Lambda Calculus

Oded Padon & Mooly Sagiv

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

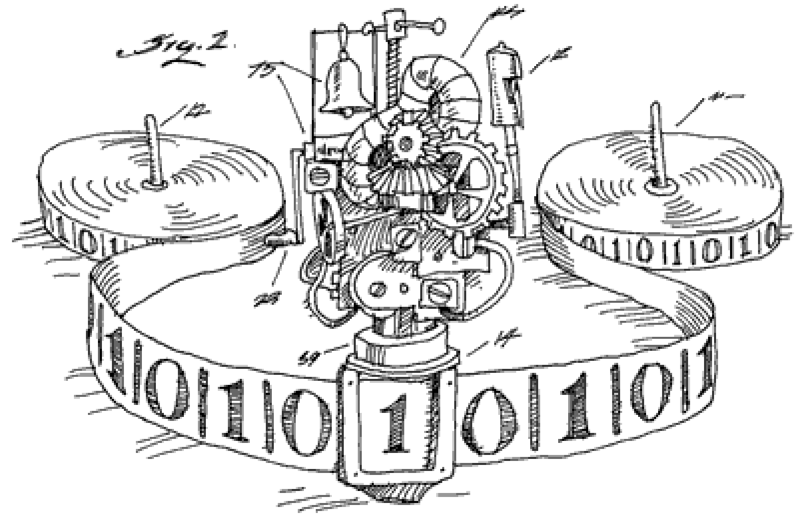
Benjamin Pierce

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

What is λ calculus

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

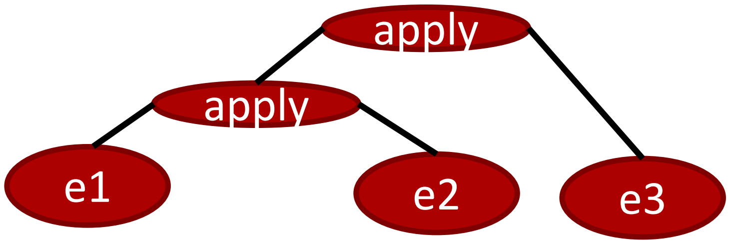
Basics

- 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

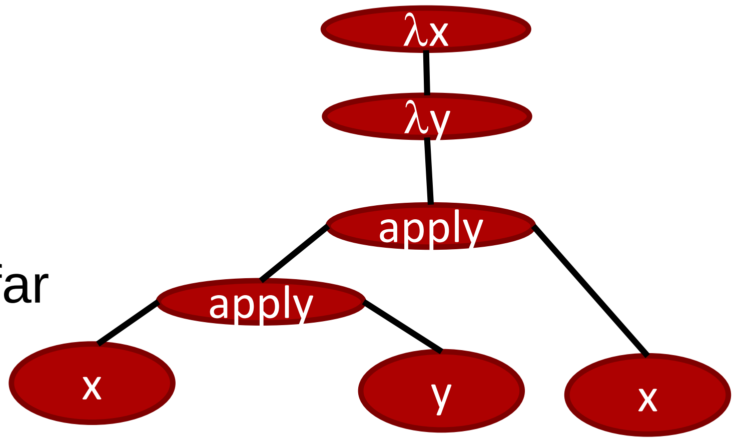
Syntactic Conventions

- 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

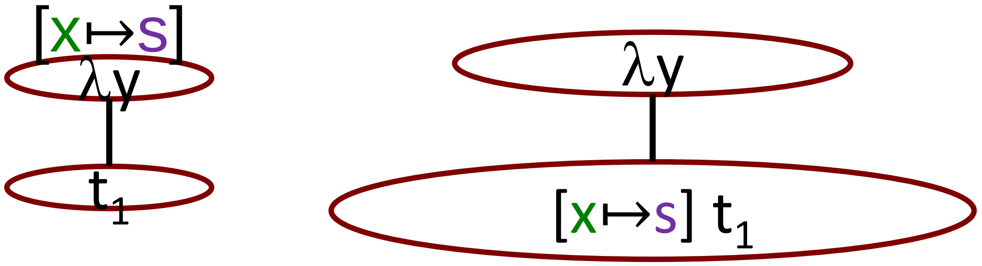
$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

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

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

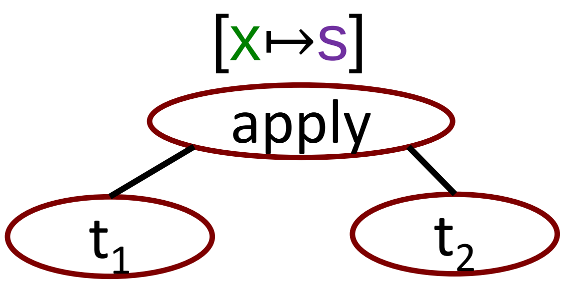
$$(\lambda\, x.\, t_1)\; t_2 \Rightarrow_\beta [x \mapsto t_2]\; t_1 \qquad (\beta\text{-reduction})$$

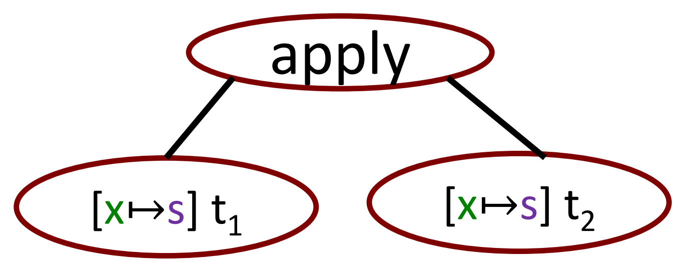
$[x \mapsto s]\; x = s$

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

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

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



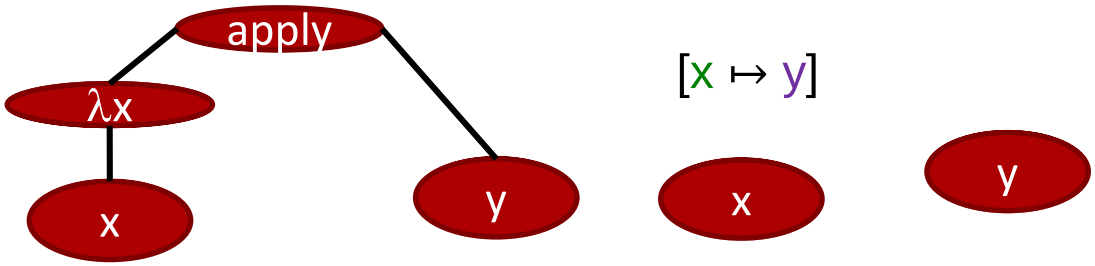


Example Beta-Reduction

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

redex

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

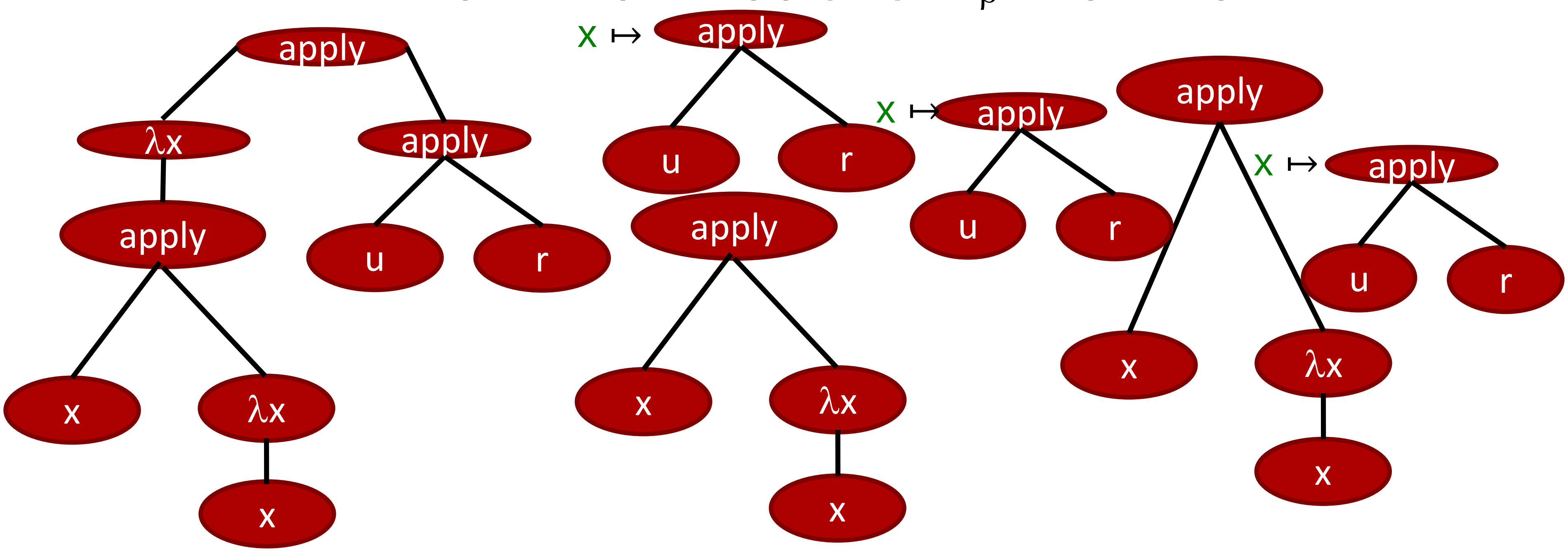


Example Beta-Reduction (ex 2)

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

redex

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

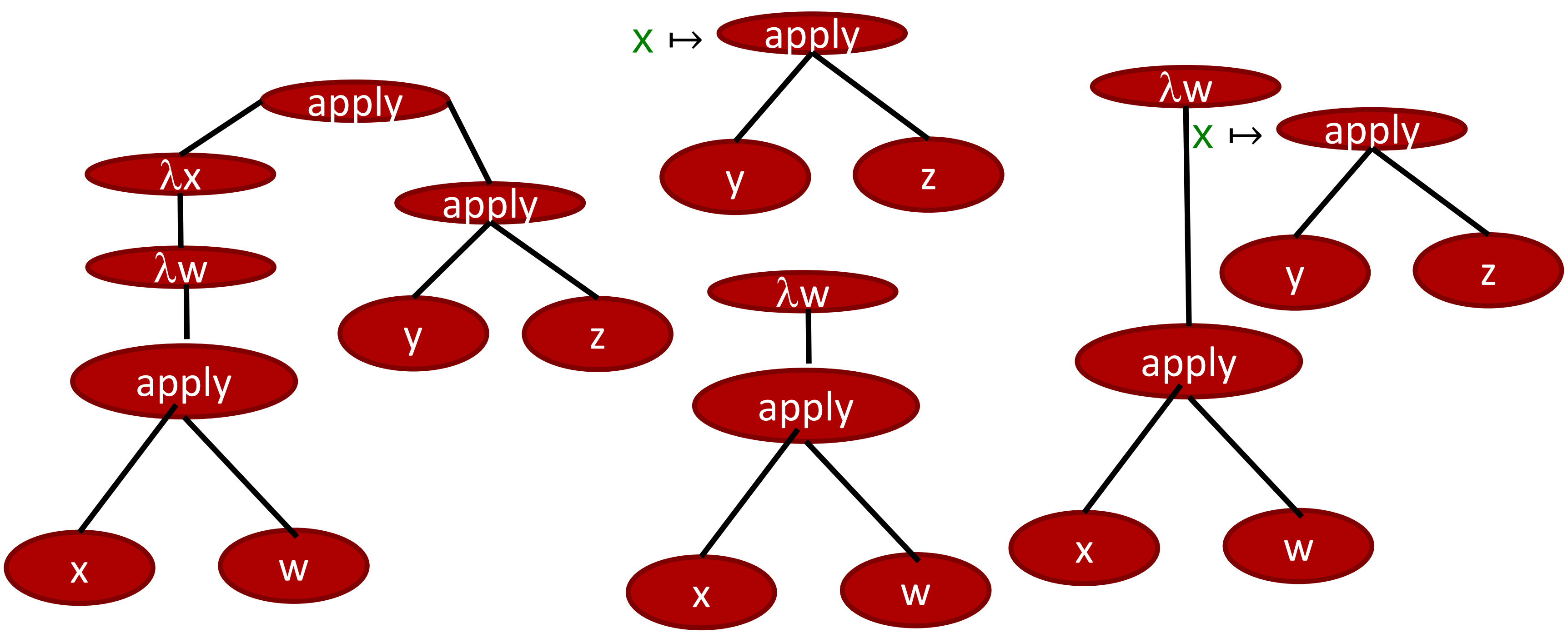


Example Beta-Reduction (ex 3)

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

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

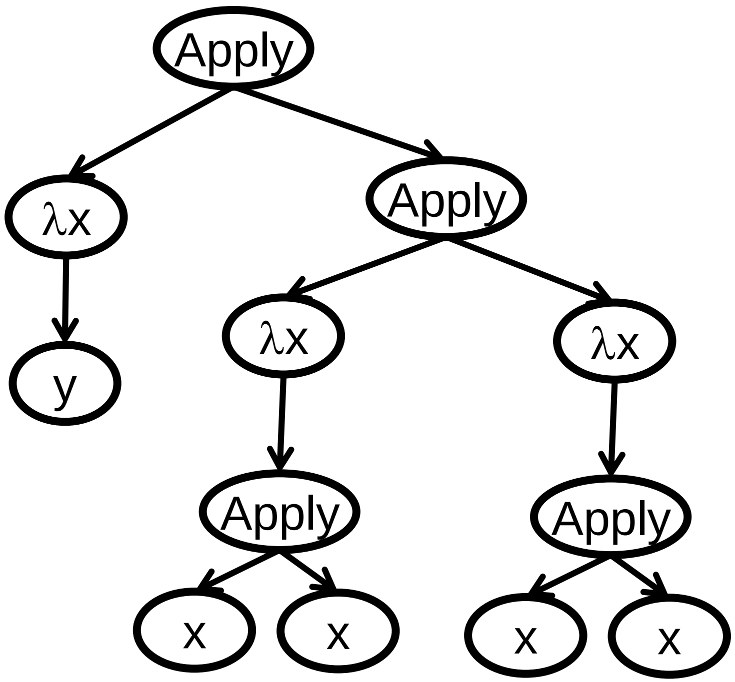
Simple Exercise

- 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?

Divergence

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

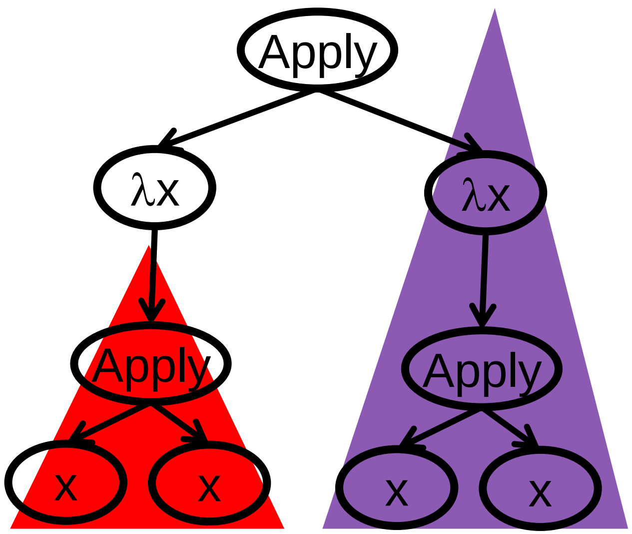
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



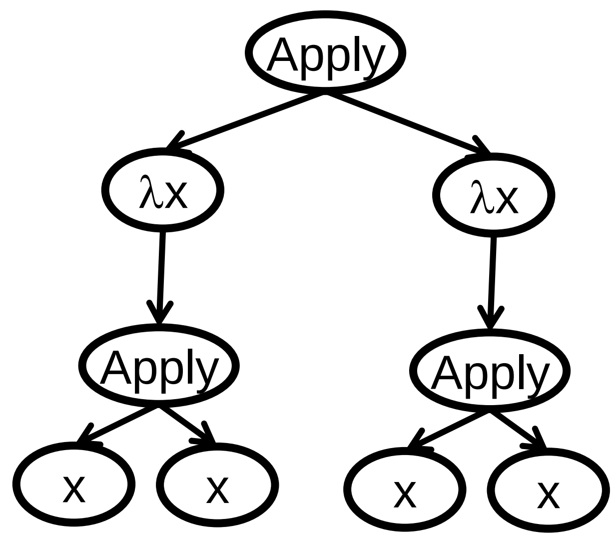
Divergence

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

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



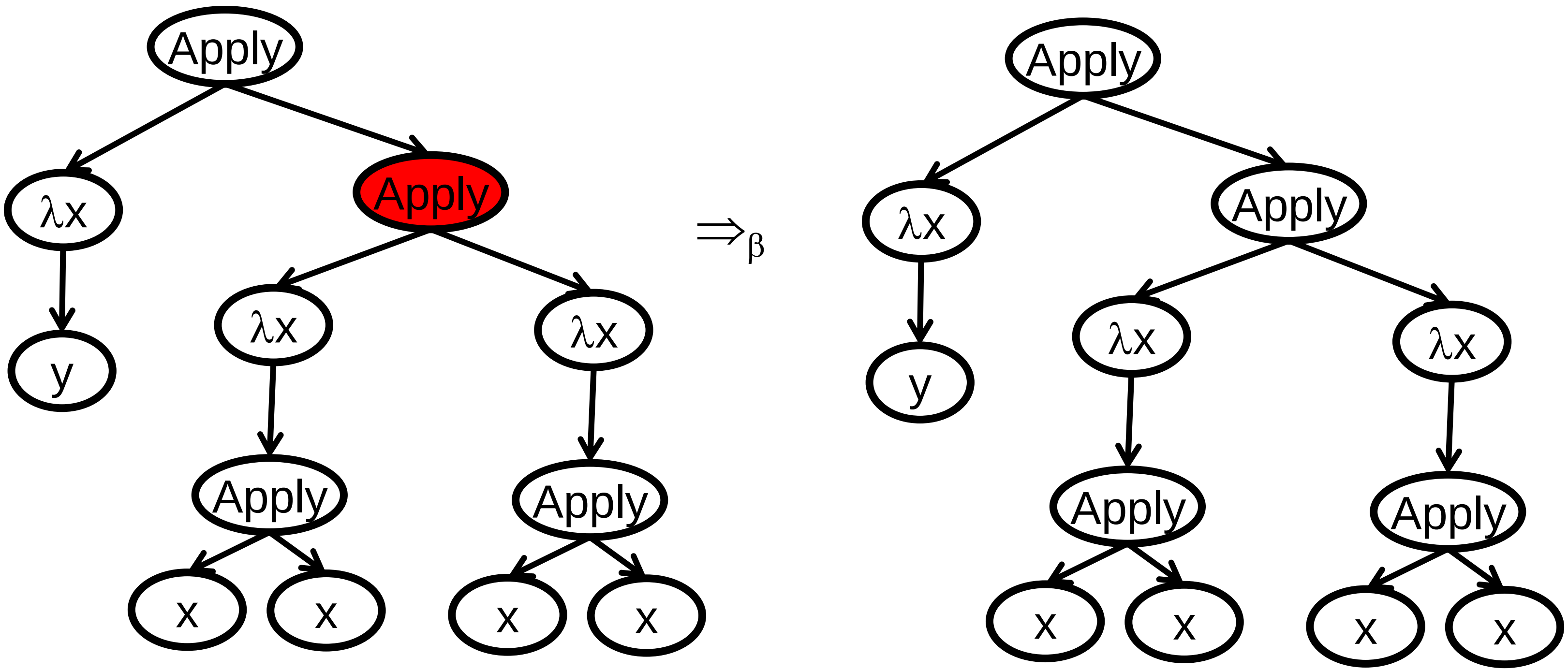
$\Rightarrow_\beta$



Different Evaluation Orders

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

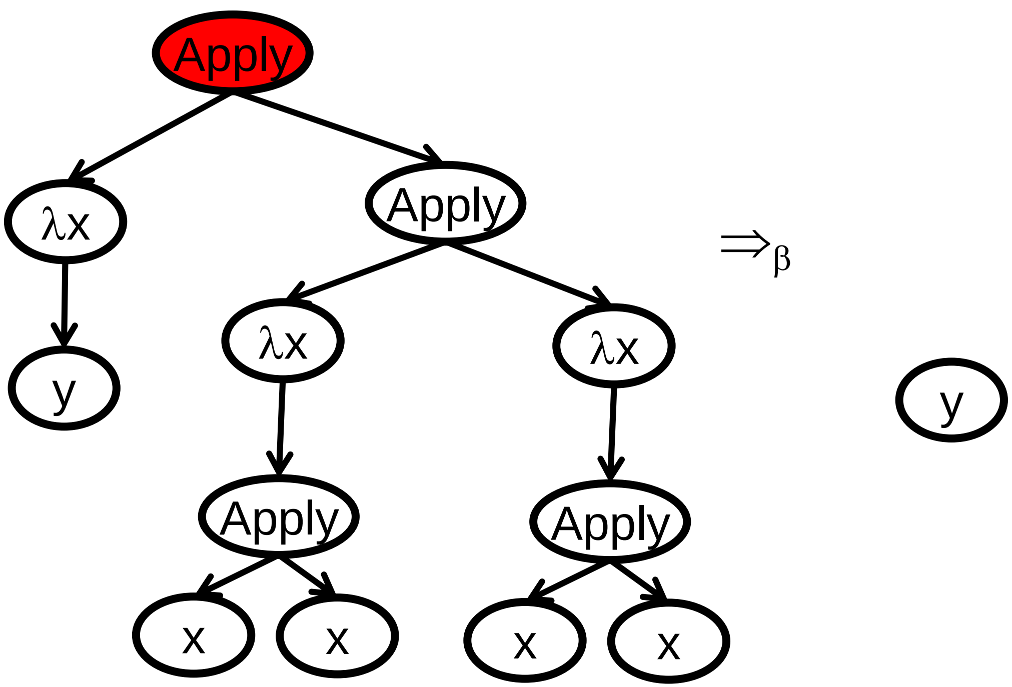
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

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

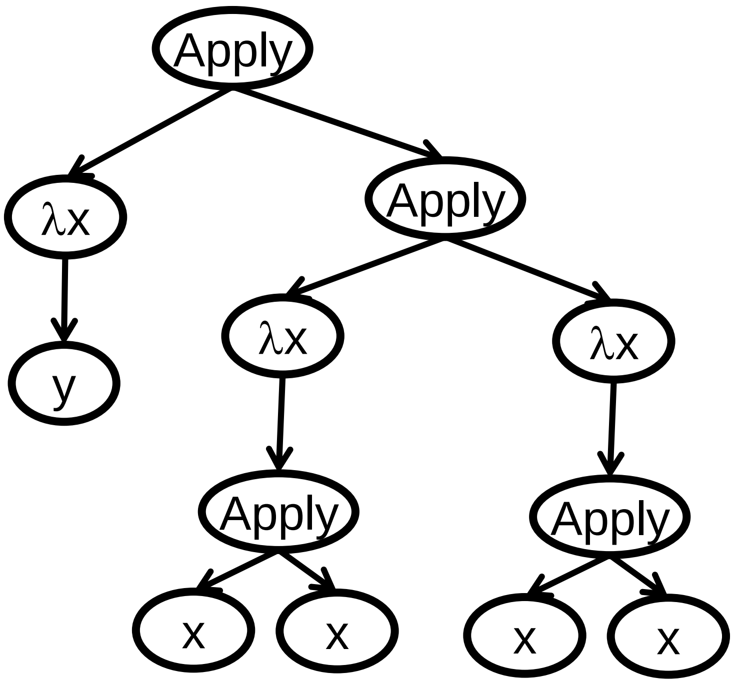
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

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

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

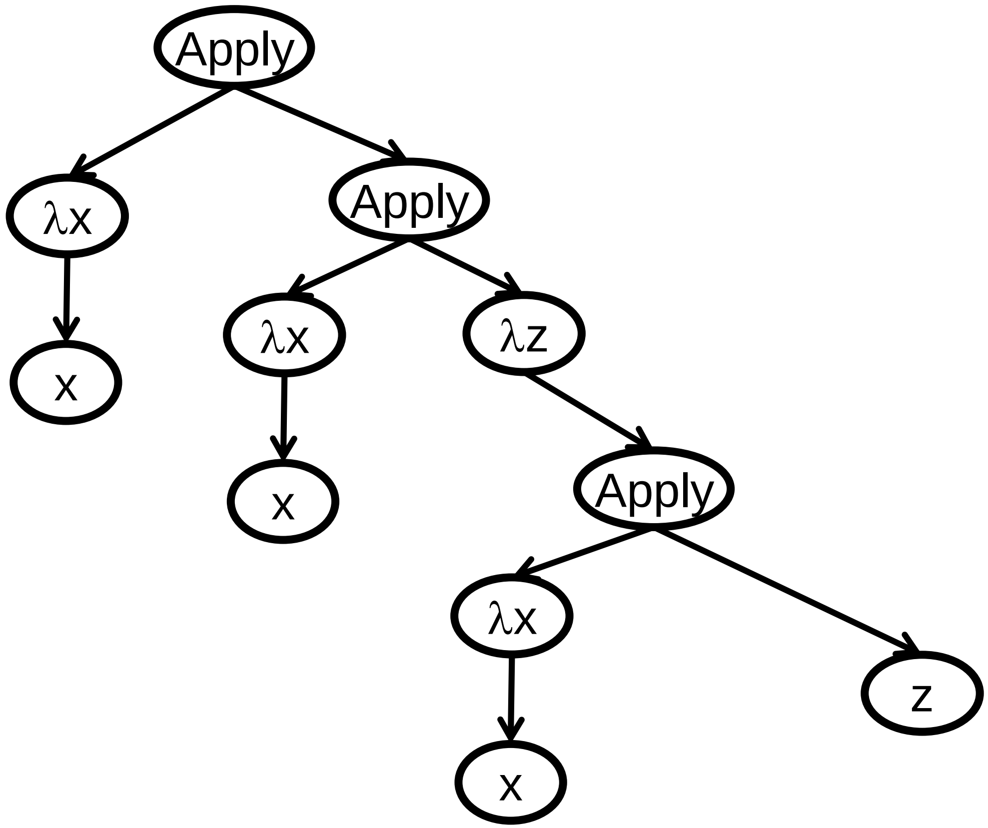



```
def f():
    while True: pass

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)

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

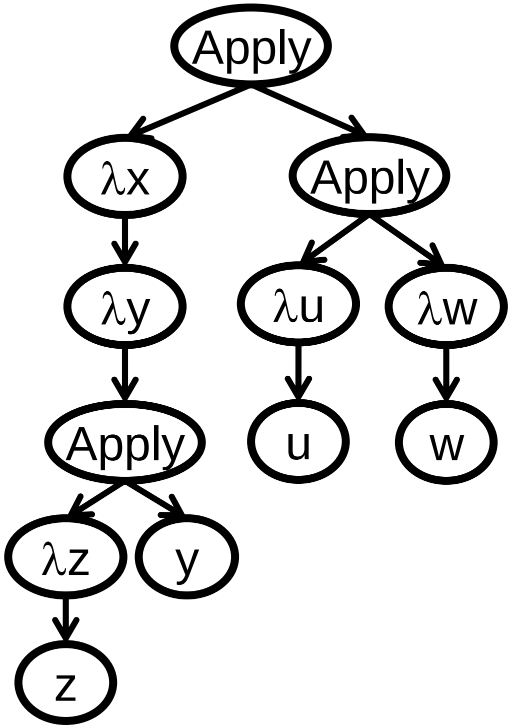


Order of Evaluation

- 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

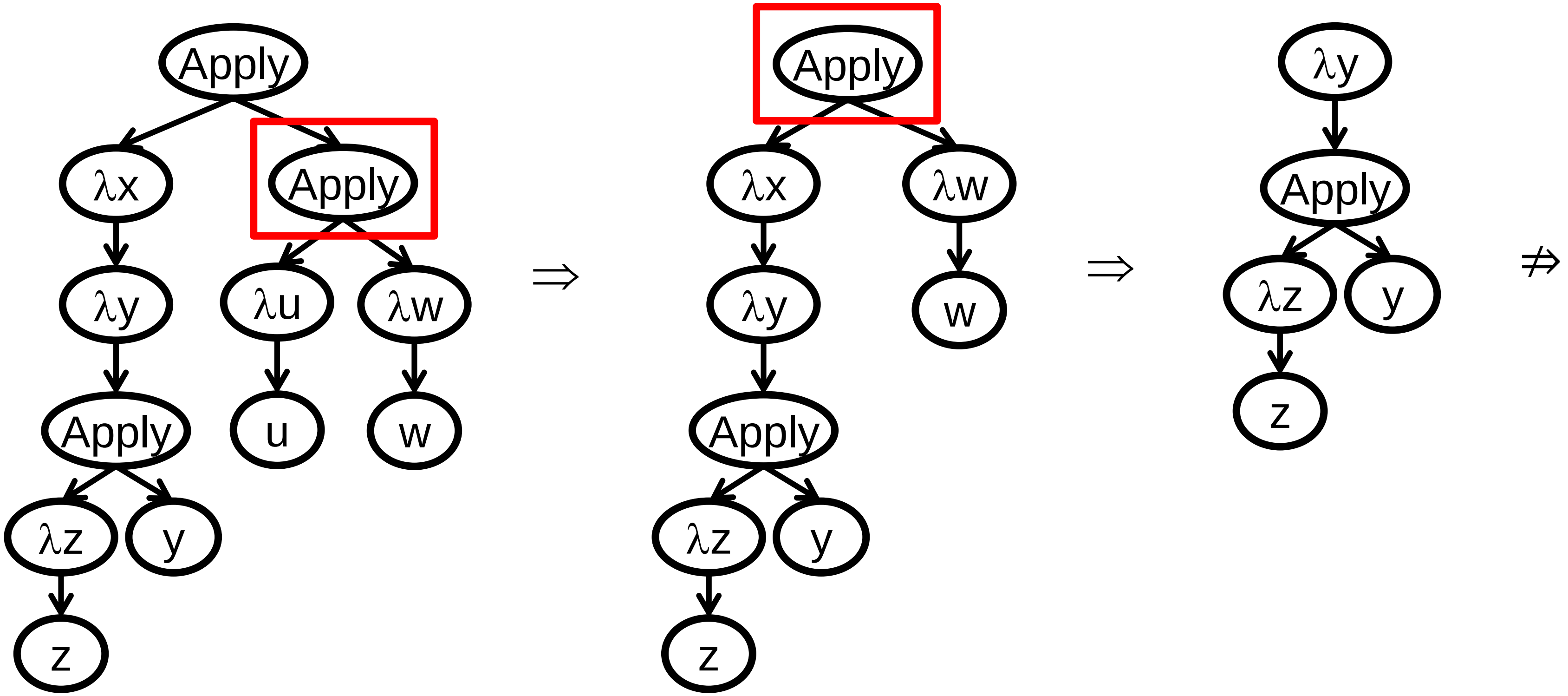
Different Evaluation Orders

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



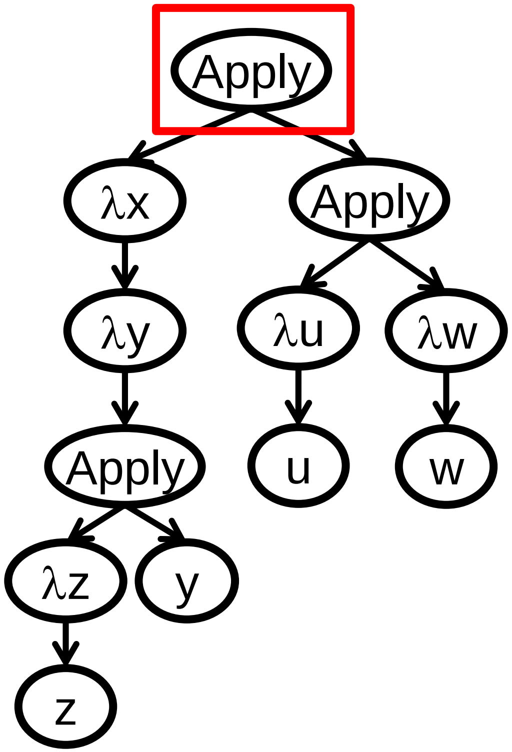
Call By Value

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

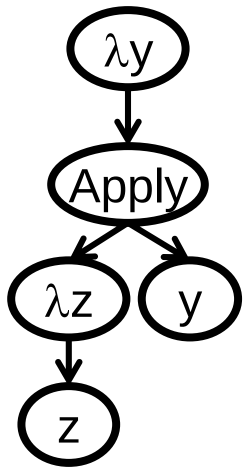


Call By Name (Lazy)

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



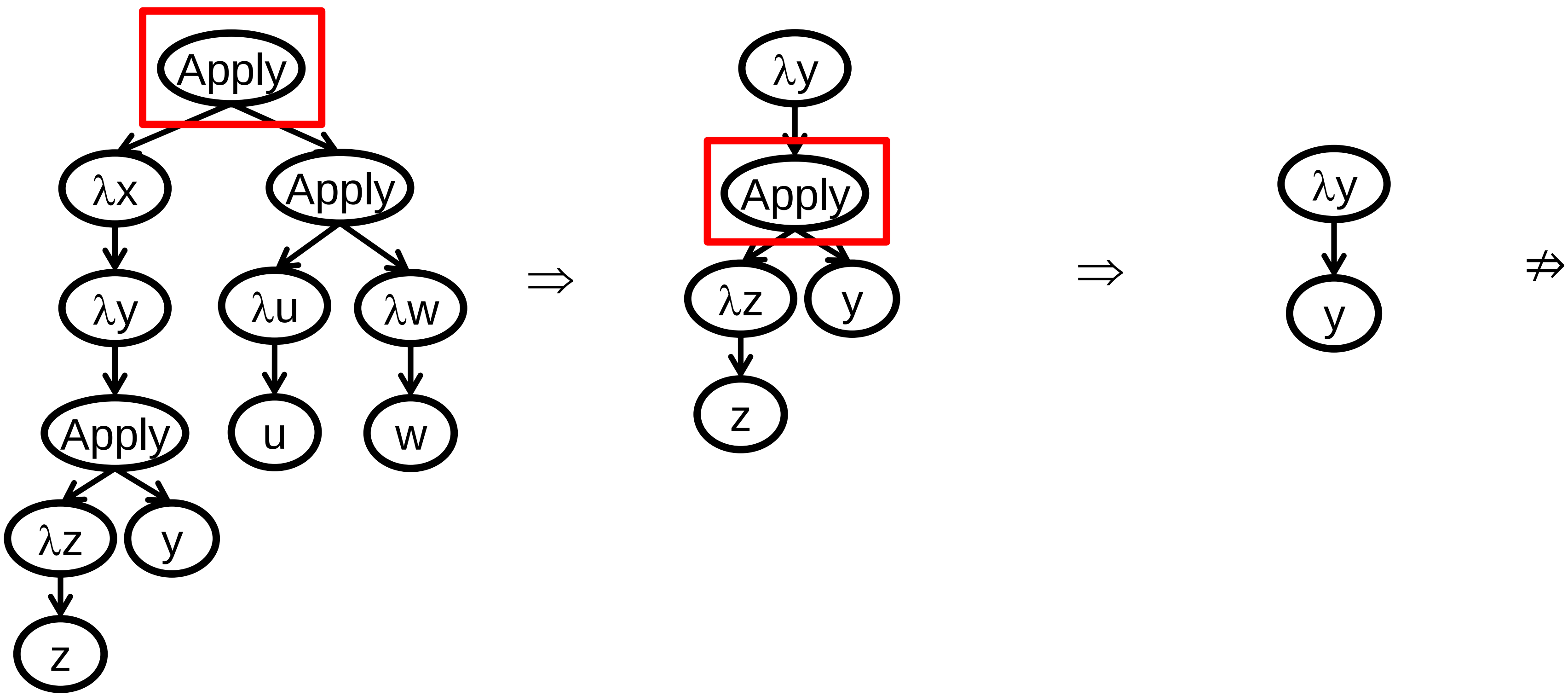
⇒



⇏

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	

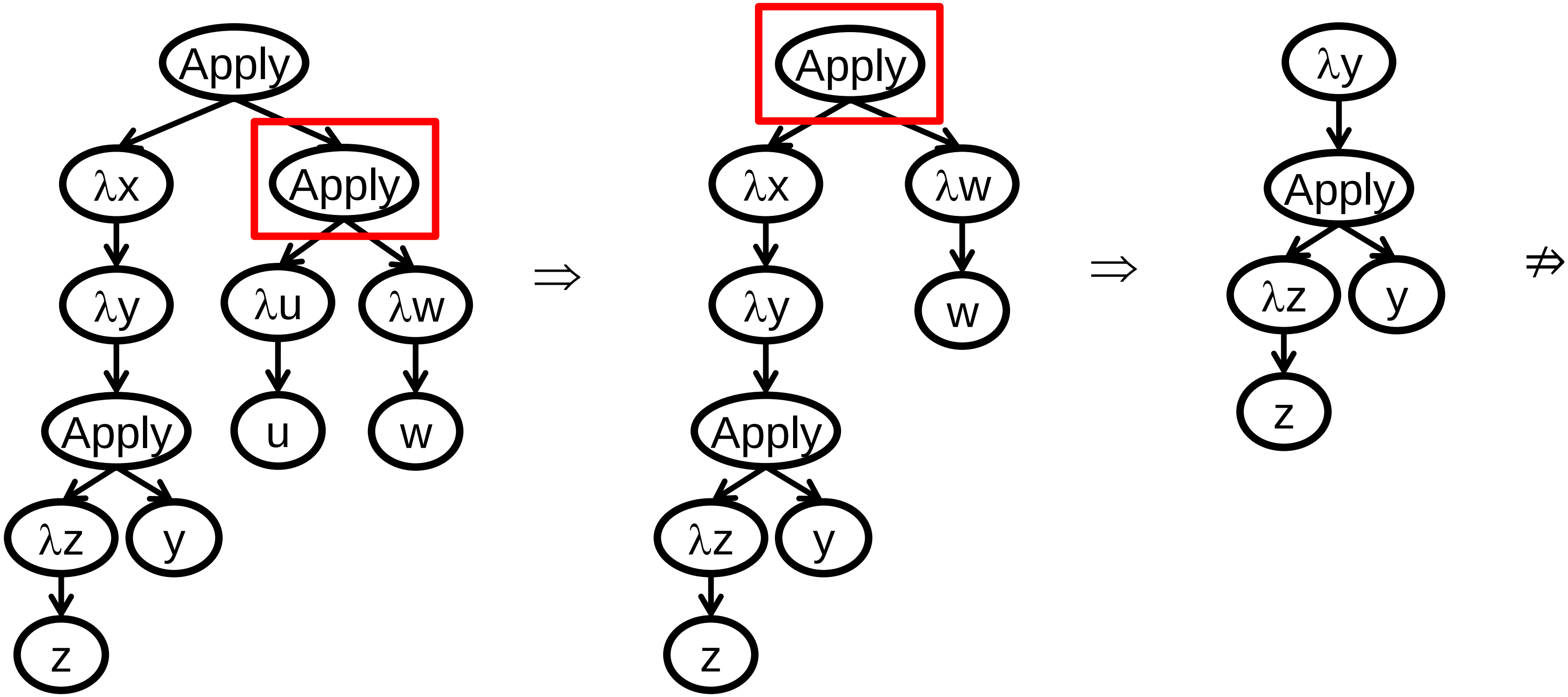
$$(\lambda\ x.\ t_1)\ v_2 \Rightarrow [x \mapsto v_2]\ t_1$$ (E-AppAbs)

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

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

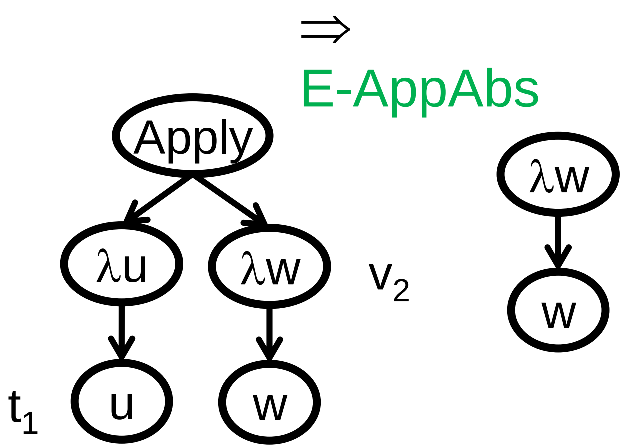
Call By Value

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



Call By Value OS

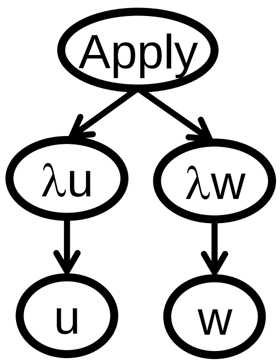
$(\lambda\ x.\ \lambda\ y.\ (\lambda\ z.\ z)\ y)\ \underline{((\lambda\ u.\ u)\ (\lambda\ w.\ w))}$



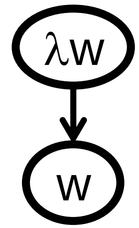
Call By Value OS (2)

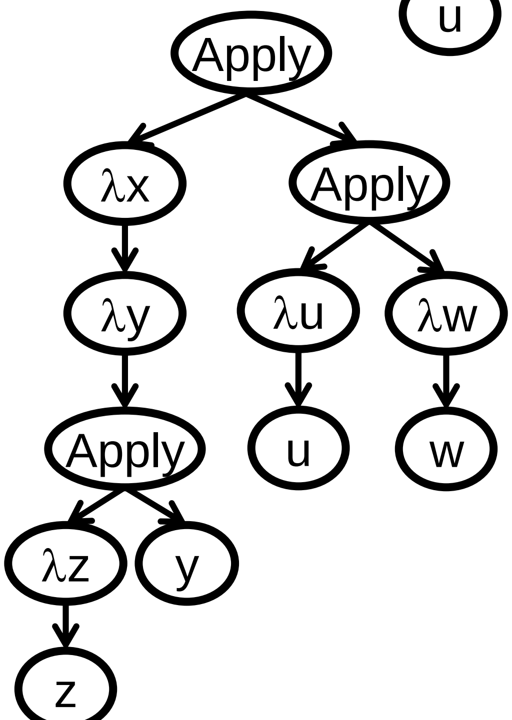
v_1 t_2

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



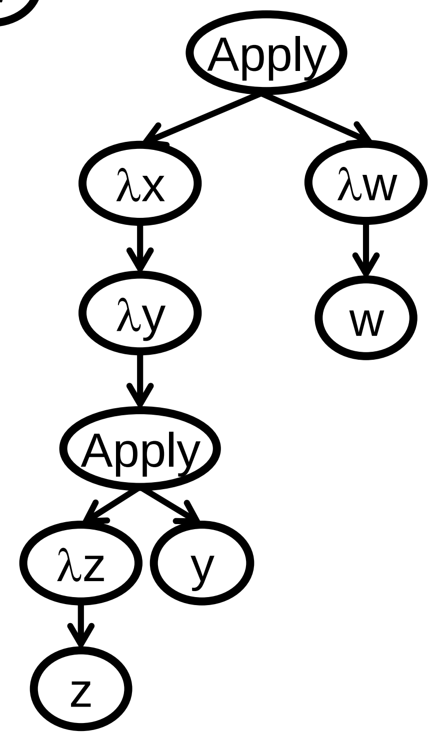
$\Rightarrow$





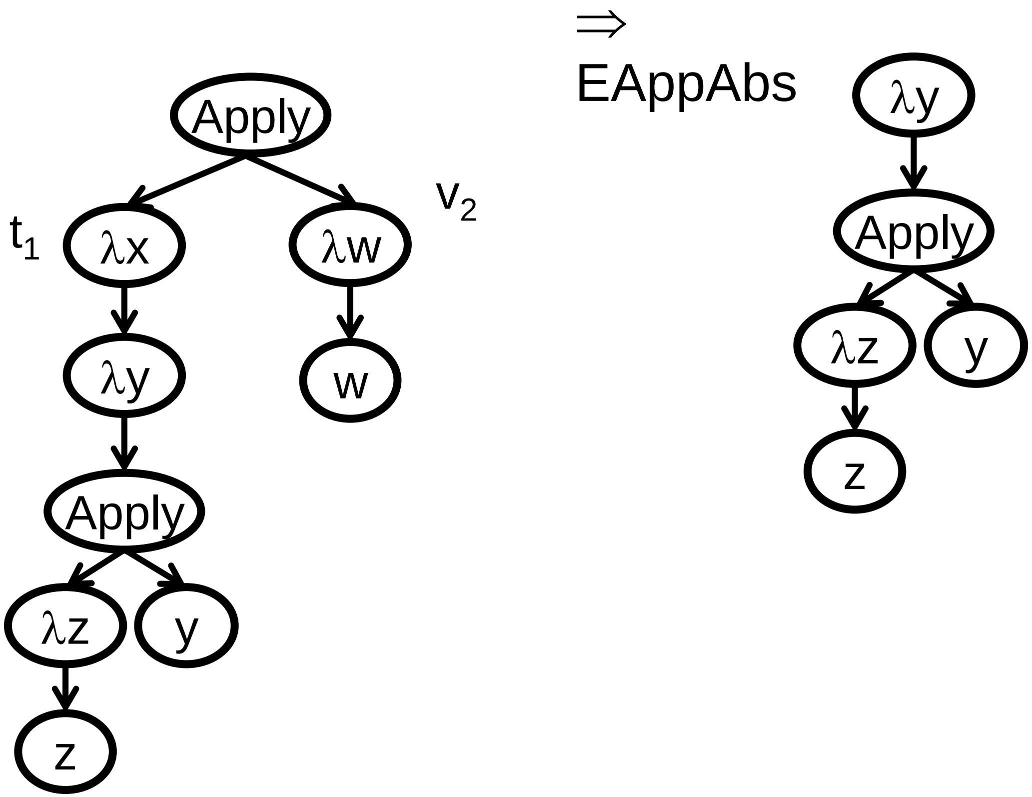
$\Rightarrow$

E-Appl2



Call By Value OS (3)

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



Programming in λ Calculus

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

Programming in the λ 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
λ x. t	abstraction
t t	application
TT | FF | 1 | 2 | …	

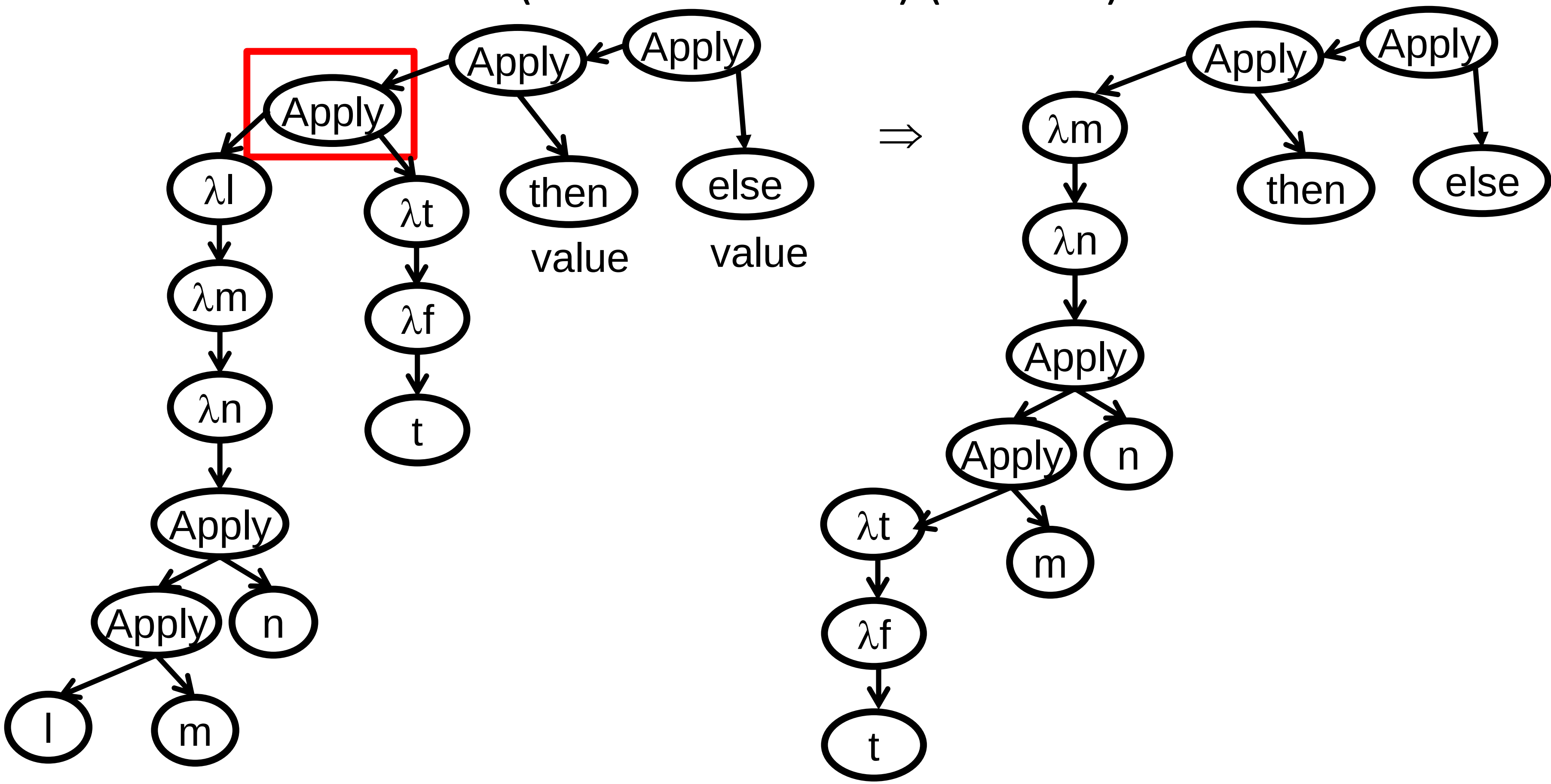
- Can be simulated

Simulating Booleans

- tru = λt. λf. t
- fls = λt. λf. f

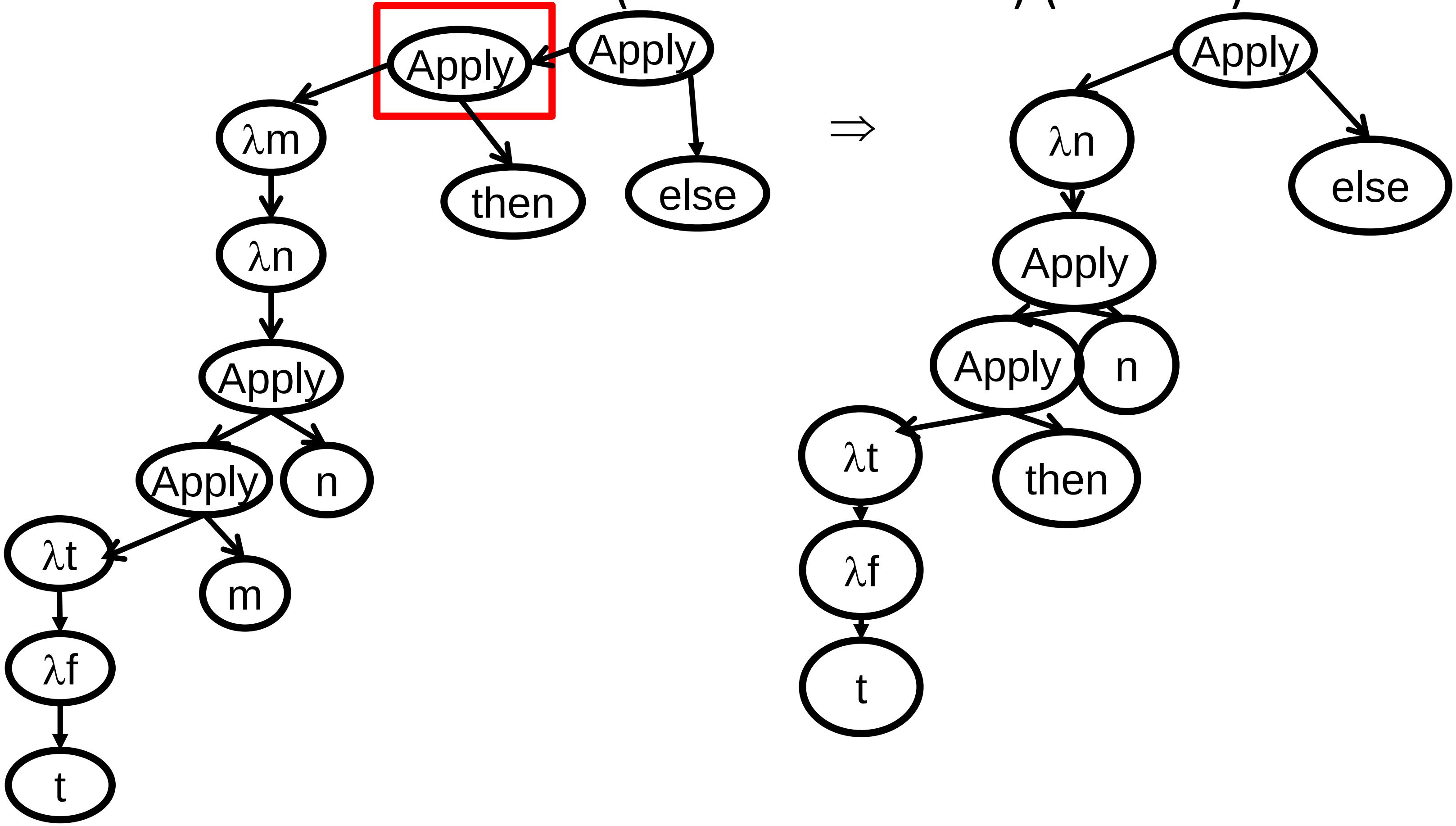
Simulating Tests

- 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) then else



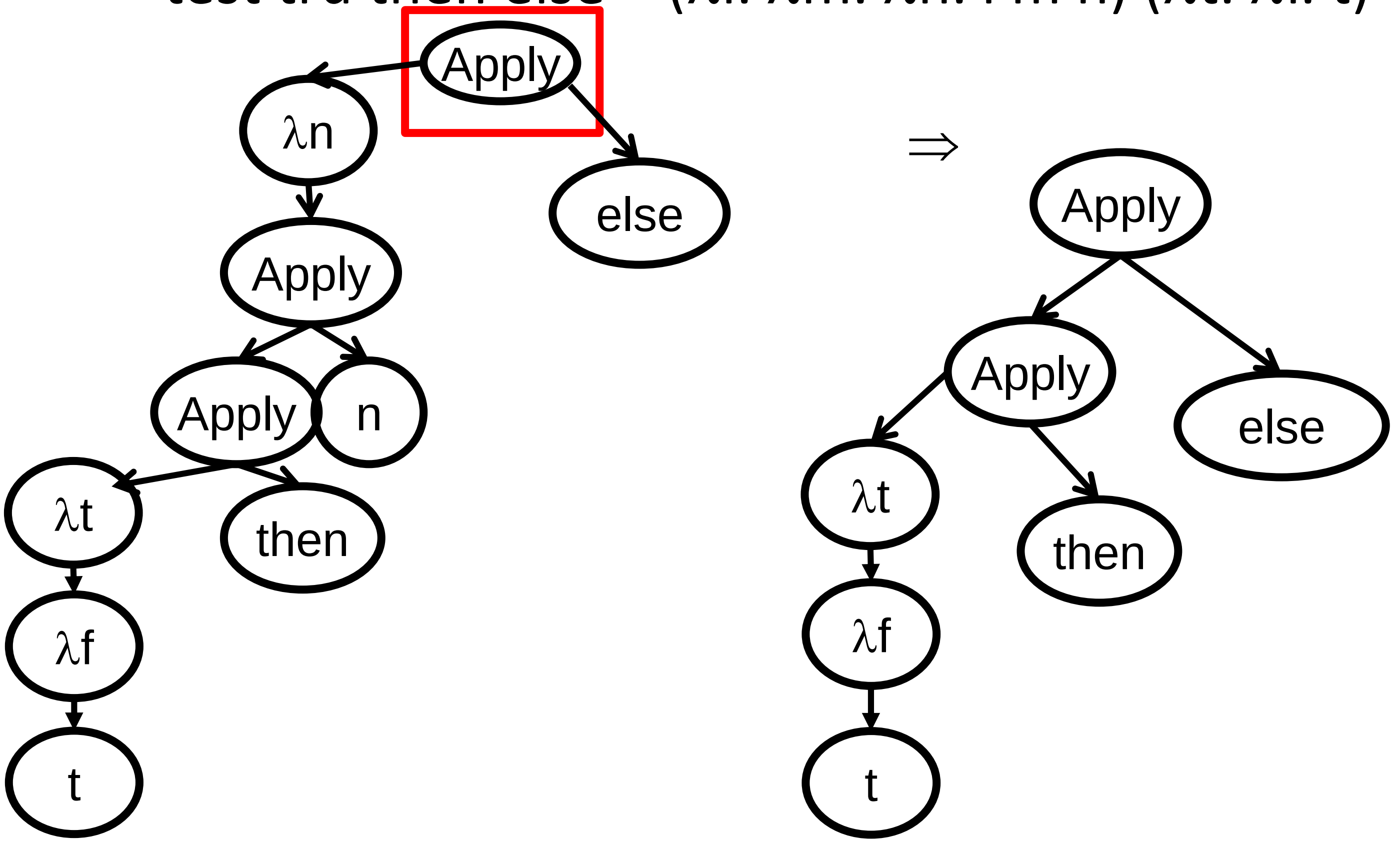
Simulating Tests(2)

- 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) then else



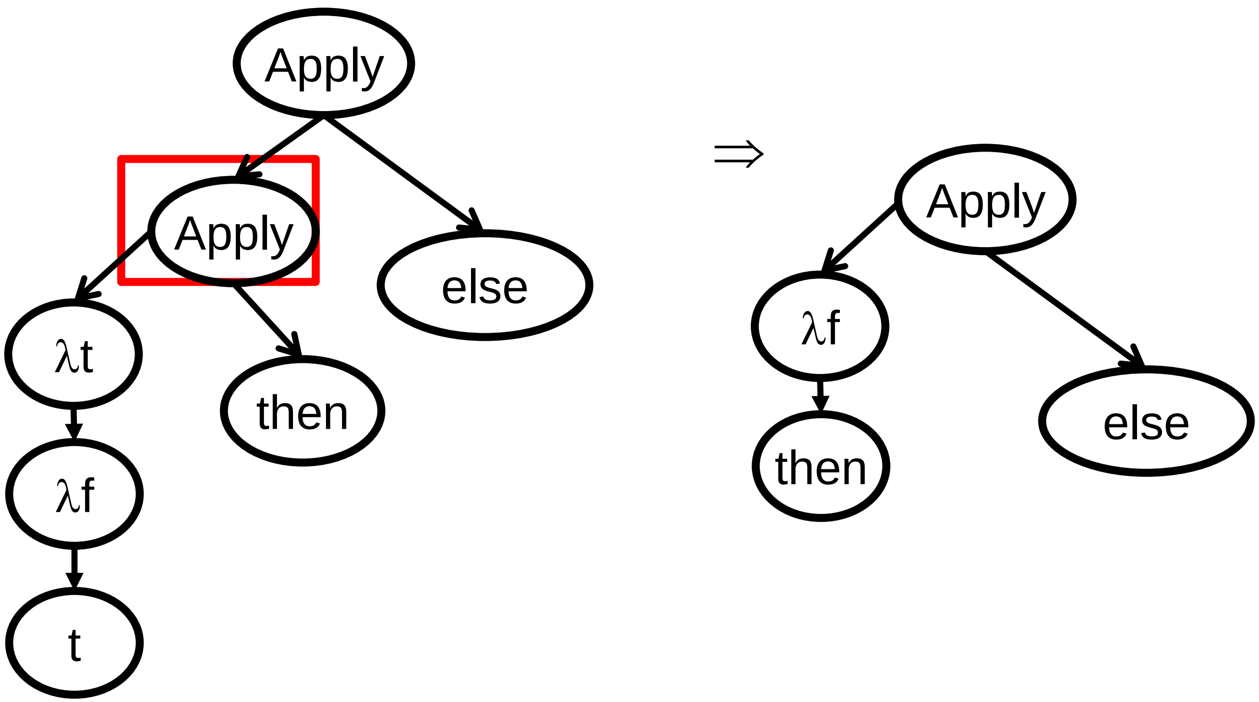
Simulating Tests(3)

- 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) then else



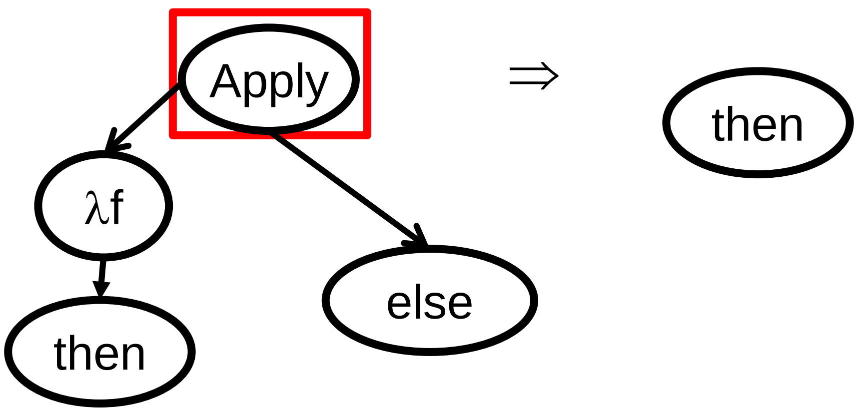
Simulating Tests(4)

- 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) then else



Simulating Tests(5)

- 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) then else



Simulating Tests

- 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)

Programming in λ 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. λb'. test b b' fls
- or = ?
- not = ?

Programming in λ 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

- 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 = (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)

Typed Lambda Calculus

Chapter 9

Benjamin Pierce

Types and Programming Languages

Call-by-value Operational Semantics

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

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

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)

A Naïve Attempt

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

Simple Types

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

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

Explicit vs. Implicit Types

- How to define the type of λ abstractions?
  - Explicit: defined by the programmer

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

T::=	types
$T \to T$	types of functions

Typing Function Declarations

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

A typing context Γ maps free variables into types

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

Typing Free Variables

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

Typing Function Applications

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

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 (λx: Bool. x) else (λy: Bool. not y)

SOS for Simple Typed Lambda Calculus

t ::= terms

x variable

λ x: T. t abstraction

t t application

v::= values

λ x: T. t abstraction values

T::= types

$T \rightarrow T$ types of functions

$t_1 \rightarrow t_2$

$$\frac{t_1 \rightarrow t'_1}{t_1\ t_2 \rightarrow t'_1\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \rightarrow t'_2}{v_1\ t_2 \rightarrow v_1\ t'_2} \quad \text{(E-APP2)}$$

$(\lambda\ x{:}\ T_{11}.\ t_{12})\ v_2 \rightarrow [x \mapsto v_2]\ t_{12}$ (E-APPABS)

Type Rules

$\Gamma \vdash t : T$

t ::= terms

x variable

λ x: T. t abstraction

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

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

T::= types

$T \to T$ types of functions

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

Γ::= context

$\varnothing$ empty context

$\Gamma, x : T$ term variable binding

t ::=	terms
x	variable
$\lambda x{:}\, T.\, t$	abstraction
$t\ t$	application
true	constant true
false	constant false
if t then t else t	conditional
T::=	types
Bool	Boolean type
$T \rightarrow T$	types of functions
Γ::=	context
$\varnothing$	empty context
$\Gamma, x : T$	term variable binding

$\Gamma \vdash t : T$

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

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

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

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

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

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

$t_1 : \text{Bool}\ \ t_2 : T\ \ t_3 : T$

$\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T$

Examples

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

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 Γ (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

- $\Gamma \vdash x : R \Rightarrow x : R \in \Gamma$
- $\Gamma \vdash \lambda x : T_1 .\ t2 : R \Rightarrow R = T_1 \rightarrow R_2$ 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 Δ is a permutation of Γ then $\Delta \vdash$ t : T [Permutation]
- If $\Gamma \vdash$ t : T and x $\notin$dom(Γ) then Γ,t $\vdash$ t : T with a proof of the same depth [Weakening]
- If Γ, 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

x variable

λ x: T. t abstraction

t t application

v::= values

λ x: T. t abstraction values

T::= types

$T \to T$ types of functions

$t_1 \to t_2$

$$\frac{t_1 \to t'_1}{t_1\ t_2 \to t'_1\ t_2} \text{ (E-APP1)}$$

$$\frac{t_2 \to t'_2}{v_1\ t_2 \to v_1\ t'_2} \text{ (E-APP2)}$$

$$(\lambda\, x{:}\ T_{11}.\ t_{12})\ v_2 \to [x \mapsto v_2]\ t_{12} \text{ (E-APPABS)}$$

Erasure and Typability

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

$$\mathrm{erase}(x) = x$$
$$\mathrm{erase}(\lambda x{:}\ T_1.\ t_2) = \lambda x.\mathrm{erase}(t_2)$$
$$\mathrm{erase}(t_1\ t_2) = \mathrm{erase}(t_1)\ \mathrm{erase}(t_2)$$

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

Summary

- Constructive rules for preventing runtime errors in a Turing complete programming language
- Efficient type checking
  - Code is described in Chapter 10
- 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
  - …