Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

If Then Else Command

$$(\text{if true then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (C, m)$$

$$(\text{if false then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (C', m)$$

$$\frac{(B,m) \dashrightarrow (B',m)}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (\text{if } B' \text{ then } C \text{ else } C' \text{ fi}, m)}$$

While Command

(while B do C od, m)

--> (if B then C; while B do C od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.

Example Evaluation

- First step:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,

{x -> 7})

--> ?

Example Evaluation

- First step:

$$\frac{(x > 5, \{x \to 7\}) \dashrightarrow ?}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \to 7\}) \dashrightarrow ?}$$

Example Evaluation

- First step:

$$\frac{\dfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } ?}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \text{ --> } ?}$$

Example Evaluation

- First step:

$$\dfrac{\dfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } (7 > 5, \{x \text{ -> } 7\})}}{(\text{if } x > 5 \text{ then } y{:}= 2 + 3 \text{ else } y{:}=3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \text{ --> } ?}$$

Example Evaluation

- First step:

$$\dfrac{\dfrac{(x,\{x \text{ -> } 7\}) \text{ --> } (7, \{x \text{ -> } 7\})}{(x > 5, \{x \text{ -> } 7\}) \text{ --> } (7 > 5, \{x \text{ -> } 7\})}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\})}$$

$$\text{-->} (\text{if } 7 > 5 \text{ then } y{:=}2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\})$$

Example Evaluation

- Second Step:

$$\frac{(7 > 5, \{x \text{ -> } 7\}) \text{ --> } (\text{true}, \{x \text{ -> } 7\})}{\begin{array}{c}(\text{if } 7 > 5 \text{ then } y{:=}2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \\ \{x \text{ -> } 7\}) \\ \text{--> } (\text{if true then } y{:=}2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \\ \{x \text{ -> } 7\})\end{array}}$$

- Third Step:

(if true then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})

-->(y:=2+3, {x->7})

Example Evaluation

- Fourth Step:

$$\frac{(2+3, \{x\text{-}>7\}) \text{ --> } (5, \{x \text{ -> } 7\})}{(y:=2+3, \{x\text{-}>7\}) \text{ --> } (y:=5, \{x\text{-}>7\})}$$

- Fifth Step:

(y:=5, {x->7}) --> {y -> 5, x -> 7}

Example Evaluation

- Bottom Line:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})

--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

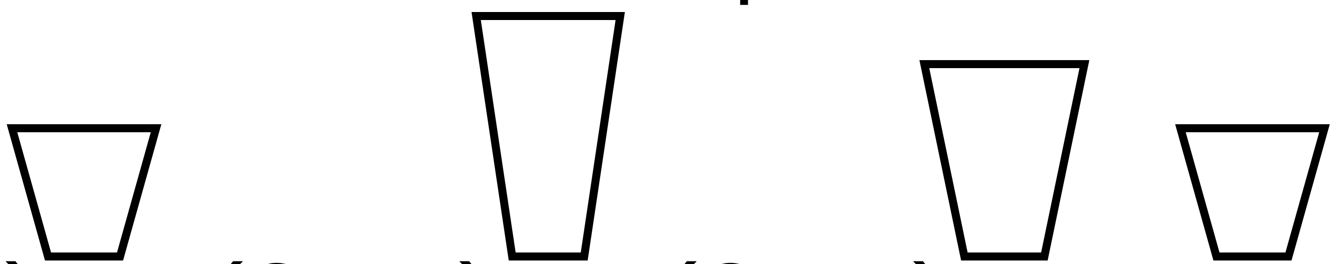
-->(if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

-->(y:=2+3, {x->7})

--> (y:=5, {x->7}) --> {y -> 5, x -> 7}

Transition Semantics Evaluation

- A sequence of steps with trees of justification for each step



$$(C_1,m_1) \text{ --> } (C_2,m_2) \text{ --> } (C_3,m_3) \text{ --> } \ldots \text{ --> } m$$

- Let -->* be the transitive closure of -->
- Ie, the smallest transitive relation containing -->

Adding Local Declarations

- Add to expressions:
- E ::= … | let I = E in E' | fun I -> E | E E'
- fun I -> E is a value
- Could handle local binding using state, but have assumption that evaluating expressions doesn't alter the environment
- We will use substitution here instead
- **Notation**: $E\,[E' / I\,]$ means replace all free occurrence of I by E' in E

Call-by-value (Eager Evaluation)

$$(\text{let } I = V \text{ in } E, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E, m) \dashrightarrow (E'', m)}{(\text{let } I = E \text{ in } E', m) \dashrightarrow (\text{let } I = E'' \text{ in } E')}$$

$$((\text{fun } I \to E)\ V, m) \dashrightarrow (E[V/I], m)$$

$$\frac{(E', m) \dashrightarrow (E'', m)}{((\text{fun } I \to E)\ E', m) \dashrightarrow ((\text{fun } I \to E)\ E'', m)}$$

Call-by-name (Lazy Evaluation)

- (let I = E in E', m) --> ($E'[E / I]$,m)

- ((fun I -> E') E, m) --> ($E'[E / I]$,m)

- Question: Does it make a difference?
- It can depending on the language

Church-Rosser Property

- Church-Rosser Property: If E-->* E_1 and E-->* E_2, if there exists a value V such that E_1 -->* V, then E_2 -->* V
- Also called **confluence** or **diamond property**
- Example:

$$\begin{tikzcd}[column sep=small] & E = 2 + 3 + 4 \arrow[dl] \arrow[dr] & \\ E_1 = 5 + 4 \arrow[dr] & & E_2 = 2 + 7 \arrow[dl] \\ & V = 9 & \end{tikzcd}$$

Does It always Hold?

- No. Languages with side-effects tend not be Church-Rosser with the combination of call-by-name and call-by-value
- Alonzo Church and Barkley Rosser proved in 1936 the λ-calculus does have it
- Benefit of Church-Rosser: can check equality of terms by evaluating them (Given evaluation strategy might not terminate, though)

Lambda Calculus - Motivation

- Aim is to capture the essence of functions, function applications, and evaluation

- λ−calculus is a theory of computation

- "The Lambda Calculus: Its Syntax and Semantics". H. P. Barendregt. North Holland, 1984

Lambda Calculus - Motivation

- All *sequential programs* may be viewed as functions from input (initial state and input values) to output (resulting state and output values).
- λ-calculus is a mathematical formalism of functions and functional computations
- Two flavors: typed and untyped

Untyped λ-Calculus

- Only three kinds of expressions:
  - Variables: x, y, z, w, …
  - Abstraction: λ x. e

    (Function creation, think fun x -> e)
  - Application: $e_1\ e_2$

Untyped λ-Calculus Grammar

- Formal BNF Grammar:
  - <expression> ::= <variable>
| <abstraction>
| <application>
| (<expression>)
  - <abstraction>
::= λ<variable>.<expression>
  - <application>
::= <expression> <expression>

Untyped λ-Calculus Terminology

- Occurrence: a location of a subterm in a term
- Variable binding: λ x. e is a binding of x in e
- Bound occurrence: all occurrences of x in λ x. e
- Free occurrence: one that is not bound
- Scope of binding: in λ x. e, all occurrences in e not in a subterm of the form λ x. e′ (same x)
- Free variables: all variables having free occurrences in a term

Example

- Label occurrences and scope:

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

Example

- Label occurrences and scope:

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

Untyped λ-Calculus

- How do you compute with the λ-calculus?
- Roughly speaking, by substitution:
- $(\lambda\ x.\ e_1)\ e_2 \Rightarrow^* e_1\ [e_2 / x]$
- * Modulo all kinds of subtleties to avoid free variable capture

Transition Semantics for λ-Calculus

$$\frac{E \dashrightarrow E''}{E\ E' \dashrightarrow E''\ E'}$$

- Application (version 1 - Lazy Evaluation)

$$(\lambda\ x\ .\ E)\ E' \dashrightarrow E[E'/x]$$

- Application (version 2 - Eager Evaluation)

$$\frac{E' \dashrightarrow E''}{(\lambda\ x\ .\ E)\ E' \dashrightarrow (\lambda\ x\ .\ E)\ E''}$$

$$\frac{}{(\lambda\ x\ .\ E)\ V \dashrightarrow E[V/x]}$$

V - variable or abstraction (value)

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is Turing Complete
  - Can express any sequential computation
- Problems:
  - How to express basic data: booleans, integers, etc?
  - How to express recursion?
  - Constants, if_then_else, etc, are conveniences; can be added as syntactic sugar

Typed vs Untyped λ-Calculus

- The *pure* λ-calculus has no notion of type: (f f) is a legal expression
- Types restrict which applications are valid
- Types are not syntactic sugar! They disallow some terms
- Simply typed λ-calculus is less powerful than the untyped λ-Calculus: NOT Turing Complete (no recursion)

Uses of λ-Calculus

- Typed and untyped λ-calculus used for theoretical study of sequential programming languages
- Sequential programming languages are essentially the λ-calculus, extended with predefined constructs, constants, types, and syntactic sugar
- Ocaml is close to the λ-Calculus:

$$\text{fun x -> exp} \;\; \text{-->} \;\; \lambda\, x.\ \text{exp}$$

$$\text{let } x = e_1 \text{ in } e_2 \text{ --> } (\lambda\, x.\ e_2)e_1$$

α Conversion

- α-conversion:

  λ x. exp --α--> λ y. (exp [y/x])
- Provided that
  1. y is not free in exp
  2. No free occurrence of x in exp becomes bound in exp when replaced by y

α Conversion Non-Examples

1. Error: y is not free in termsecond

$$\lambda\, x.\ x\ y \ \cancel{\text{--}\alpha\text{-->}}\ \lambda\, y.\ y\ y$$

2. Error: free occurrence of x becomes bound in wrong way when replaced by y

$$\lambda\, x.\ \underbrace{\lambda\, y.\ x\ y}_{\text{exp}} \ \cancel{\text{--}\alpha\text{-->}}\ \lambda\, y.\ \underbrace{\lambda\, y.\ y\ y}_{\text{exp[y/x]}}$$

But λ x. (λ y. y) x --α--> λ y. (λ y. y) y

And λ y. (λ y. y) y --α--> λ x. (λ y. y) x

Congruence

- Let ~ be a relation on lambda terms. ~ is a congruence if
- it is an equivalence relation
- If $e_1 \sim e_2$ then
  - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
  - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$

α Equivalence

- α equivalence is the smallest congruence containing α conversion

- One usually treats α-equivalent terms as equal - i.e. use α equivalence classes of terms

Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

- λ x. (λ y. y x) x --α--> λ z. (λ y. y z) z so
  λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z
- (λ y. y z) --α--> (λ x. x z) so
  (λ y. y z) ~α~ (λ x. x z) so
  λ z. (λ y. y z) z ~α~ λ z. (λ x. x z) z
- λ z. (λ x. x z) z --α--> λ y. (λ x. x y) y so
  λ z. (λ x. x z) z ~α~ λ y. (λ x. x y) y
- λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

Substitution

- Defined on α-equivalence classes of terms
- P [N / x] means replace every free occurrence of x in P by N
  - P called *redex*; N called *residue*
- Provided that no variable free in P becomes bound in P [N / x]
  - Rename bound variables in P to avoid capturing free variables of N

Substitution

- $x\ [N / x] = N$
- $y\ [N / x] = y$ if $y \neq x$
- $(e_1\ e_2)\ [N / x] = ((e_1\ [N / x])\ (e_2\ [N / x]))$
- $(\lambda\ x.\ e)\ [N / x] = (\lambda\ x.\ e)$
- $(\lambda\ y.\ e)\ [N / x] = \lambda\ y.\ (e\ [N / x])$ provided $y \neq x$ and y not free in N
  - Rename y in redex if necessary

Example

$$(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] = ?$$

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- $(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z]$ --α-->
  $(\lambda\ w.w\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] =$
  $\lambda\ w.\ w\ (\lambda\ x.\ x\ y)$

Example

- Only replace free occurrences
- $(\lambda\ y.\ y\ z\ (\lambda\ z.\ z))\ [(\lambda\ x.\ x)\ /\ z] =$

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

Not

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

β reduction

- β Rule: $(\lambda\ x.\ P)\ N$ --β--> $P\ [N\ /x]$
- Essence of computation in the lambda calculus
- Usually defined on α-equivalence classes of terms

Example

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

  $\text{--}\beta\text{-->}\ (\lambda\ x.\ x\ y)\ (\lambda\ y.\ y\ z)$

  $\text{--}\beta\text{-->}\ (\lambda\ y.\ y\ z)\ y\ \text{--}\beta\text{-->}\ y\ z$

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

  $\text{--}\beta\text{-->}\ (\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)$

  $\text{--}\beta\text{-->}\ (\lambda\ x.\ x\ x)\ (\lambda\ x.\ x\ x)\ \text{--}\beta\text{-->}\ \ldots.$

α β Equivalence

- α β equivalence is the smallest congruence containing α equivalence and β reduction
- A term is in *normal form* if no subterm is α equivalent to a term that can be β reduced
- Hard fact (Church-Rosser): if e_1 and e_2 are $\alpha\beta$-equivalent and both are normal forms, then they are α equivalent

Order of Evaluation

- Not all terms reduce to normal forms

- Not all reduction strategies will produce a normal form if one exists

Lazy evaluation:

- Always reduce the left-most application in a top-most series of applications (i.e. Do not perform reduction inside an abstraction)
- Stop when term is not an application, or left-most application is not an application of an abstraction to a term

Example 1

- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$
- Lazy evaluation:
- Reduce the left-most application:
- $(\lambda\ z.\ (\lambda\ x.\ x))\ ((\lambda\ y.\ y\ y)\ (\lambda\ y.\ y\ y))$ $\text{--}\beta\text{-->}$ $(\lambda\ x.\ x)$

Eager evaluation

- (Eagerly) reduce left of top application to an abstraction
- Then (eagerly) reduce argument
- Then β-reduce the application

Example 1

- (λ z. (λ x. x))((λ y. y y) (λ y. y y))
- Eager evaluation:
- Reduce the rator of the top-most application to an abstraction: Done.
- Reduce the argument:
- (λ z. (λ x. x))((λ y. y y) (λ y. y y))

--β--> (λ z. (λ x. x))((λ y. y y) (λ y. y y))

--β--> (λ z. (λ x. x))((λ y. y y) (λ y. y y))…

Example 2

- $(\lambda$ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ \boxed{x}\ \boxed{x}\ )(\underline{(\lambda\ y.\ y\ y)\ (\lambda\ z.\ z)})$ --β-->

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. $\boxed{x}$ $\boxed{x}$)$\underline{((λ y. y y) (λ z. z))}$ --β-->

$\boxed{((λ y. y\ \ y) (λ z. z))}$ $\boxed{((λ y. y\ \ y) (λ z. z))}$

Example 2

- $(\lambda\, x.\ x\ x)((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$
- Lazy evaluation:

$(\lambda\, x.\ x\ \ x\ )((\lambda\, y.\ y\ y)\ (\lambda\, z.\ z))$ --β-->

$\boxed{((\lambda\, y.\ y\ \ y\ )\ (\lambda\, z.\ z))}\ ((\lambda\, y.\ y\ \ y\ )\ (\lambda\, z.\ z)$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ \boxed{y}\ \boxed{y}\ )\ \underline{(\lambda\ z.\ z)})\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

Example 2

- $(\lambda x.\ x\ x)((\lambda y.\ y\ y)\ (\lambda z.\ z))$
- Lazy evaluation:

$(\lambda x.\ x\ \ x)((\lambda y.\ y\ y)\ (\lambda z.\ z))$ --β-->

$((\lambda y.\ \boxed{y}\ \boxed{y})\ \underline{(\lambda z.\ z)})\ ((\lambda y.\ y\ \ y)\ (\lambda z.\ z))$

--β--> $(\boxed{(\lambda z.\ z)}\ \boxed{(\lambda z.\ z)})((\lambda y.\ y\ \ y)\ (\lambda z.\ z))$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $\boxed{((\lambda\ z.\ z\ )\ (\lambda\ z.\ z))}((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $((\lambda\ z.\ \boxed{z}\ )\ \underline{(\lambda\ z.\ z)})((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $((\lambda\ z.\ \boxed{z}\ )\ \underline{(\lambda\ z.\ z)})((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $\boxed{(\lambda\ z.\ z\ )}\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

Example 2

- $(\lambda\ x.\ x\ x)((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$
- Lazy evaluation:

$(\lambda\ x.\ x\ \ x\ )((\lambda\ y.\ y\ y)\ (\lambda\ z.\ z))$ --β-->

$((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))\ ((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $((\lambda\ z.\ z\ )\ (\lambda\ z.\ z))((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))$

--β--> $(\lambda\ z.\ \boxed{z}\ )\ \underline{((\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z))}$ --β-->

$(\lambda\ y.\ y\ \ y\ )\ (\lambda\ z.\ z)$

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z)

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Lazy evaluation:

(λ x. x x)((λ y. y y) (λ z. z)) --β-->

((λ y. y y) (λ z. z)) ((λ y. y y) (λ z. z))

--β--> ((λ z. z) (λ z. z))((λ y. y y) (λ z. z))

--β--> (λ z. z) ((λ y. y y) (λ z. z)) --β-->

(λ y. y y) (λ z. z) ~β~ λ z. z

Example 2

- (λ x. x x)((λ y. y y) (λ z. z))
- Eager evaluation:

(λ x. x x) $\boxed{\text{((λ y. y y) (λ z. z))}}$ --β-->

(λ x. x x) $\boxed{\text{((λ z. z) (λ z. z))}}$ --β-->

(λ x. x x) $\boxed{\text{(λ z. z)}}$ --β-->

(λ z. z) (λ z. z) --β--> λ z. z

η (Eta) Reduction

- η Rule: λ x. f x --η--> f if x not free in f
  - Can be useful in each direction
  - Not valid in Ocaml
    - recall lambda-lifting and side effects
  - Not equivalent to (λ x. f) x --> f (inst of β)

- Example: λ x. (λ y. y) x --η--> λ y. y