

# Lecture 4 – Evaluation Contexts and Lambda Calculus

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Small-step operational semantics
  - Example: VAE
  - Multi-Step Evaluation
  - Derivation
  - Big-step vs. Small-step Operational Semantics
- Soundness
  - Well-Formed Configurations
  - Progress
  - Preservation
  - Termination
  - Soundness vs. Completeness
- Simple Imperative Language – IMP
  - Expressions
  - Statements

## 1. Evaluation Contexts

## 2. Lambda Calculus

$\alpha$ -Equivalence

$\beta$ -Reduction

Call by-Value (CBV) vs. Call by-Name (CBN)

Substitution

De Bruijn Indices

## 3. Definitional Translation

Church Encoding

Pairs and Let Bindings

Laziness

Adequacy

## 1. Evaluation Contexts

## 2. Lambda Calculus

$\alpha$ -Equivalence

$\beta$ -Reduction

Call by-Value (CBV) vs. Call by-Name (CBN)

Substitution

De Bruijn Indices

## 3. Definitional Translation

Church Encoding

Pairs and Let Bindings

Laziness

Adequacy

$$e ::= n \mid e + e \mid e * e \mid x := e; e \mid x$$

The small-step operational semantics of VAE is defined by the following **small-step evaluation relation**  $\rightarrow$  on configurations  $\langle \sigma, e \rangle$ :

$$\boxed{\langle \sigma, e \rangle \rightarrow \langle \sigma, e \rangle}$$

where a **configuration**  $\langle \sigma, e \rangle$  consists of

- 1 an environment  $\sigma : \mathbb{X} \rightarrow \mathbb{Z}$  that maps variables to integers, and
- 2 an expression  $e$ .

There are two kinds of rules: **congruence rules** that specify evaluation order, and **computation rules** that specify how to perform computations.

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle}{\langle \sigma, e_1 + e_2 \rangle \rightarrow \langle \sigma', e'_1 + e_2 \rangle}$$

$$\frac{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e'_2 \rangle}{\langle \sigma, n_1 + e_2 \rangle \rightarrow \langle \sigma', n_1 + e'_2 \rangle}$$

$$\langle \sigma, n_1 + n_2 \rangle \rightarrow \langle \sigma, n_1 + n_2 \rangle$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle}{\langle \sigma, e_1 * e_2 \rangle \rightarrow \langle \sigma', e'_1 * e_2 \rangle}$$

$$\frac{\langle \sigma, e_2 \rangle \rightarrow \langle \sigma', e'_2 \rangle}{\langle \sigma, n_1 * e_2 \rangle \rightarrow \langle \sigma', n_1 * e'_2 \rangle}$$

$$\langle \sigma, n_1 * n_2 \rangle \rightarrow \langle \sigma, n_1 \times n_2 \rangle$$

$$\frac{\langle \sigma, e_1 \rangle \rightarrow \langle \sigma', e'_1 \rangle}{\langle \sigma, x := e_1; e_2 \rangle \rightarrow \langle \sigma', x := e'_1; e_2 \rangle}$$

$$\langle \sigma, x := n; e_2 \rangle \rightarrow \langle \sigma[x \mapsto n], e_2 \rangle$$

$$\frac{x \in \text{dom}(\sigma)}{\langle \sigma, x \rangle \rightarrow \langle \sigma, \sigma(x) \rangle}$$

**Evaluation contexts** separate the part of an expression that is currently being evaluated from the rest of the expression.

An evaluation context is defined as follows:

$$E ::= [\cdot] \mid E + e \mid n + E \mid E * e \mid n * E \mid x := E; e$$

where  $[\cdot]$  is a hole that can be filled with an expression.

$E[e]$  denotes the expression obtained by filling the hole in  $E$  with  $e$ .

All the congruence rules can be expressed as a single rule:

$$\frac{\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle}{\langle \sigma, E[e] \rangle \rightarrow \langle \sigma', E[e'] \rangle}$$

We can define the small-step operational semantics of VAE using the **congruence rule** with **evaluation context** and the **computation rules**:

$$E ::= [\cdot] \mid E + e \mid n + E \mid E * e \mid n * E \mid x := E; e$$

$$\frac{\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle}{\langle \sigma, E[e] \rangle \rightarrow \langle \sigma', E[e'] \rangle}$$

$$\frac{}{\langle \sigma, n_1 + n_2 \rangle \rightarrow \langle \sigma, n_1 + n_2 \rangle}$$

$$\frac{}{\langle \sigma, n_1 * n_2 \rangle \rightarrow \langle \sigma, n_1 \times n_2 \rangle}$$

$$\frac{}{\langle \sigma, x := n; e_2 \rangle \rightarrow \langle \sigma[x \mapsto n], e_2 \rangle}$$

$$\frac{x \in \text{dom}(\sigma)}{\langle \sigma, x \rangle \rightarrow \langle \sigma, \sigma(x) \rangle}$$

$$\begin{aligned}
 E &::= [\cdot] \mid E + e \mid n + E \mid E * e \mid n * E \mid E < e \mid n < E \\
 S &::= [\cdot] \mid x := E \mid S; s \mid \text{if } E \text{ then } s \text{ else } s
 \end{aligned}$$

$$\frac{\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle}{\langle \sigma, E[e] \rangle \rightarrow \langle \sigma', E[e'] \rangle} \quad \frac{\langle \sigma, e \rangle \rightarrow \langle \sigma', e' \rangle}{\langle \sigma, S[e] \rangle \rightarrow \langle \sigma', S[e'] \rangle} \quad \frac{\langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle}{\langle \sigma, S[s] \rangle \rightarrow \langle \sigma', S[s'] \rangle}$$

$$\frac{x \in \text{dom}(\sigma)}{\langle \sigma, x \rangle \rightarrow \langle \sigma, \sigma(x) \rangle} \quad \frac{}{\langle \sigma, n_1 + n_2 \rangle \rightarrow \langle \sigma, n_1 + n_2 \rangle}$$

$$\frac{}{\langle \sigma, n_1 * n_2 \rangle \rightarrow \langle \sigma, n_1 \times n_2 \rangle} \quad \frac{}{\langle \sigma, n_1 < n_2 \rangle \rightarrow \langle \sigma, n_1 < n_2 \rangle}$$

$$\frac{}{\langle \sigma, x := v \rangle \rightarrow \langle \sigma[x \mapsto v], \text{skip} \rangle} \quad \frac{}{\langle \sigma, \text{skip}; s_2 \rangle \rightarrow \langle \sigma, s_2 \rangle}$$

$$\frac{}{\langle \sigma, \text{if true then } s_1 \text{ else } s_2 \rangle \rightarrow \langle \sigma, s_1 \rangle}$$

$$\frac{}{\langle \sigma, \text{if false then } s_1 \text{ else } s_2 \rangle \rightarrow \langle \sigma, s_2 \rangle}$$

$$\frac{}{\langle \sigma, \text{while } e \text{ do } s \rangle \rightarrow \langle \sigma, \text{if } e \text{ then } (s; \text{while } e \text{ do } s) \text{ else skip} \rangle}$$

## 1. Evaluation Contexts

## 2. Lambda Calculus

$\alpha$ -Equivalence

$\beta$ -Reduction

Call by-Value (CBV) vs. Call by-Name (CBN)

Substitution

De Bruijn Indices

## 3. Definitional Translation

Church Encoding

Pairs and Let Bindings

Laziness

Adequacy

$$\begin{array}{l} e ::= x \quad (\text{variable}) \\ \quad | \lambda x.e \quad (\text{abstraction}) \\ \quad | e e \quad (\text{application}) \end{array}$$

The **lambda ( $\lambda$ ) calculus** is the foundation of functional programming.

Alonzo Church invented the lambda calculus in the 1930s as a formal system for studying computability and function definition.

The “ $\lambda$ ” term is used to denote anonymous functions.

The lambda calculus is **Turing complete**, meaning it can express any function computable by a Turing machine.

## Definition (Free and Bound Variables)

If a variable is bound by a lambda abstraction, it is called a **bound variable**; otherwise, it is called a **free variable**.

$$\text{fvs}(x) \triangleq \{x\}$$

$$\text{fvs}(\lambda x.e) \triangleq \text{fvs}(e) \setminus \{x\}$$

$$\text{fvs}(e_1 e_2) \triangleq \text{fvs}(e_1) \cup \text{fvs}(e_2)$$

For example,  $\text{fvs}(\lambda x.(x y)) = \{y\}$  and  $\text{fvs}((\lambda x.x) z) = \{z\}$ .

## Definition ( $\alpha$ -Equivalence)

Two lambda expressions  $e$  and  $e'$  are  $\alpha$ -**equivalent**, denoted  $e \equiv_\alpha e'$ , if they differ only in the names of their bound variables.

For example,  $\lambda x.\lambda y.(y x) \equiv_\alpha \lambda a.\lambda b.(b a)$ .

## Definition ( $\beta$ -Reduction)

The  $\beta$ -**reduction** is the process of applying a function to an argument by substituting the argument for the bound variable in the function body.

The following rule defines the **full  $\beta$ -reduction** evaluation strategy where we can reduce under lambda abstractions:

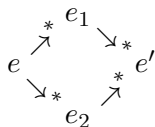
$$\overline{(\lambda x.e) e' \rightarrow e[x \mapsto e']}$$

with the following evaluation context:

$$E ::= [\cdot] \mid E e \mid e E \mid \lambda x.E$$

However, it is **non-deterministic** because we can choose to reduce any **redex** (i.e., reducible expression) in the expression in any order.

The full  $\beta$ -reduction strategy is **confluent**, meaning that if an expression can be reduced to two different expressions, there exists a common expression to which both can be further reduced.



## Theorem (Confluence)

$$\forall e, e_1, e_2. e \rightarrow^* e_1 \wedge e \rightarrow^* e_2 \implies \exists e'. e_1 \rightarrow^* e' \wedge e_2 \rightarrow^* e'$$

For deterministic evaluation, we need to specify the order of evaluation of redexes. There are two common evaluation strategies:

- **Call by-Value (CBV)**: The argument is evaluated before being passed to the function. It is called the **eager evaluation** strategy.

$$E ::= [\cdot] \mid E e \mid v E$$

$$\overline{(\lambda x.e) v \rightarrow e[x \mapsto v]}$$

- **Call by-Name (CBN)**: The argument is not evaluated until it is used. It is called the **lazy evaluation** strategy.

$$E ::= [\cdot] \mid E e$$

$$\overline{(\lambda x.e) e' \rightarrow e[x \mapsto e']}$$

The following expression  $\Omega$  has an infinite reduction sequence under both CBV and CBN because it is a self-application that keeps reducing to itself:

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

However, the following expression has an infinite reduction sequence under CBV but not under CBN:

- **Call by-Value (CBV):**

$$\begin{aligned} (\lambda y.(\lambda z.z))((\lambda x.xx)(\lambda x.xx)) &\rightarrow (\lambda y.(\lambda z.z))((\lambda x.xx)(\lambda x.xx)) \\ &\rightarrow \dots \end{aligned}$$

- **Call by-Name (CBN):**

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

A **substitution** is the process of replacing all free occurrences of a variable with an expression.  $e[x \mapsto e']$  denotes the substitution of  $x$  with  $e'$  in  $e$ .

$$\begin{aligned}
 y[x \mapsto e'] &\triangleq \begin{cases} e' & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
 (\lambda y. e)[x \mapsto e'] &\triangleq \begin{cases} \lambda y. e & \text{if } y = x \\ \lambda y. (e[x \mapsto e']) & \text{if } y \neq x \end{cases} \\
 (e_1 e_2)[x \mapsto e'] &\triangleq e_1[x \mapsto e'] e_2[x \mapsto e']
 \end{aligned}$$

For example,

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

Is it correct? No, it leads to **variable capture**:

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

To avoid variable capture, we need to apply  $\alpha$ -conversion by renaming the bound variable before substitution.

$$\begin{aligned}
 y[x \mapsto e'] &\triangleq \begin{cases} e' & \text{if } y = x \\ y & \text{if } y \neq x \end{cases} \\
 (\lambda y.e)[x \mapsto e'] &\triangleq \begin{cases} \lambda y.e & \text{if } y = x \\ \lambda y.(e[x \mapsto e']) & \text{if } y \neq x \wedge y \notin \text{fvs}(e') \\ (\lambda z.e[y \mapsto z])[x \mapsto e'] & \text{if } y \neq x \wedge y \in \text{fvs}(e') \end{cases} \\
 (e_1 e_2)[x \mapsto e'] &\triangleq e_1[x \mapsto e'] e_2[x \mapsto e']
 \end{aligned}$$

where  $z$  is a fresh variable that does not appear in  $e$  and  $e'$ .

Now, the substitution is correct:

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

Another way to avoid variable capture is to use **De Bruijn indices**, which represent variables by their distance from their binding lambda abstraction.

$$e ::= n \mid \lambda.e \mid e e$$

For example,

$$\lambda x.\lambda y.(x y) \quad \text{is represented as} \quad \lambda.\lambda.(1 0)$$

It is easy to check  $\alpha$ -equivalence using De Bruijn indices by comparing the indices of bound variables.

However, substitution becomes more complex with De Bruijn indices, as we need to adjust the indices of variables when substituting under lambda abstractions.

We can convert a lambda expression with named variables to one with De Bruijn indices as follows:

$$\begin{aligned}
 [x](\mathcal{X}) &\triangleq \mathcal{X}(x) \\
 [\lambda x.e](\mathcal{X}) &\triangleq \lambda.[e](\uparrow \mathcal{X}[x \mapsto 0]) \\
 [e_1 e_2](\mathcal{X}) &\triangleq [e_1](\mathcal{X}) [e_2](\mathcal{X})
 \end{aligned}$$

where  $\mathcal{X} : \mathbb{X} \rightarrow \mathbb{N}$  is a mapping from variables to their De Bruijn indices, and  $\uparrow \mathcal{X}$  is the mapping obtained by incrementing all indices in  $\mathcal{X}$  by 1.

For example,

$$\begin{aligned}
 [\lambda x.\lambda y.(x y)](\emptyset) &= \lambda.[\lambda y.(x y)]([x \mapsto 0]) \\
 &= \lambda.\lambda.[(x y)]([x \mapsto 1, y \mapsto 0]) \\
 &= \lambda.\lambda.(1 0)
 \end{aligned}$$

The substitution of lambda calculus with De Bruijn indices is defined as:

$$\begin{aligned}
 n[k \mapsto e'] &\triangleq \begin{cases} e' & \text{if } n = k \\ n & \text{otherwise} \end{cases} \\
 (\lambda.e)[k \mapsto e'] &\triangleq \lambda.e[k + 1 \mapsto \uparrow^1(e')] \\
 (e_1 e_2)[k \mapsto e'] &\triangleq e_1[k \mapsto e'] e_2[k \mapsto e']
 \end{aligned}$$

where  $\uparrow_c^d(e)$  is the operation that increments all free variables in  $e$  that are greater than or equal to  $c$  by  $d$ , and  $\uparrow^d(e) = \uparrow_0^d(e)$ :

$$\begin{aligned}
 \uparrow_c^d(n) &\triangleq \begin{cases} n & \text{if } n < c \\ n + d & \text{if } n \geq c \end{cases} \\
 \uparrow_c^d(\lambda.e) &\triangleq \lambda.\uparrow_{c+1}^d(e) \\
 \uparrow_c^d(e_1 e_2) &\triangleq \uparrow_c^d(e_1) \uparrow_c^d(e_2)
 \end{aligned}$$

## 1. Evaluation Contexts

## 2. Lambda Calculus

$\alpha$ -Equivalence

$\beta$ -Reduction

Call by-Value (CBV) vs. Call by-Name (CBN)

Substitution

De Bruijn Indices

## 3. Definitional Translation

Church Encoding

Pairs and Let Bindings

Laziness

Adequacy

A **definitional translation** is a mapping from the source language to the target language to define the semantics of a language by translating it to a well-understood language.

$$\mathcal{T}[\cdot] : (\text{Source Language}) \rightarrow (\text{Target Language})$$

For example, we can represent an extended lambda calculus with multi-argument functions as follows:

$$e ::= x \mid \lambda(x_1, \dots, x_n).e \mid e(e_1, \dots, e_n)$$

$$\begin{aligned}\mathcal{T}[x] &= x \\ \mathcal{T}[\lambda(x_1, \dots, x_n).e] &= \lambda x_1. \dots \lambda x_n. \mathcal{T}[e] \\ \mathcal{T}[e(e_1, \dots, e_n)] &= \mathcal{T}[e] \mathcal{T}[e_1] \dots \mathcal{T}[e_n]\end{aligned}$$

If we define definitional translation in the same language, the source language elements are called **syntactic sugar**, and this process is called **desugaring**.

The **Church encoding** is a way to represent data structures and operations in the lambda calculus.

$$e ::= n \mid e + e \mid e * e \mid e \text{ pow } e$$

For example, Church numerals represent natural numbers as functions:

$$\begin{aligned} \mathcal{T}[0] &\triangleq \lambda f. \lambda x. x \\ \mathcal{T}[1] &\triangleq \lambda f. \lambda x. f \ x \\ \mathcal{T}[2] &\triangleq \lambda f. \lambda x. f \ (f \ x) \\ \mathcal{T}[n] &\triangleq \lambda f. \lambda x. f \ (f \ (\dots (f \ x) \dots)) \end{aligned}$$

and their operations as follows:

$$\begin{aligned} \mathcal{T}[e_1 + e_2] &\triangleq \lambda f. \lambda x. \mathcal{T}[e_1] \ f \ (\mathcal{T}[e_2] \ f \ x) \\ \mathcal{T}[e_1 * e_2] &\triangleq \lambda f. \lambda x. \mathcal{T}[e_1] \ (\lambda g. \mathcal{T}[e_2] \ f \ g) \ x \\ \mathcal{T}[e_1 \text{ pow } e_2] &\triangleq \lambda f. \lambda x. \mathcal{T}[e_2] \ \mathcal{T}[e_1] \ f \ x \end{aligned}$$

We can show that  $1 + 1 = 2$  in the Church encoding as follows:

$$\begin{aligned}\mathcal{T}[\![1 + 1]\!] &= \lambda f.\lambda x.\mathcal{T}[\![1]\!] f (\mathcal{T}[\![1]\!] f x) \\ &= \lambda f.\lambda x.(\lambda f.\lambda x.f x) f ((\lambda f.\lambda x.f x) f x) \\ &\rightarrow \lambda f.\lambda x.(\lambda x.f x) ((\lambda f.\lambda x.f x) f x) \\ &\rightarrow \lambda f.\lambda x.f ((\lambda f.\lambda x.f x) f x) \\ &\rightarrow \lambda f.\lambda x.f (\lambda x.f x) x \\ &\rightarrow \lambda f.\lambda x.f (f x) \\ &= \mathcal{T}[\![2]\!]\end{aligned}$$

There are many other data structures and operations that can be represented in the lambda calculus using Church encoding, such as pairs, lists, booleans, etc.

$$e ::= \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \neg e \mid e \wedge e \mid e \vee e$$

Church encoding of booleans and their operations can be defined as:

$$\mathcal{T}[\text{true}] \triangleq \lambda t. \lambda f. t$$

$$\mathcal{T}[\text{false}] \triangleq \lambda t. \lambda f. f$$

$$\mathcal{T}[\text{if } e \text{ then } e_1 \text{ else } e_2] \triangleq \mathcal{T}[e] \mathcal{T}[e_1] \mathcal{T}[e_2]$$

$$\mathcal{T}[\neg e] \triangleq \mathcal{T}[e] \mathcal{T}[\text{false}] \mathcal{T}[\text{true}]$$

$$\mathcal{T}[e_1 \wedge e_2] \triangleq \mathcal{T}[e_1] \mathcal{T}[e_2] \mathcal{T}[\text{false}]$$

$$\mathcal{T}[e_1 \vee e_2] \triangleq \mathcal{T}[e_1] \mathcal{T}[\text{true}] \mathcal{T}[e_2]$$

$e ::= x \mid \lambda x.e \mid e e \mid (e, e) \mid e.\text{fst} \mid e.\text{snd} \mid \text{let } x = e \text{ in } e$

We can represent pairs and let bindings in the lambda calculus as follows:

$$\mathcal{T}[[x]] \triangleq x$$

$$\mathcal{T}[[\lambda x.e]] \triangleq \lambda x.\mathcal{T}[[e]]$$

$$\mathcal{T}[[e_1 e_2]] \triangleq \mathcal{T}[[e_1]] \mathcal{T}[[e_2]]$$

$$\mathcal{T}[[ (e_1, e_2) ]] \triangleq \lambda p.p \mathcal{T}[[e_1]] \mathcal{T}[[e_2]]$$

$$\mathcal{T}[[e.\text{fst}]] \triangleq \mathcal{T}[[e]] (\lambda x.\lambda y.x)$$

$$\mathcal{T}[[e.\text{snd}]] \triangleq \mathcal{T}[[e]] (\lambda x.\lambda y.y)$$

$$\mathcal{T}[[\text{let } x = e_1 \text{ in } e_2]] \triangleq (\lambda x.\mathcal{T}[[e_2]]) \mathcal{T}[[e_1]]$$

$$e ::= x \mid \lambda x.e \mid e e$$

$$E ::= [\cdot] \mid E e$$

$$\frac{e \rightarrow e'}{E[e] \rightarrow E[e']} \quad \frac{}{(\lambda x.e) e' \rightarrow e[x \mapsto e']}$$

We can represent laziness (i.e., call by name, CBN) in the lambda calculus with the eager evaluation strategy (i.e., call by value, CBV) as follows:

$$\mathcal{T}[[x]] \triangleq x(\lambda y.y)$$

$$\mathcal{T}[[\lambda x.e]] \triangleq \lambda x.\mathcal{T}[[e]]$$

$$\mathcal{T}[[e_1 e_2]] \triangleq \mathcal{T}[[e_1]] (\lambda z.\mathcal{T}[[e_2]])$$

where  $z$  is a not a free variable in  $\mathcal{T}[[e_2]]$  to avoid variable capture.

To show that a definitional translation is correct, we need to prove its **soundness** and **completeness** with the equivalence relation between source and target values:  $\equiv \subseteq \mathbb{V}_{\text{src}} \times \mathbb{V}_{\text{tgt}}$ .

### Definition (Soundness)

Every target evaluation should represent a source evaluation:

$$\forall e \in \mathbb{E}_{\text{src}}. \forall v' \in \mathbb{V}_{\text{tgt}}. \\ (\mathcal{T}[e] \rightarrow_{\text{tgt}}^* v') \implies (\exists v \in \mathbb{V}_{\text{src}}. e \rightarrow_{\text{src}}^* v) \wedge (v \equiv v')$$

### Definition (Completeness)

Every source evaluation should be represented by a target evaluation:

$$\forall e \in \mathbb{E}_{\text{src}}. \forall v \in \mathbb{V}_{\text{src}}. \\ (e \rightarrow_{\text{src}}^* v) \implies (\exists v' \in \mathbb{V}_{\text{tgt}}. \mathcal{T}[e] \rightarrow_{\text{tgt}}^* v') \wedge (v \equiv v')$$

- Denotational Semantics

Jihyeok Park

[jihyeok\\_park@korea.ac.kr](mailto:jihyeok_park@korea.ac.kr)

<https://plrg.korea.ac.kr>