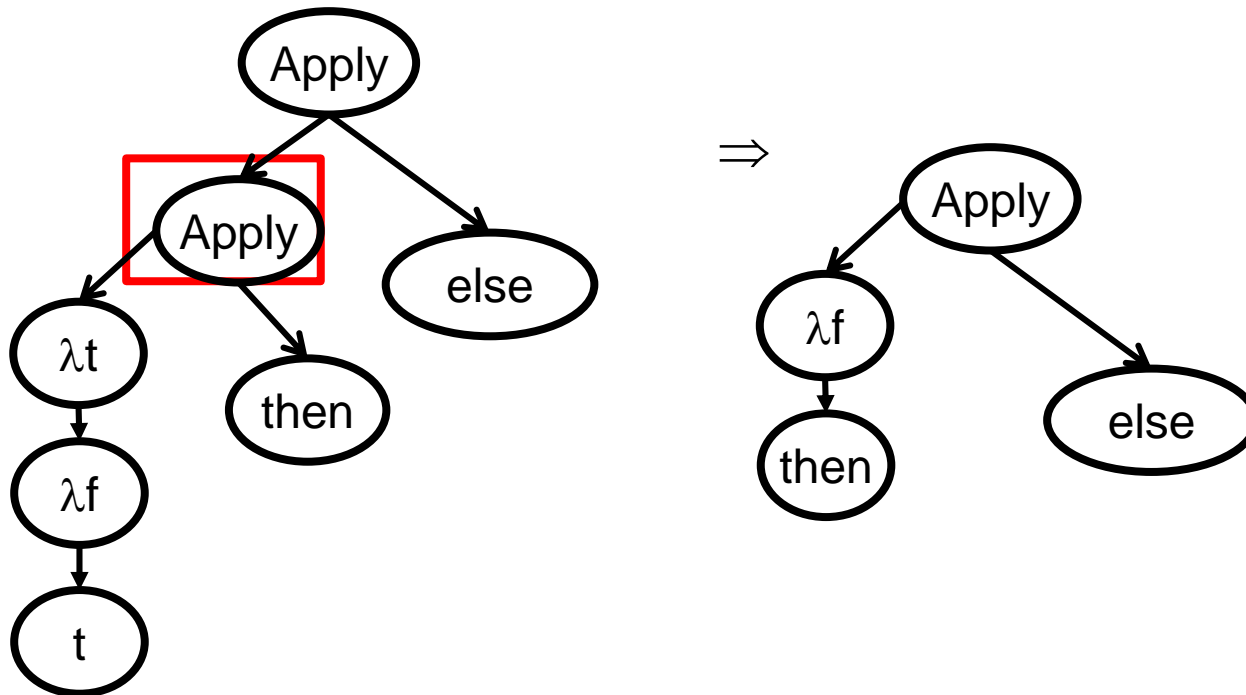
## Simulating Tests(3)

- $\text{tru} = \lambda t. \lambda f. t$                        $\text{fls} = \lambda t. \lambda f. f$
- $\text{test} = \lambda l. \lambda m. \lambda n. l m n$
- $\text{test tru then else} = (\lambda l. \lambda m. \lambda n. l m n) (\lambda t. \lambda f. t) \text{ then else}$



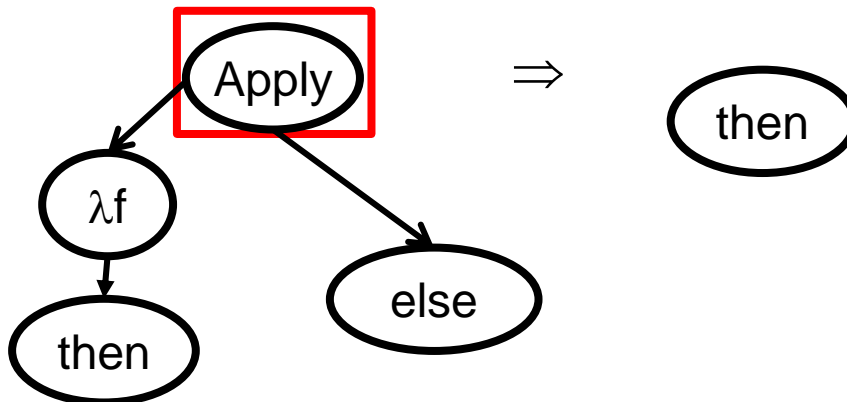
## Simulating Tests(4)

- $\text{tru} = \lambda t. \lambda f. t$                        $\text{fls} = \lambda t. \lambda f. f$
- $\text{test} = \lambda l. \lambda m. \lambda n. l m n$
- $\text{test tru then else} = (\lambda l. \lambda m. \lambda n. l m n) (\lambda t. \lambda f. t) \text{ then else}$



## Simulating Tests(5)

- $\text{tru} = \lambda t. \lambda f. t$                        $\text{fls} = \lambda t. \lambda f. f$
- $\text{test} = \lambda l. \lambda m. \lambda n. l m n$
- $\text{test tru then else} = (\lambda l. \lambda m. \lambda n. l m n) (\lambda t. \lambda f. t) \text{ then else}$



# Simulating Tests

- $\text{tru} = \lambda t. \lambda f. t$
- $\text{fls} = \lambda t. \lambda f. f$
- $\text{test} = \lambda l. \lambda m. \lambda n. l m n$
- $\text{test tru then else} = (\lambda l. \lambda m. \lambda n. l m n) (\lambda t. \lambda f. t)$
- $\text{test fls then else} = (\lambda l. \lambda m. \lambda n. l m n) (\lambda t. \lambda f. f)$

# Programming in $\lambda$ Booleans

- $\text{tru} = \lambda t. \lambda f. t$
- $\text{fls} = \lambda t. \lambda f. f$
- $\text{test} = \lambda l. \lambda m. \lambda n. l\ m\ n$
- $\text{test tru then else} = (\lambda l. \lambda m. \lambda n. l\ m\ n) (\lambda t. \lambda f. t)$
- $\text{test fls then else} = (\lambda l. \lambda m. \lambda n. l\ m\ n) (\lambda t. \lambda f. f)$
- $\text{and} = \lambda b. \lambda b'. \text{test } b\ b'\ \text{fls}$
- $\text{or} = ?$
- $\text{not} = ?$

## Programming in $\lambda$ 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
  - $\lambda x. x$  is a combinator
  - $\lambda x. \lambda y. (x y)$  is a combinator
  - $\lambda x. \lambda y. (x z)$  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

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



# 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

$$H = (\lambda f. \lambda n. (ISZERO\ n)\ 1\ (MULT\ n\ (f\ (PRED\ n))))$$

Then we can create

$$FACTORIAL = Y\ H$$

$$= (\lambda x. (\lambda y. x\ (y\ y))\ (\lambda y. x\ (y\ y)))\ (\lambda f. \lambda n. (ISZERO\ n)\ 1\ (MULT\ n\ (f\ (PRED\ n))))$$

Reference: [http://en.wikipedia.org/wiki/Y\\_combinator](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  $(\lambda x. x)$

# Typed Lambda Calculus

Chapter 9

Benjamin Pierce

Types and Programming Languages

# Call-by-value Operational Semantics

|                |             |                |                    |
|----------------|-------------|----------------|--------------------|
| $t ::=$        | terms       | $v ::=$        | values             |
| $x$            | variable    | $\lambda x. t$ | abstraction values |
| $\lambda x. t$ | abstraction |                |                    |
| $t t$          | application |                |                    |

$$(\lambda x. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12} \text{ (E-AppAbs)}$$

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

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

# 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  $(\lambda x. x)$

# A Naïve Attempt

- Add function type  $\rightarrow$
- Type rule  $\lambda x. t : \rightarrow$ 
  - $\lambda x. x : \rightarrow$
  - If true then  $(\lambda x. x)$  else  $(\lambda x. \lambda y y) : \rightarrow$
- Too Coarse

# Simple Types

$T ::=$                     types  
          Bool            type of Booleans  
           $T \rightarrow T$       type of functions

$$T_1 \rightarrow T_2 \rightarrow T_3 = T_1 \rightarrow (T_2 \rightarrow T_3)$$

# Explicit vs. Implicit Types

- How to define the type of  $\lambda$  abstractions?
  - **Explicit**: defined by the programmer

|                   |                      |
|-------------------|----------------------|
| $t ::=$           | Type $\lambda$ terms |
| $x$               | variable             |
| $\lambda x: T. t$ | abstraction          |
| $t t$             | application          |

- **Implicit**: Inferred by analyzing the body
- The **type checking problem**: Determine if typed term is well typed
- The **type inference problem**: Determine if there exists a type for (an untyped) term which makes it well typed



# Simple Typed Lambda Calculus

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

|                   |                    |
|-------------------|--------------------|
| $T ::=$           | types              |
| $T \rightarrow T$ | types of functions |

# Typing Function Declarations

$$\frac{x : T_1 \vdash t_2 : T_2}{\vdash (\lambda x : T_1. t_2) : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

A typing context  $\Gamma$  maps free variables into types

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\lambda x : T_1. t_2) : T_1 \rightarrow T_2} \quad (\text{T-ABS})$$

# Typing Free Variables

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T} \text{ (T-VAR)}$$

# Typing Function Applications

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-APP})$$

# Typing Conditionals

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{(T-IF)}$$

if true then  $(\lambda x: \text{Bool}. x)$  else  $(\lambda y: \text{Bool}. \text{not } y)$

# SOS for Simple Typed Lambda Calculus

|                   |                    |  |            |
|-------------------|--------------------|--|------------|
| $t ::=$           | terms              | $t_1 \rightarrow t_2$  |            |
| $x$               | variable           |  |            |
| $\lambda x: T. t$ | abstraction        | $t_1 \rightarrow t'_1$   |            |
| $t t$             | application        | $t_1 t_2 \rightarrow t'_1 t_2$                                       | (E-APP1)   |
| $v ::=$           | values             | $t_2 \rightarrow t'_2$   |            |
| $\lambda x: T. t$ | abstraction values | $v_1 t_2 \rightarrow v_1 t'_2$                                       | (E-APP2)   |
|                   |                    | $(\lambda x: T_{11}. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12}$ | (E-APPABS) |

$T ::=$  types

$T \rightarrow T$  types of functions

# Type Rules

|                    |                       |   |
|--------------------|-----------------------|---|
| $t ::=$            | terms                 | $\Gamma \vdash t : T$   |
| $x$                | variable              | $\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$  |
| $\lambda x : T. t$ | abstraction           | $\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \text{ (T-ABS)}$                     |
| $T ::=$            | types                 | $\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}$  |
| $T \rightarrow T$  | types of functions    | $\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ (T-APP)}$ |
| $\Gamma ::=$       | context               |   |
| $\emptyset$        | empty context         |   |
| $\Gamma, x : T$    | term variable binding |   |

|                    |                       |   |
|--------------------|-----------------------|---|
| $t ::=$            | terms                 | $\Gamma \vdash t : T$   |
| $x$                | variable              | $\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$  |
| $\lambda x : T. t$ | abstraction           | $\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \text{ (T-ABS)}$                                       |
| $t t$              | application           | $\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ (T-APP)}$                   |
| true               | constant true         | $\Gamma \vdash \text{true} : \text{Bool} \text{ (T-TRUE)}$  |
| false              | constant false        | $\Gamma \vdash \text{false} : \text{Bool} \text{ (T-FALSE)}$  |
| if t then t else t | conditional           | $\frac{\Gamma \vdash t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{ (T-IF)}$ |
| $T ::=$            | types                 |   |
| Bool               | Boolean type          |   |
| $T \rightarrow T$  | types of functions    |   |
| $\Gamma ::=$       | context               |   |
| $\emptyset$        | empty context         |   |
| $\Gamma, x : T$    | term variable binding |   |



# Examples

- $(\lambda x:\text{Bool}. x)$  true
- if true then  $(\lambda x:\text{Bool}. x)$  else  $(\lambda x:\text{Bool}. x)$
- if true then  $(\lambda x:\text{Bool}. x)$  else  $(\lambda x:\text{Bool}.$   
 $\lambda y:\text{Bool}. x)$

# Simple Example

$(\lambda x:\text{Bool}. x) \text{ true}$

$\Gamma \vdash t : T$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ (T-APP)}$$

$\Gamma \vdash \text{true} : \text{Bool}$  (T-TRUE)

$\Gamma \vdash \text{false} : \text{Bool}$  (T-FALSE)

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{ (T-IF)}$$

# Another Example

$$\Gamma \vdash t : T$$

if true then  $(\lambda x:\text{Bool}. x)$  else  $(\lambda x:\text{Bool}. x)$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$
$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \text{ (T-ABS)}$$
$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ (T-APP)}$$
$$\Gamma \vdash \text{true} : \text{Bool} \text{ (T-TRUE)}$$
$$\Gamma \vdash \text{false} : \text{Bool} \text{ (T-FALSE)}$$
$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{ (T-IF)}$$

# Another Example

if true then  $(\lambda x:\text{Bool}. x)$  else  
 $\lambda x:\text{Bool}. \lambda y:\text{Bool}. x)$

$$\Gamma \vdash t : T$$
$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$
$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \text{ (T-ABS)}$$
$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \text{ (T-APP)}$$
$$\Gamma \vdash \text{true} : \text{Bool} \text{ (T-TRUE)}$$
$$\Gamma \vdash \text{false} : \text{Bool} \text{ (T-FALSE)}$$
$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \text{ (T-IF)}$$

# The Typing Relation

- Formally the typing relation is the smallest ternary relation on contexts, terms and types
  - in terms of inclusion
- A term  $t$  is **typable** in a given context  $\Gamma$  (**well typed**) if there exists some type  $T$  such that  $\Gamma \vdash t : T$
- Interesting on closed terms (empty contexts)

# Inversion of the typing relation

- $\Gamma \vdash x : R \Rightarrow x : R \in \Gamma$
- $\Gamma \vdash \lambda x : T_1. t_2 : R \Rightarrow R = T_1 \rightarrow R_2$  for some  $R_2$  with  $\Gamma \vdash t_2 : R_2$
- $\Gamma \vdash t_1 t_2 : R \Rightarrow$  there exists  $T_{11}$  such that  $\Gamma \vdash t_1 : T_{11} \rightarrow R$  and  $\Gamma \vdash t_2 : T_{11}$
- $\Gamma \vdash \text{true} : R \Rightarrow R = \text{Bool}$
- $\Gamma \vdash \text{false} : R \Rightarrow R = \text{Bool}$
- $\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : R \Rightarrow \Gamma \vdash t_1 : \text{Bool}, \Gamma \vdash t_2 : R, \Gamma \vdash t_3 : R$

# Uniqueness of Types

- Each term  $t$  has at most one type in any given context
  - If  $t$  is typable then
    - its type is unique
    - There is a unique type derivation tree for  $t$

# Type Safety

- Well typed programs cannot go wrong
- If  $t$  is well typed then either  $t$  is a value or there exists an evaluation step  $t \rightarrow t'$   
[Progress]
- If  $t$  is well typed and there exists an evaluation step  $t \rightarrow t'$  then  $t'$  is also well typed  
[Preservation]



# Canonical Forms

- If  $v$  is a value of type `Bool` then  $v$  is either `true` or `false`
- If  $v$  is a value of type  $T_1 \rightarrow T_2$  then  $v = \lambda x: T_1. t_2$

# Progress Theorem

- Does not hold on terms with free variables
- For every closed well typed term  $t$ , either  $t$  is a value or there exists  $t'$  such that  $t \rightarrow t'$

# Preservation Theorem

- If  $\Gamma \vdash t : T$  and  $\Delta$  is a permutation of  $\Gamma$  then  $\Delta \vdash t : T$  [Permutation]
- If  $\Gamma \vdash t : T$  and  $x \notin \text{dom}(\Gamma)$  then  $\Gamma, x \vdash t : T$  with a proof of the same depth [Weakening]
- If  $\Gamma, x : S \vdash t : T$  and  $\Gamma \vdash s : S$  then  $\Gamma \vdash [x \mapsto s] t : T$  [Preservation of types under substitution]
- $\Gamma \vdash t : T$  and  $t \rightarrow t'$  then  $\Gamma \vdash t' : T$

# SOS for Simple Typed Lambda Calculus

|                   |                    |  |            |
|-------------------|--------------------|--|------------|
| $t ::=$           | terms              | $t_1 \rightarrow t_2$  |            |
| $x$               | variable           |  |            |
| $\lambda x: T. t$ | abstraction        | $t_1 \rightarrow t'_1$   |            |
| $t t$             | application        | $t_1 t_2 \rightarrow t'_1 t_2$                                       | (E-APP1)   |
| $v ::=$           | values             | $t_2 \rightarrow t'_2$   |            |
| $\lambda x: T. t$ | abstraction values | $v_1 t_2 \rightarrow v_1 t'_2$                                       | (E-APP2)   |
|                   |                    | $(\lambda x: T_{11}. t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12}$ | (E-APPABS) |

$T ::=$  types

$T \rightarrow T$  types of functions

# Erasure and Typability

- Types are used for preventing errors and generating more efficient code
- Types are not used at runtime

$$\text{erase}(x) = x$$

$$\text{erase}(\lambda x: T_1. t_2) = \lambda x. \text{erase}(t_2)$$

$$\text{erase}(t_1 t_2) = \text{erase}(t_1) \text{erase}(t_2)$$

- If  $t \rightarrow t'$  under typed evaluation relation, then  $\text{erase}(t) \rightarrow \text{erase}(t')$
- A term  $t$  in the untyped lambda calculus is **typable** if there exists a typed term  $t'$  such that  $\text{erase}(t') = t$

# Summary

- Constructive rules for preventing runtime errors in a Turing complete programming language
- Efficient type checking
- Unique types
- Type safety
- But limits programming

# 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
  - ...