Lecture 4 – Evaluation Contexts and Lambda Calculus

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

- 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

Contents

1. Evaluation Contexts

2. Lambda Calculus
   - α-Equivalence
   - β-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

Contents

1. Evaluation Contexts

2. Lambda Calculus
   - α-Equivalence
   - β-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

Small-Step Operational Semantics – VAE

$$e ::= n \mid e \texttt{ + } e \mid e \texttt{ * } e \mid x \texttt{ := } e; \ e \mid x$$

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

Small-Step Operational Semantics – VAE

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

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 \texttt{ + } e_2 \rangle \to \langle \sigma', e_1' \texttt{ + } e_2 \rangle}$$

$$\frac{\langle \sigma, e_2 \rangle \to \langle \sigma', e_2' \rangle}{\langle \sigma, n_1 \texttt{ + } e_2 \rangle \to \langle \sigma', n_1 \texttt{ + } e_2' \rangle}$$

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

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, e_1 \texttt{ * } e_2 \rangle \to \langle \sigma', e_1' \texttt{ * } e_2 \rangle}$$

$$\frac{\langle \sigma, e_2 \rangle \to \langle \sigma', e_2' \rangle}{\langle \sigma, n_1 \texttt{ * } e_2 \rangle \to \langle \sigma', n_1 \texttt{ * } e_2' \rangle}$$

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

$$\frac{\langle \sigma, e_1 \rangle \to \langle \sigma', e_1' \rangle}{\langle \sigma, x \texttt{ := } e_1\texttt{; } e_2 \rangle \to \langle \sigma', x \texttt{ := } e_1'\texttt{; } e_2 \rangle}$$

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

$$\frac{x \in \mathsf{dom}(\sigma)}{\langle \sigma, x \rangle \to \langle \sigma, \sigma(x) \rangle}$$

Evaluation Contexts

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.

Evaluation Contexts

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:

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

Evaluation Contexts - VAE

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

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

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

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

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

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

$$\frac{x \in \mathsf{dom}(\sigma)}{\langle \sigma, x \rangle \to \langle \sigma, \sigma(x) \rangle}$$

Evaluation Contexts – IMP

PLRG

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

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

$$
\frac{x \in \mathsf{dom}(\sigma)}{\langle \sigma, x\rangle \to \langle \sigma, \sigma(x)\rangle} \qquad \frac{}{\langle \sigma, n_1 \texttt{ + } n_2\rangle \to \langle \sigma, n_1 + n_2\rangle}
$$

$$
\frac{}{\langle \sigma, n_1 \texttt{ * } n_2\rangle \to \langle \sigma, n_1 \times n_2\rangle} \qquad \frac{}{\langle \sigma, n_1 \texttt{ < } n_2\rangle \to \langle \sigma, n_1 < n_2\rangle}
$$

$$
\frac{}{\langle \sigma, x \texttt{ := } v\rangle \to \langle \sigma[x \mapsto v], \texttt{skip}\rangle} \qquad \frac{}{\langle \sigma, \texttt{skip;}s_2\rangle \to \langle \sigma, s_2\rangle}
$$

$$
\frac{}{\langle \sigma, \texttt{if true then } s_1 \texttt{ else } s_2\rangle \to \langle \sigma, s_1\rangle}
$$

$$
\frac{}{\langle \sigma, \texttt{if false then } s_1 \texttt{ else } s_2\rangle \to \langle \sigma, s_2\rangle}
$$

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

Contents

1. Evaluation Contexts

2. Lambda Calculus
   - α-Equivalence
   - β-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

Lambda Calculus

$$
\begin{array}{lll}
e & ::= x & \text{(variable)} \\
& \mid \lambda x.e & \text{(abstraction)} \\
& \mid e\ e & \text{(application)}
\end{array}
$$

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

Lambda Calculus

$$
\begin{array}{lll}
e & ::= x & \text{(variable)} \\
& \mid \lambda x.e & \text{(abstraction)} \\
& \mid e\ e & \text{(application)}
\end{array}
$$

The **lambda (λ) 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.

Lambda Calculus

$$
\begin{array}{lll}
e & ::= x & \text{(variable)} \\
& \mid \lambda x.e & \text{(abstraction)} \\
& \mid e\ e & \text{(application)}
\end{array}
$$

The **lambda (λ) 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 "λ" term is used to denote anonymous functions.

Lambda Calculus

$$
\begin{array}{lll}
e & ::= x & \text{(variable)} \\
& \mid \lambda x.e & \text{(abstraction)} \\
& \mid e\ e & \text{(application)}
\end{array}
$$

The **lambda (λ) 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 "λ" term is used to denote anonymous functions.

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

α-Equivalence

PLRG

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**.

$$
\begin{array}{lcl}
\mathsf{fvs}(x) & \triangleq & \{x\} \\
\mathsf{fvs}(\lambda x.e) & \triangleq & \mathsf{fvs}(e) \setminus \{x\} \\
\mathsf{fvs}(e_1\ e_2) & \triangleq & \mathsf{fvs}(e_1) \cup \mathsf{fvs}(e_2)
\end{array}
$$

α-Equivalence

PLRG

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**.

$$\begin{array}{lcl} \mathsf{fvs}(x) & \triangleq & \{x\} \\ \mathsf{fvs}(\lambda x.e) & \triangleq & \mathsf{fvs}(e) \setminus \{x\} \\ \mathsf{fvs}(e_1\ e_2) & \triangleq & \mathsf{fvs}(e_1) \cup \mathsf{fvs}(e_2) \end{array}$$

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

α-Equivalence

PLRG

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**.

$$\begin{array}{lcl} \mathsf{fvs}(x) & \triangleq & \{x\} \\ \mathsf{fvs}(\lambda x.e) & \triangleq & \mathsf{fvs}(e) \setminus \{x\} \\ \mathsf{fvs}(e_1\ e_2) & \triangleq & \mathsf{fvs}(e_1) \cup \mathsf{fvs}(e_2) \end{array}$$

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

Definition (α-Equivalence)

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

α-Equivalence

PLRG

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**.

$$\begin{array}{lcl} \mathsf{fvs}(x) & \triangleq & \{x\} \\ \mathsf{fvs}(\lambda x.e) & \triangleq & \mathsf{fvs}(e) \setminus \{x\} \\ \mathsf{fvs}(e_1\ e_2) & \triangleq & \mathsf{fvs}(e_1) \cup \mathsf{fvs}(e_2) \end{array}$$

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

Definition (α-Equivalence)

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

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

β-Reduction

PLRG

Definition (β-Reduction)

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

β-Reduction

PLRG

Definition (β-Reduction)

The β-**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** β-**reduction** evaluation strategy where we can reduce under lambda abstractions:

$$\frac{}{(\lambda x.e)\ e' \to e[x \mapsto e']}$$

with the following evaluation context:

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

β-Reduction

PLRG

Definition (β-Reduction)

The β-**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** β-**reduction** evaluation strategy where we can reduce under lambda abstractions:

$$\frac{}{(\lambda x.e)\ e' \to e[x \mapsto e']}$$

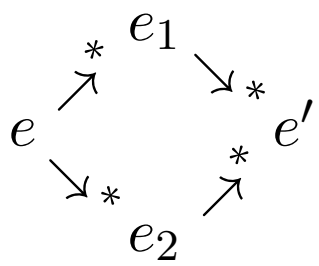
with the following evaluation context:

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

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

Confluence

The full β-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)

$$\forall e, e_1, e_2.\ e \to^* e_1 \land e \to^* e_2 \implies \exists e'.e_1 \to^* e' \land e_2 \to^* e'$$

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

PLRG

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.

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

$$\frac{}{(\lambda x.e)\ v \to e[x \mapsto v]}$$

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

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

$$\frac{}{(\lambda x.e)\ e' \to e[x \mapsto e']}$$

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

PLRG

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

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

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

The following expression Ω 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)**:

$$\begin{array}{rcl} (\lambda y.(\lambda z.z))((\lambda x.xx)(\lambda x.xx)) & \rightarrow & (\lambda y.(\lambda z.z))((\lambda x.xx)(\lambda x.xx)) \\ & \rightarrow & \ldots \end{array}$$

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

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

Substitution

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.

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

For example,

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

Substitution

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.

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

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?

Substitution

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**:

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

Substitution

To avoid variable capture, we need to apply **α-conversion** by renaming the bound variable before substitution.

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

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

Substitution

To avoid variable capture, we need to apply **α-conversion** by renaming the bound variable before substitution.

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

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

Now, the substitution is correct:

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

De Bruijn Indices

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

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

De Bruijn Indices

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,

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

De Bruijn Indices

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 α-equivalence using De Bruijn indices by comparing the indices of bound variables.

De Bruijn Indices

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 α-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.

De Bruijn Indices

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

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

where $\mathcal{X} : \mathbb{X} \rightharpoonup \mathbb{N}$ 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.

De Bruijn Indices

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

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

where $\mathcal{X} : \mathbb{X} \rightharpoonup \mathbb{N}$ 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,

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

De Bruijn Indices

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

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

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 $\uparrow^d (e) = \uparrow_0^d (e)$:

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

Contents

1. Evaluation Contexts

2. Lambda Calculus
   - α-Equivalence
   - β-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

Definitional Translation

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.

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

Definitional Translation

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.

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

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

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

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

Definitional Translation

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.

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

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

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

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

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

Church Encoding

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

Church Encoding

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

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

For example, Church numerals represent natural numbers as functions:

$$\begin{array}{lcl} \mathcal{T}[\![0]\!] & \triangleq & \lambda f.\lambda x.x \\ \mathcal{T}[\![1]\!] & \triangleq & \lambda f.\lambda x.f\ x \\ \mathcal{T}[\![2]\!] & \triangleq & \lambda f.\lambda x.f\ (f\ x) \\ \mathcal{T}[\![n]\!] & \triangleq & \lambda f.\lambda x.f\ (f\ (\ldots(f\ x)\ldots)) \end{array}$$

Church Encoding

PLRG

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

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

For example, Church numerals represent natural numbers as functions:

$$\begin{array}{lcl} \mathcal{T}[\![0]\!] & \triangleq & \lambda f.\lambda x.x \\ \mathcal{T}[\![1]\!] & \triangleq & \lambda f.\lambda x.f\ x \\ \mathcal{T}[\![2]\!] & \triangleq & \lambda f.\lambda x.f\ (f\ x) \\ \mathcal{T}[\![n]\!] & \triangleq & \lambda f.\lambda x.f\ (f\ (\ldots(f\ x)\ldots)) \end{array}$$

and their operations as follows:

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

Church Encoding – Numerals

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

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

Church Encoding – Numerals

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

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

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.

Church Encoding – Booleans

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

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

$$\begin{aligned}
\mathcal{T}[\![\texttt{true}]\!] &\triangleq \lambda t.\lambda f.t \\
\mathcal{T}[\![\texttt{false}]\!] &\triangleq \lambda t.\lambda f.f \\
\mathcal{T}[\![\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2]\!] &\triangleq \mathcal{T}[\![e]\!]\ \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\neg e]\!] &\triangleq \mathcal{T}[\![e]\!]\ \mathcal{T}[\![\texttt{false}]\!]\ \mathcal{T}[\![\texttt{true}]\!] \\
\mathcal{T}[\![e_1 \wedge e_2]\!] &\triangleq \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!]\ \mathcal{T}[\![\texttt{false}]\!] \\
\mathcal{T}[\![e_1 \vee e_2]\!] &\triangleq \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![\texttt{true}]\!]\ \mathcal{T}[\![e_2]\!]
\end{aligned}$$

Pairs and Let Bindings

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

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

$$\begin{aligned}
\mathcal{T}[\![x]\!] &\triangleq x \\
\mathcal{T}[\![\lambda x.e]\!] &\triangleq \lambda x.\mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &\triangleq \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![(e_1, e_2)]\!] &\triangleq \lambda p.p\ \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![e.\texttt{fst}]\!] &\triangleq \mathcal{T}[\![e]\!]\ (\lambda x.\lambda y.x) \\
\mathcal{T}[\![e.\texttt{snd}]\!] &\triangleq \mathcal{T}[\![e]\!]\ (\lambda x.\lambda y.y) \\
\mathcal{T}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] &\triangleq (\lambda x.\mathcal{T}[\![e_2]\!])\ \mathcal{T}[\![e_1]\!]
\end{aligned}$$

Laziness

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

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

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

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:

$$\begin{aligned}
\mathcal{T}[\![x]\!] &\triangleq x(\lambda y.y) \\
\mathcal{T}[\![\lambda x.e]\!] &\triangleq \lambda x.\mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &\triangleq \mathcal{T}[\![e_1]\!]\ (\lambda z.\mathcal{T}[\![e_2]\!])
\end{aligned}$$

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

Adequacy

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: $\equiv \subseteq \mathbb{V}_{\mathsf{src}} \times \mathbb{V}_{\mathsf{tgt}}$.

Definition (Soundness)

Every target evaluation should represent a source evaluation:

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

Definition (Completeness)

Every source evaluation should be represented by a target evaluation:

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

Next Lecture

- Denotational Semantics

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr