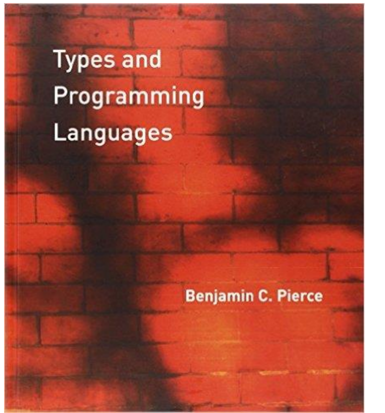
Advanced Topics in Programming Languages
Untyped Lambda Calculus

Oded Padon & Mooly Sagiv

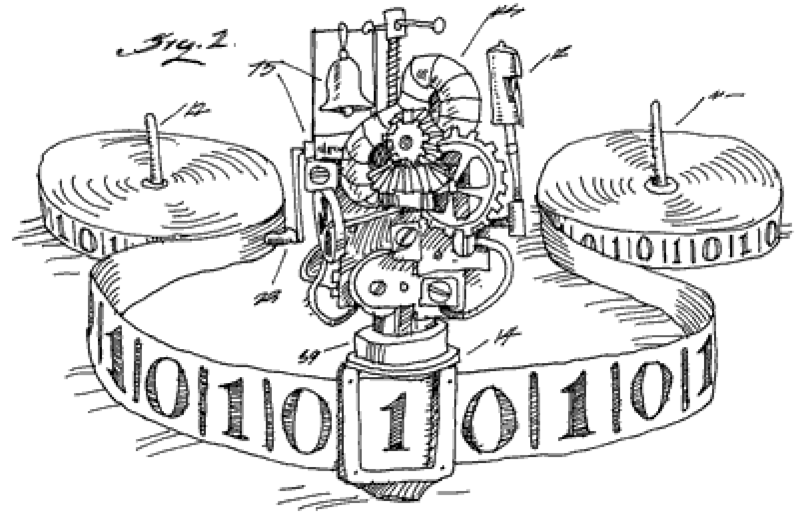
(original slides by Kathleen Fisher, John Mitchell, Shachar Itzhaky, S. Tanimoto)

Reference:
Types and Programming Languages
by Benjamin C. Pierce, Chapter 5



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

Untyped Lambda Calculus - Syntax

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:
    $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ (x\ y)\ x)$
- Examples:
  - (λx. λx. (λx.x) x) ((λx. x x) λx.x)
  - (λt. λf. t) (λx.x) ((λx.x) (λs. λz. s z))

Free vs. Bound Variables

- An occurrence of x in t is **bound** in $\lambda x.\ t$
  - otherwise it is **free**
  - λx is a **binder**
- FV: $t \rightarrow P(Var)$ is the set free variables of t
  - $FV(x) = \{x\}$
  - $FV(\lambda x.\ t) = FV(t) - \{x\}$
  - $FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$
- Examples:
  - $FV(\ x\ (y\ z)\ ) =$
  - $FV(\ \lambda x.\ \lambda y.\ x\ (y\ z)\ ) =$
  - $FV(\ (\lambda x.\ x)\ ) =$
  - $FV(\ (\lambda x.\ x)\ x\ ) =$

Semantics: Substitution, β-reduction, α-conversion

- Substitution

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

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

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

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

- β-reduction

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

- α-conversion

  $(\lambda x.\ t) \Rightarrow_\alpha \lambda y.\ [x \mapsto y]\ t$ if $y \notin FV(t)$

Beta-Reduction: Examples

$\underbrace{(\lambda\ x.\ t_1)\ t_2}_{\text{redex}} \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

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

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

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

Substitution Subtleties

$(\lambda\, x.\, t_1)\; t_2 \Rightarrow_\beta [x \mapsto t_2]\; t_1$ (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 $y \notin FV(s)$**

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

$(\lambda x.\, (\lambda x.\, x))\; y \Rightarrow_\beta [x \mapsto y]\; (\lambda x.\, x) = \lambda x.\, y$?

$(\lambda x.\, (\lambda y.\, x))\; y \Rightarrow_\beta [x \mapsto y]\; (\lambda y.\, x) = \lambda y.\, y$?

$(\lambda x.\, (\lambda x.\, x))\; y$ and $(\lambda x.\, (\lambda y.\, x))\; y$ are stuck! They have no β-reduction

Alpha – Conversion

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

$(\lambda x.\ t) \Rightarrow_{\alpha} \lambda y.\ [x \mapsto y]\ t \quad$ if $y \notin FV(t)$

$$\lambda x.\ x$$

$$\Uparrow_{\alpha}$$

$$(\lambda x.\ (\lambda x.\ x))\ y \Rightarrow_{\alpha} (\lambda x.\ (\lambda z.\ z))\ y \Rightarrow_{\beta} [x \mapsto y]\ (\lambda z.\ z) = \lambda z.\ z \neq \lambda x.\ y$$

$$(\lambda x.\ (\lambda y.\ x))\ y \Rightarrow_{\alpha} (\lambda x.\ (\lambda z.\ x))\ y \Rightarrow_{\beta} [x \mapsto y]\ (\lambda z.\ x) = \lambda z.\ y \neq \lambda y.\ y$$

Examples of β-reduction, α-conversion

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

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \Rightarrow_{\text{"}\beta+\alpha\text{"}}\ u\ r\ (\lambda\ x.\ x)$

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

$(\lambda x.\ (\lambda x.\ x))\ y \Rightarrow_\alpha (\lambda x.\ (\lambda z.\ z))\ y \Rightarrow_\beta \quad \lambda z.\ z$

$(\lambda x.\ (\lambda y.\ x))\ y \Rightarrow_\alpha (\lambda x.\ (\lambda z.\ x))\ y \Rightarrow_\beta \quad \lambda z.\ y$

Non-Deterministic Operational Semantics

(E-AppAbs) $(\lambda\, x.\, t_1)\; t_2 \Rightarrow [x \mapsto t_2]\; t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'} \quad \text{(E-Abs)}$$

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

$$\frac{t_2 \Rightarrow t'_2}{t_1\; t_2 \Rightarrow t_1\; t'_2} \quad \text{(E-App}_2\text{)}$$

Why is this semantics non-deterministic?

Different Evaluation Orders

(E-AppAbs) $(\lambda\, x.\, t_1)\, t_2 \Rightarrow [x \mapsto t_2]\, t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'} \quad \text{(E-Abs)}$$

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

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2} \quad \text{(E-App}_2\text{)}$$

$(\lambda x.\ (\text{add } x\ x))\ (\text{add } 2\ 3) \Rightarrow (\lambda x.\ (\text{add } x\ x))\ (5) \Rightarrow \text{add } 5\ 5 \Rightarrow 10$

$(\lambda x.\ (\text{add } x\ x))\ (\text{add } 2\ 3) \Rightarrow (\text{add } (\text{add } 2\ 3)\ (\text{add } 2\ 3)) \Rightarrow$

$(\text{add } 5\ (\text{add } 2\ 3)) \Rightarrow (\text{add } 5\ 5) \Rightarrow 10$

This example: same final result but lazy performs more computations

Different Evaluation Orders

(E-AppAbs) $(\lambda\, x.\, t_1)\; t_2 \Rightarrow [x \mapsto t_2]\; t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'} \quad \text{(E-Abs)}$$

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

$$\frac{t_2 \Rightarrow t'_2}{t_1\; t_2 \Rightarrow t_1\; t'_2} \quad \text{(E-App}_2\text{)}$$

$(\lambda x.\, \lambda y.\, x)\; 3\; (\text{div } 5\; 0) \Rightarrow$ Exception: Division by zero

$(\lambda x.\, \lambda y.\, x)\; 3\; (\text{div } 5\; 0) \Rightarrow (\lambda y.\, 3)\; (\text{div } 5\; 0) \Rightarrow 3$

This example: lazy suppresses erroneous division and reduces to final result

Can also suppress non-terminating computation.

Many times we want this, for example:

```
if i < len(a) and a[i]==0: print “found zero”
```

Strict	Lazy	Normal Order

Strict

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

precedence

(E-App$_2$)

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$

precedence

(E-AppAbs)

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

Lazy

(E-AppAbs)

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

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

Normal Order

(E-AppAbs)

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

precedence

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

precedence

(E-App$_2$)

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$

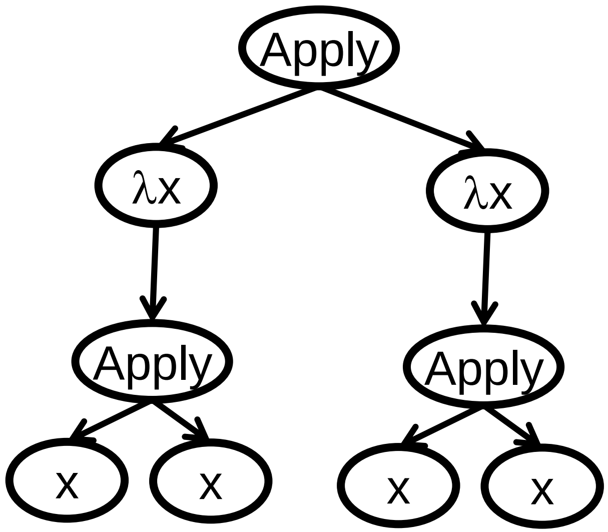
(E-Abs)

$$\frac{t \Rightarrow t'}{\lambda\ x.\ t \Rightarrow \lambda\ x.\ t'}$$

Divergence

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

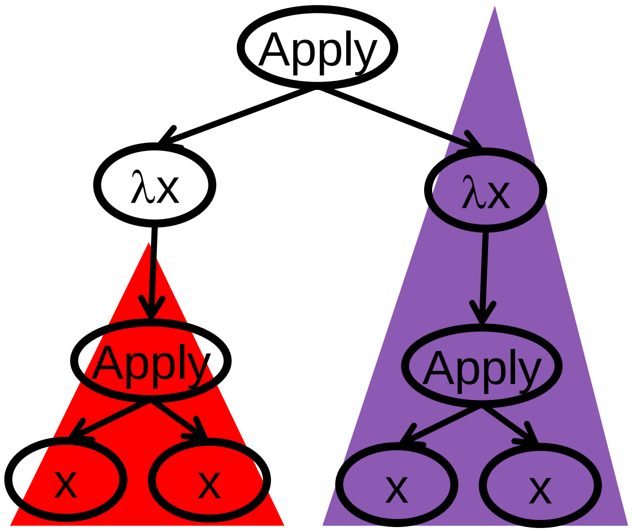
$(\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x))$



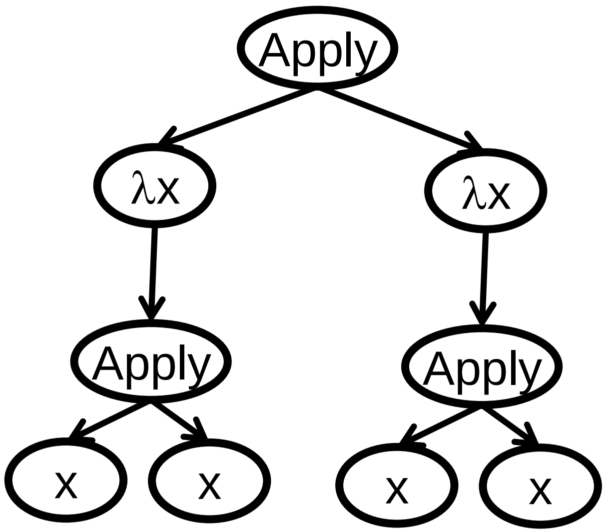
Divergence

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

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



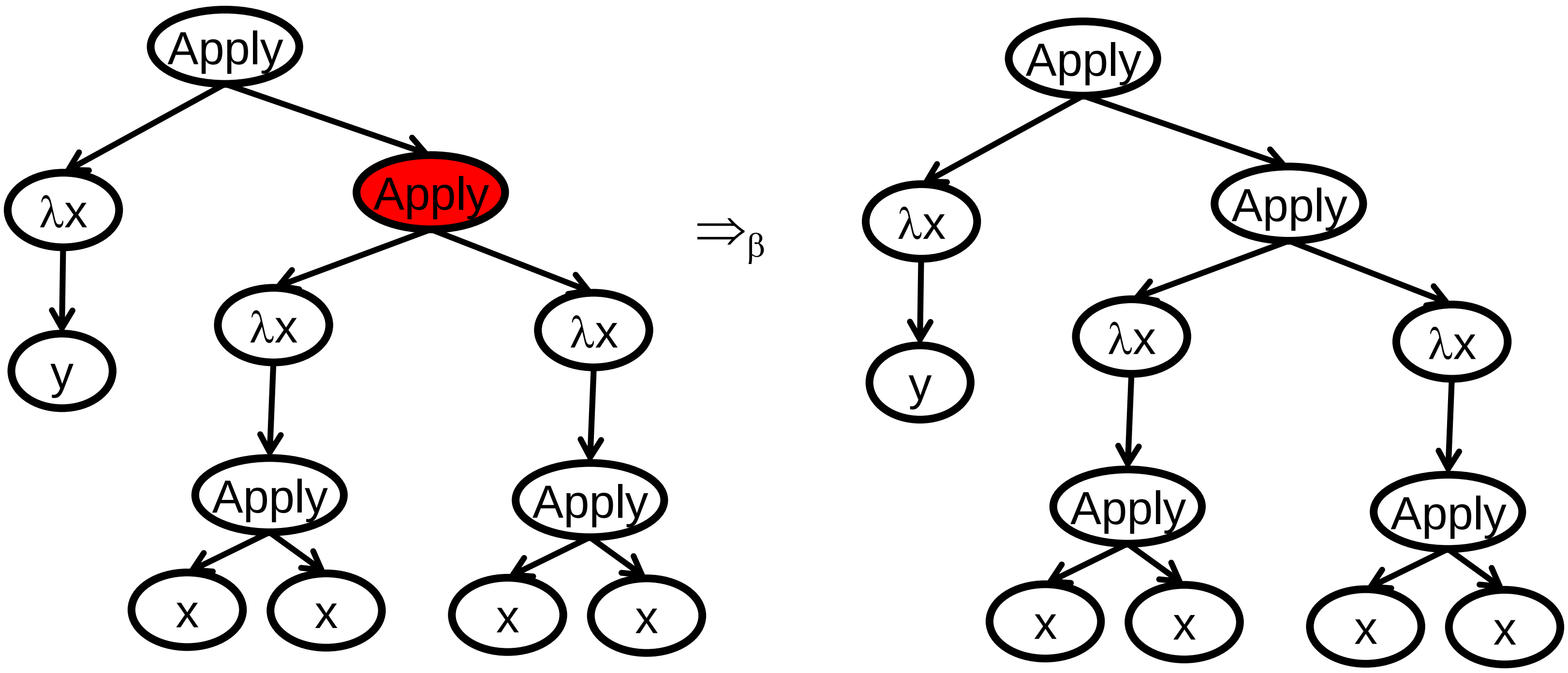
$\Rightarrow_\beta$



Different Evaluation Orders

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

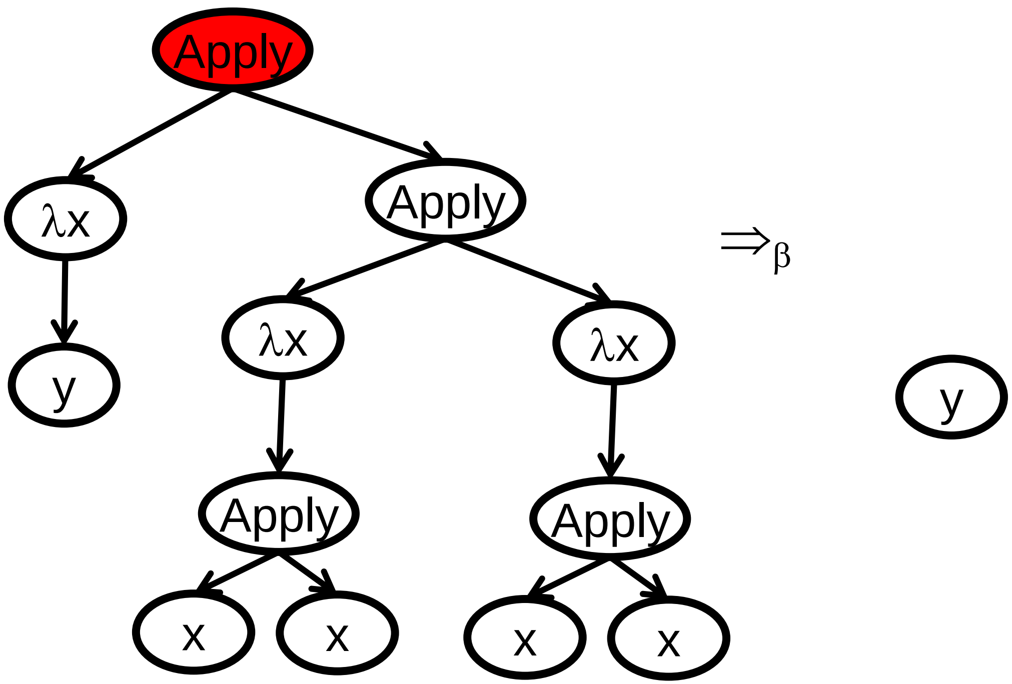
(λ x.y) ((λ x.(x x)) (λ x.(x x)))



Different Evaluation Orders

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

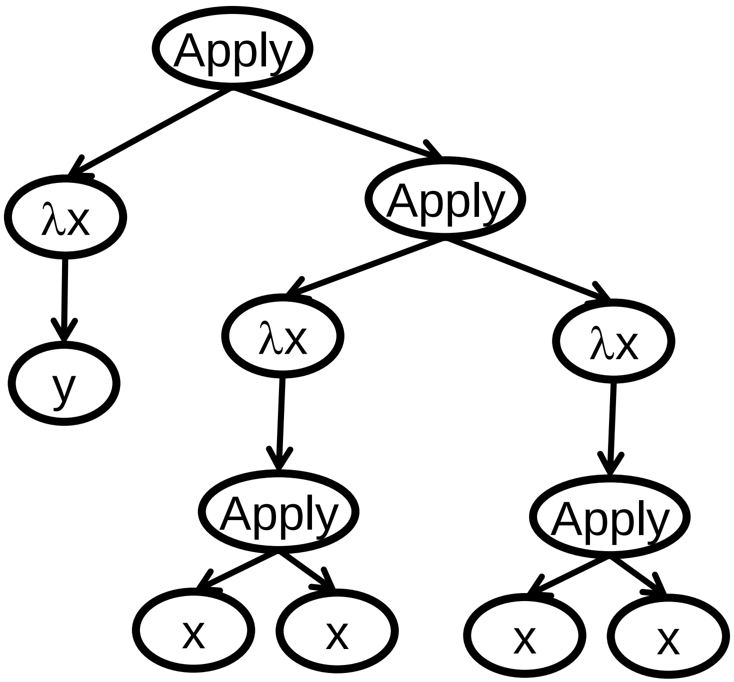
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

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

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



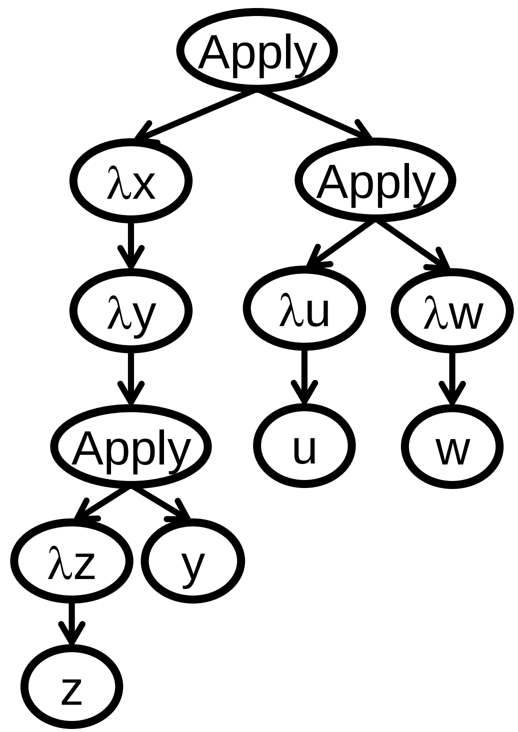

```
def f():
    while True: pass

def g(x):
    return 2

print g(f())
```

Summary Order of Evaluation

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

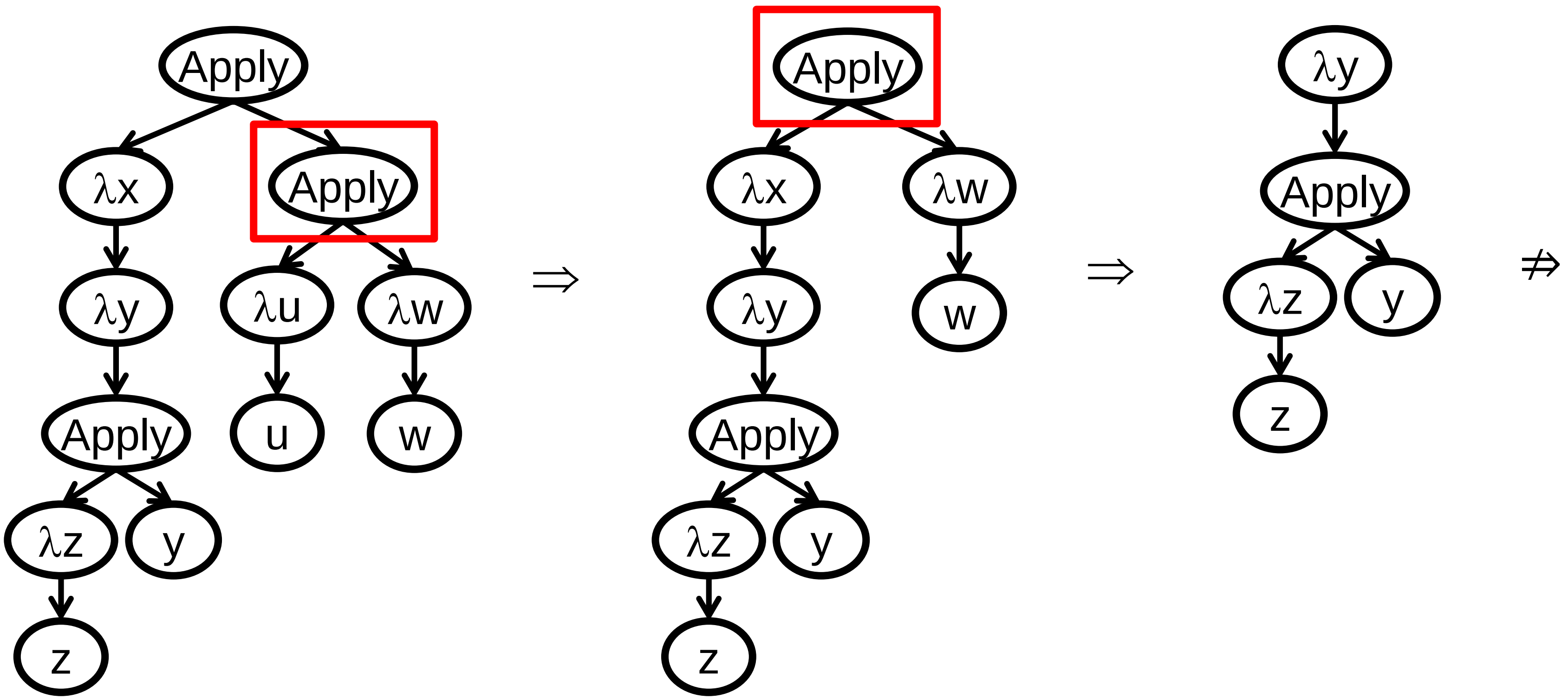
Different Evaluation Orders

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



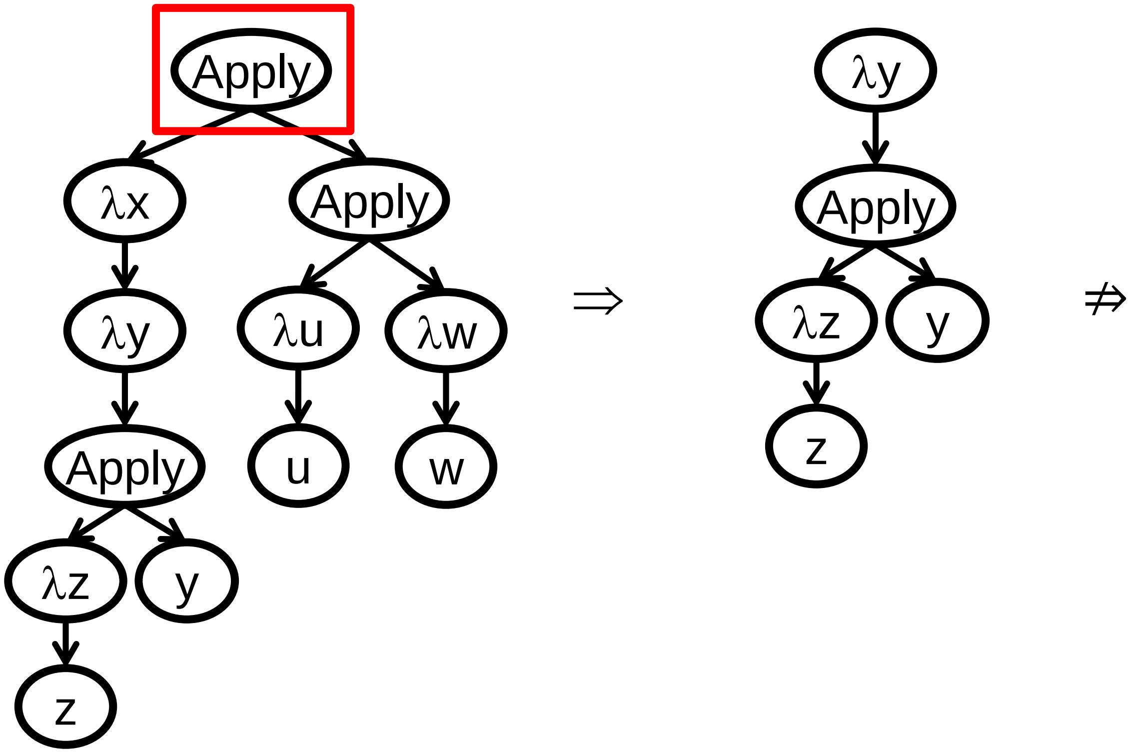
Call By Value

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



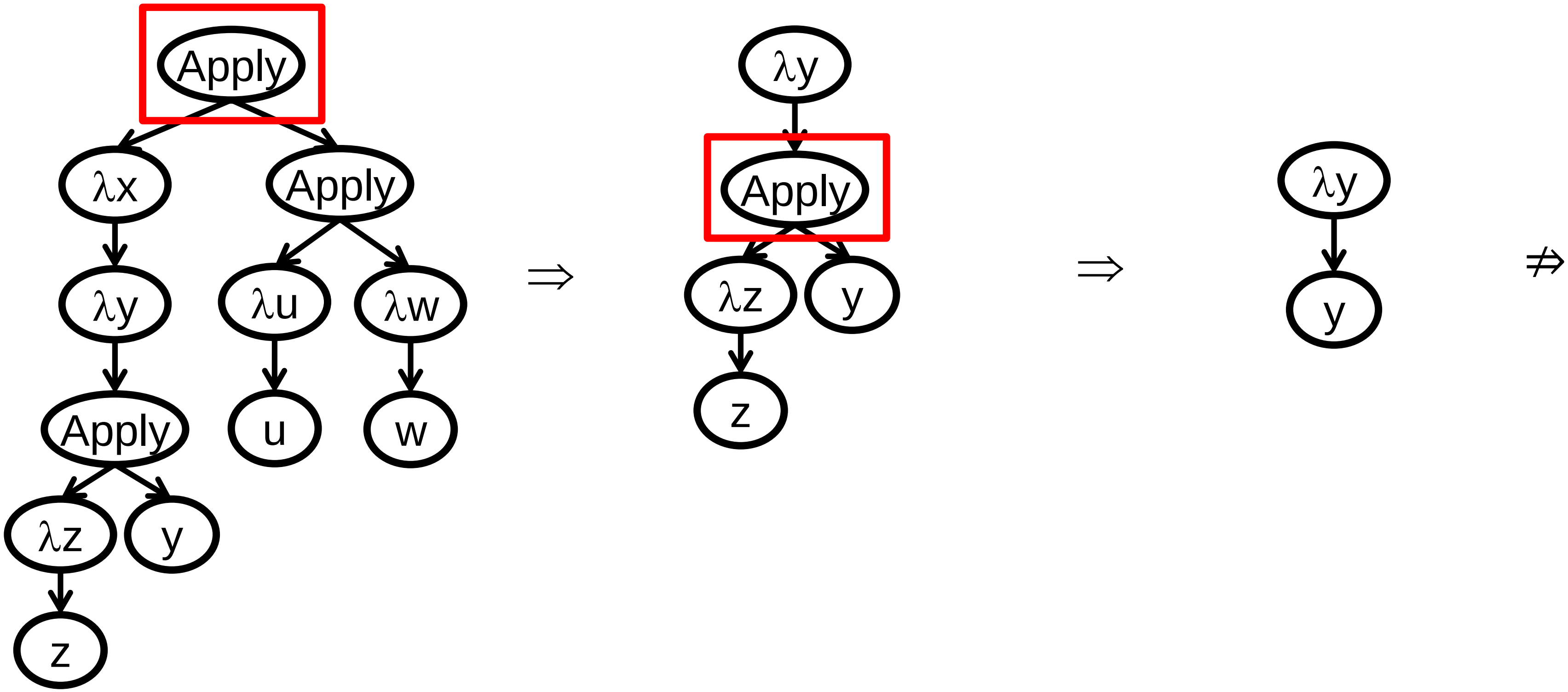
Call By Name (Lazy)

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



Normal Order

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



Currying – Multiple arguments

- Say we want to define a function with two arguments:
  - "$f = \lambda(x, y).\ s$"
- We do this by Currying:
  - $f = \lambda x.\ \lambda y.\ s$
  - f is now "a function of x that returns a function of y"
- Currying and β-reduction:

$$f\ v\ w \quad = (f\ v)\ w = \ ((\lambda x.\ \lambda y.\ s)\ v)\ w$$

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

- Conclusion:
  - "$f = \lambda(x, y).\ s$" → $f = \lambda x.\ \lambda y.\ s$
  - "$f\ (v,w)$" → $f\ v\ w$

Church Booleans

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

  $\Rightarrow(\lambda m.\ \lambda n.\ (\lambda t.\ \lambda f.\ t)\ m\ n)$ then else

  $\Rightarrow(\lambda n.\ (\lambda t.\ \lambda f.\ t)\ \text{then}\ n)$ else

  $\Rightarrow(\lambda t.\ \lambda f.\ t)$ then else

  $\Rightarrow(\lambda f.\ \text{then})$ else

  $\Rightarrow$then
- test fls then else = $(\lambda l.\ \lambda m.\ \lambda n.\ l\ m\ n)\ (\lambda t.\ \lambda f.\ f)$ then else

  $\Rightarrow(\lambda m.\ \lambda n.\ (\lambda t.\ \lambda f.\ f)\ m\ n)$ then else

  $\Rightarrow(\lambda n.\ (\lambda t.\ \lambda f.\ f)\ \text{then}\ n)$ else

  $\Rightarrow(\lambda t.\ \lambda f.\ f)$ then else

  $\Rightarrow(\lambda f.\ f)$ else

  $\Rightarrow$else
- and = $\lambda b.\ \lambda c.\ b\ c$ fls
- or =
- not =

Church Numerals

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

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

Iteration in Lambda Calculus

- omega = (λx. x x) (λx. x x)
  – (λx. x x) (λx. x x) ⇒ (λx. x x) (λx. x x)

Y Combinator

Williams + Hirakawa

- Y = λf. (λx. f (x x)) (λx. f (x x))
- Z = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
- Recursion can be simulated
  – Y only works with call-by-name semantics
  – Z works with call-by-value semantics
- Defining factorial:
  – g = λf. λn. if n==0 then 1 else (n * (f (n - 1)))
  – fact = Y g (for call-by-name)
  – fact = Z g (for call-by-value)

Y-Combinator in action (lazy)

“$g = \lambda f.\ \lambda n.\ \text{if } n{=}{=}0 \text{ then } 1 \text{ else } (n * (f\ (n - 1)))$”

$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$

$Y\ g\ v = (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g\ v$

$\Rightarrow ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))\ v$

$\Rightarrow (g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x)))\ )\ v$

$\sim\ (g\ (Y\ g))\ v$

What happens to Y in strict semantics?

Z-Combinator in action (strict)

“g = λf. λn. if n==0 then 1 else (n * (f (n - 1)))”

Z = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

Z g v = (λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))) g v

⇒((λx. g (λy. x x y)) (λx. g (λy. x x y))) v

⇒(g (λy. (λx. g (λy. x x y)) (λx. g (λy. x x y)) y)) v

~ (g (λy. (Z g) y)) v

~ (g (Z g)) v

```
def f1(y):
    return f2(y)
```

Simulating laziness like Z-Combinator

```
def f(x):
    if ask_user("wanna see it?"):
        print x

def g(x, y, z):
    # very expensive computation without side effects

def main():
    # compute a, b, c with side effects
    f(g(a, b, c))
```

- In strict semantics, the above code computes g anyway
  - Lazy will avoid it
- How can achieve this in a strict programming language?

Simulating laziness like Z-Combinator

```
def f(x):
    if ask_user(“?"):
        print x

def g(x, y, z):
    # expensive

def main():
    # compute a, b, c
    f(g(a, b, c))
```

```
def f(x):
    if ask_user("?"):
        print x()

def g(x, y, z):
    # expensive

def main():
    # compute a, b, c
    f(lambda: g(a, b, c))
```

$$Z = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

~~(E-Abs)~~

$$\frac{t \Rightarrow t'}{\lambda x.\ t \Rightarrow \lambda x.\ t'}$$

Church–Rosser Theorem

If:

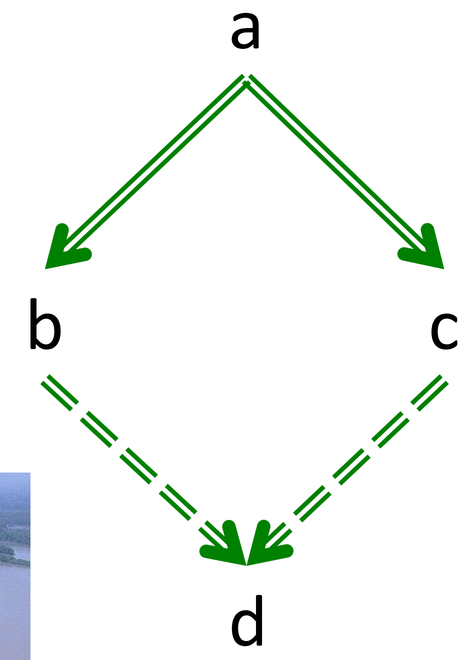
$a \Rightarrow^* b$,

$a \Rightarrow^* c$

then there exists d such that:

$b \Rightarrow^* d$, and

$c \Rightarrow^* d$



Normal Form & Halting Problem

- A term is in normal form if it is stuck in normal order semantics
- Under normal order every term either:
  - Reduces to normal form, or
  - Reduces infinitely
- For a given term, it is undecidable to decide which is the case