Lecture 9 – Abstract Machines

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

- State and Trace Properties
- Safety Properties
  - Irreducibility and Finite Bad Prefix
  - Prefix Closure and Limit
  - Formal Definition of Safety Properties
  - Verification by Invariance
- Liveness Properties
  - Eventuality
  - Formal Definition of Liveness Properties
  - Verification by Variance
- Decomposition of Trace Properties

Contents

Continuations

PLRG

A **continuation** is a representation of the "rest" of the computation.

For example, consider the following expression:

```
(1 + 3) * 5
```

It **implicitly** represents the following computation:

1. Evaluate 1. (Result: `1`)
2. Evaluate 3. (Result: 3)
3. Add the results of step 1 and 2. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step 3 and 4. (Result: `4 * 5 = 20`)

The **continuation** of k-th step is the steps from $(k+1)$-th to the last one.

For instance, the **continuation** of the 3rd step is the 4th and 5th steps.

Continuations

Let's split the expression into: 1) current evaluation and 2) continuation.

```
(1 + 3) * 5
```

1. Evaluate 1. (Result: 1)
2. Evaluate 3. (Result: 3)
3. Add the results of step 1 and 2. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step 3 and 4. (Result: `4 * 5 = 20`)

Continuations

Let's split the expression into: 1) current evaluation and 2) continuation.

```
{
  x1 => (x1 + 3) * 5        // step 2-5 (continuation of step 1)
}(1)                        // step 1
```

2. Evaluate 3. (Result: 3)
3. Add the results of step 1 and 2. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step 3 and 4. (Result: `4 * 5 = 20`)

Continuations

```
{
  x1 => {
    x2 => (x1 + x2) * 5    // step 3-5 (continuation of step 2)
  }(3)                     // step 2
}(1)                       // step 1
```

3. Add the results of step 1 and 2. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step 3 and 4. (Result: `4 * 5 = 20`)

Continuations

```
{
  x1 => {
    x2 => {
      x3 => x3 * 5          // step 4-5 (continuation of step 3)
    }(x1 + x2)              // step 3
  }(3)                      // step 2
}(1)                        // step 1
```

4. Evaluate 5. (Result: 5)
5. Multiply the results of step 3 and 4. (Result: `4 * 5 = 20`)

Continuations

```
{
  x1 => {
    x2 => {
      x3 => {
        x4 => x3 * x4      // step 5 (continuation of step 4)
      }(5)                 // step 4
    }(x1 + x2)             // step 3
  }(3)                     // step 2
}(1)                       // step 1
```

1. Multiply the results of step 3 and 4. (Result: `4 * 5 = 20`)

Continuations

```
{
  x1 => {
    x2 => {
      x3 => {
        x4 => {
          x5 => x5          // no more steps (continuation of step 5)
        }(x3 * x4)          // step 5
      }(5)                  // step 4
    }(x1 + x2)              // step 3
  }(3)                      // step 2
}(1)                        // step 1
```

We can explicitly give name to the continuation of each step with the following denotational translation for the `let`-binding:

$$\mathcal{D}[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!] \triangleq (\lambda x.\mathcal{D}[\![e_2]\!])\ \mathcal{D}[\![e_1]\!]$$

A-Normal Form

$$\mathcal{D}[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!] \triangleq (\lambda x.\mathcal{D}[\![e_2]\!])\ \mathcal{D}[\![e_1]\!]$$

```
let x1 = 1;                 // step 1
let x2 = 3;                 // step 2
let x3 = x1 + x2;           // step 3
let x4 = 5;                 // step 4
let x5 = x3 * x4;           // step 5
x5                          // no more steps (continuation of step 5)
```

We call this form an **A-normal form** (administrative normal form) or shortly **ANF**.

Can we represent continuations in a formal semantics?

Contents

Recall: Evaluation Contexts and λ-Calculus

Lambda calculus is the simplest functional language:

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

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.

Recall: Evaluation Contexts and λ-Calculus

Let's use the **call-by-value (CBV) evaluation strategy** where we only reduce the leftmost-outermost redex:

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

with the following evaluation context:

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

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

For example, $\overbrace{(([\cdot](\lambda x.x))(\lambda y.y))}^{E}[\overbrace{\lambda z.z}^{e}] = ((\lambda z.z)(\lambda x.x))(\lambda y.y)$

Recall: Evaluation Contexts and λ-Calculus

Let's extends λ-calculus with numbers and addition:

$$e ::= n \mid e + e \mid x \mid \lambda x.e \mid e\ e \qquad v ::= n \mid \lambda x.e$$

with the following inference rules:

$$\frac{}{(\lambda x.e)\ v \to e[x \mapsto v]} \qquad \frac{}{n_1 + n_2 \to n_1 + n_2}$$

and the evaluation context:

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

CC Machine

In fact, the **evaluation context** represents the **continuation** of the current evaluation.

A **CC machine** is an abstract machine that represents a state transition system for evaluating λ-calculus with the following form of states:

$$\sigma = \langle e, E \rangle$$

- An expression e represents the current **control** (C)

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

- An evaluation context E represents the current **continuation** (C)

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

CC Machine

Let's define the transition rules for the CC machine.

$$
\begin{array}{lcl}
\langle e_1 \texttt{ + } e_2, E\rangle & \rightarrow & \langle e_1, E[[\cdot] \texttt{ + } e_2]\rangle \\
\langle n_1, E[[\cdot] \texttt{ + } e_2]\rangle & \rightarrow & \langle n_1 \texttt{ + } e_2, E\rangle \\
\langle n_1 \texttt{ + } e_2, E\rangle & \rightarrow & \langle e_2, E[n_1 \texttt{ + } [\cdot]]\rangle \\
\langle n_2, E[n_1 \texttt{ + } [\cdot]]\rangle & \rightarrow & \langle n_1 \texttt{ + } n_2, E\rangle \\
\langle n_1 \texttt{ + } n_2, E\rangle & \rightarrow & \langle n_1 + n_2, E\rangle \\
\langle e_1 \ e_2, E\rangle & \rightarrow & \langle e_1, E[[\cdot] \ e_2]\rangle \\
\langle v, E[[\cdot] \ e]\rangle & \rightarrow & \langle v \ e, E\rangle \\
\langle v \ e, E\rangle & \rightarrow & \langle e, E[v \ [\cdot]]\rangle \\
\langle v_2, E[v_1 \ [\cdot]]\rangle & \rightarrow & \langle v_1 \ v_2, E\rangle \\
\langle (\lambda x.e) \ v, E\rangle & \rightarrow & \langle e[x \mapsto v], E\rangle
\end{array}
$$

CC Machine: Example

PLRG

For example, let's evaluate the following expression with the CC machine:

$$((\lambda x.\lambda y.x + y)\ 1)\ (2 + 3)$$

$$\begin{array}{lll}
 & \langle((\lambda x.\lambda y.x + y)\ 1)\ (2 + 3) & , \quad [\cdot] \rangle \\
\rightarrow & \langle(\lambda x.\lambda y.x + y)\ 1 & , \quad [\cdot]\ (2 + 3) \rangle \\
\rightarrow & \langle\lambda y.1 + y & , \quad [\cdot]\ (2 + 3) \rangle \\
\rightarrow & \langle(\lambda y.1 + y)\ (2 + 3) & , \quad [\cdot] \rangle \\
\rightarrow & \langle 2 + 3 & , \quad (\lambda y.1 + y)\ [\cdot] \rangle \\
\rightarrow & \langle 5 & , \quad (\lambda y.1 + y)\ [\cdot] \rangle \\
\rightarrow & \langle(\lambda y.1 + y)\ 5 & , \quad [\cdot] \rangle \\
\rightarrow & \langle 1 + 5 & , \quad [\cdot] \rangle \\
\rightarrow & \langle 6 & , \quad [\cdot] \rangle
\end{array}$$

SCC Machine

The following two rules for the CC machine can be merged:

$$\begin{array}{lcl} \langle n_1, E[[\cdot] \texttt{ + } e_2] \rangle & \rightarrow & \langle n_1 \texttt{ + } e_2, E \rangle \\ \langle n_1 \texttt{ + } e_2, E \rangle & \rightarrow & \langle e_2, E[n_1 \texttt{ + } [\cdot]] \rangle \end{array}$$

into the following single rule:

$$\langle n_1, E[[\cdot] \texttt{ + } e_2] \rangle \rightarrow \langle e_2, E[n_1 \texttt{ + } [\cdot]] \rangle$$

Let's define a **simplified CC (SCC)** machine as follows:

$$\begin{array}{lcl} \langle e_1 \texttt{ + } e_2, E \rangle & \rightarrow & \langle e_1, E[[\cdot] \texttt{ + } e_2] \rangle \\ \langle n_1, E[[\cdot] \texttt{ + } e_2] \rangle & \rightarrow & \langle e_2, E[n_1 \texttt{ + } [\cdot]] \rangle \\ \langle n_2, E[n_1 \texttt{ + } [\cdot]] \rangle & \rightarrow & \langle n_1 + n_2, E \rangle \\ \langle e_1 \ e_2, E \rangle & \rightarrow & \langle e_1, E[[\cdot] \ e_2] \rangle \\ \langle v, E[[\cdot] \ e] \rangle & \rightarrow & \langle e, E[v \ [\cdot]] \rangle \\ \langle v, E[(\lambda x.e) \ [\cdot]] \rangle & \rightarrow & \langle e[x \mapsto v], E \rangle \end{array}$$

SCC Machine: Example

Let's evaluate the same expression with the SCC machine:

$$
\begin{array}{lll}
 & \langle((\lambda x.\lambda y.x + y)\ 1)\ (2 + 3) & , \quad [\cdot]\ \rangle \\
\rightarrow & \langle(\lambda x.\lambda y.x + y)\ 1 & , \quad [\cdot]\ (2 + 3)\ \rangle \\
\rightarrow & \langle\lambda x.\lambda y.x + y & , \quad ([\cdot]\ 1)\ (2 + 3)\ \rangle \\
\rightarrow & \langle 1 & , \quad ((\lambda x.\lambda y.x + y)\ [\cdot])\ (2 + 3)\ \rangle \\
\rightarrow & \langle\lambda y.1 + y & , \quad [\cdot]\ (2 + 3)\ \rangle \\
\rightarrow & \langle 2 + 3 & , \quad (\lambda y.1 + y)\ [\cdot]\ \rangle \\
\rightarrow & \langle 2 & , \quad (\lambda y.1 + y)\ ([\cdot] + 3)\ \rangle \\
\rightarrow & \langle 3 & , \quad (\lambda y.1 + y)\ (2 + [\cdot])\ \rangle \\
\rightarrow & \langle 5 & , \quad (\lambda y.1 + y)\ [\cdot]\ \rangle \\
\rightarrow & \langle 1 + 5 & , \quad [\cdot]\ \rangle \\
\rightarrow & \langle 1 & , \quad [\cdot] + 5\ \rangle \\
\rightarrow & \langle 5 & , \quad 1 + [\cdot]\ \rangle \\
\rightarrow & \langle 6 & , \quad [\cdot]\ \rangle
\end{array}
$$

Contents

CK Machine

In CC and SCC machines, we need **search** to find the innermost context for the next reduction:

$$\langle 2, (\lambda y.1 + y)\ ([\cdot] + 3)\rangle$$

To remove inefficient searching, the **CK machine** uses a **stack** to represent the continuation with the following form of states:

$$\sigma = \langle e, \kappa \rangle$$

- An expression e represents the current **control** (C)

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

- A stack κ represents the current **continuation** (K)

$$\kappa ::= [\cdot] \mid ([\cdot] + e) :: \kappa \mid (n + [\cdot]) :: \kappa \mid ([\cdot]\ e) :: \kappa \mid ((\lambda x.e)\ [\cdot]) :: \kappa$$

CK Machine

The transition rules for the CK machine are as follows:

$$
\begin{array}{lcl}
\langle e_1 \texttt{ + } e_2, \kappa \rangle & \rightarrow & \langle e_1, ([\cdot] \texttt{ + } e_2) \ :: \ \kappa \rangle \\
\langle n_1, ([\cdot] \texttt{ + } e_2) \ :: \ \kappa \rangle & \rightarrow & \langle e_2, (n_1 \texttt{ + } [\cdot]) \ :: \ \kappa \rangle \\
\langle n_2, (n_1 \texttt{ + } [\cdot]) \ :: \ \kappa \rangle & \rightarrow & \langle n_1 + n_2, \kappa \rangle \\
\langle e_1 \ e_2, \kappa \rangle & \rightarrow & \langle e_1, ([\cdot] \ e_2) \ :: \ \kappa \rangle \\
\langle \lambda x.e, ([\cdot] \ e) \ :: \ \kappa \rangle & \rightarrow & \langle e, ((\lambda x.e) \ [\cdot]) \ :: \ \kappa \rangle \\
\langle v, ((\lambda x.e) \ [\cdot]) \ :: \ \kappa \rangle & \rightarrow & \langle e[x \mapsto v], \kappa \rangle
\end{array}
$$

CK Machine: Example

Let's evaluate the same expression with the CK machine:

$$
\begin{array}{lll}
 & \langle((\lambda x.\lambda y.x + y)\ 1)\ (2 + 3) & , \quad [\cdot]\ \rangle \\
\rightarrow & \langle(\lambda x.\lambda y.x + y)\ 1 & , \quad ([\cdot]\ (2 + 3))\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle\lambda x.\lambda y.x + y & , \quad ([\cdot]\ 1)\ ::\ ([\cdot]\ (2 + 3))\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 1 & , \quad ((\lambda x.\lambda y.x + y)\ [\cdot])\ ::\ ([\cdot]\ (2 + 3))\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle\lambda y.1 + y & , \quad ([\cdot]\ (2 + 3))\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 2 + 3 & , \quad ((\lambda y.1 + y)\ [\cdot])\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 2 & , \quad ([\cdot] + 3)\ ::\ ((\lambda y.1 + y)\ [\cdot])\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 3 & , \quad (2 + [\cdot])\ ::\ ((\lambda y.1 + y)\ [\cdot])\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 5 & , \quad ((\lambda y.1 + y)\ [\cdot])\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 1 + 5 & , \quad [\cdot]\ \rangle \\
\rightarrow & \langle 1 & , \quad ([\cdot] + 5)\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 5 & , \quad (1 + [\cdot])\ ::\ [\cdot]\ \rangle \\
\rightarrow & \langle 6 & , \quad [\cdot]\ \rangle
\end{array}
$$

CEK Machine

However, **substitution** is inefficient because it requires traversing the expression to replace all free occurrences of the variable with the value.

To remove inefficient substitution, the **CEK machine** uses an **environment** to represent the mapping from variables to values with the following form of states:

$$\sigma = \langle e, \rho, \kappa \rangle$$

- An expression e represents the current **control** (c).
- An environment (E) ρ for the mapping from variables to values:

$$\rho \in \mathbb{X} \rightharpoonup \mathbb{V}$$

- A stack κ represents the current **continuation** (K):

$$\begin{aligned} \kappa ::= {} & [\cdot] \mid ([\cdot] \texttt{ + } (e, \rho)) \ :: \ \kappa \mid (n \texttt{ + } [\cdot]) \ :: \ \kappa \\ & \mid ([\cdot] \ (e, \rho)) \ :: \ \kappa \mid ((\lambda x.e, \rho) \ [\cdot]) \ :: \ \kappa \end{aligned}$$

where $(\lambda x.e, \rho)$ is a **closure** that represents a function value with the **captured environment** ρ.

CEK Machine

PLRG

The transition rules for the CEK machine are as follows:

$$
\begin{array}{lcl}
\langle x, \rho, \kappa \rangle & \to & \langle \rho(x), \varnothing, \kappa \rangle \\
\langle e_1 \texttt{ + } e_2, \rho, \kappa \rangle & \to & \langle e_1, \rho, ([\cdot] \texttt{ + } (e_2, \rho)) \ :: \ \kappa \rangle \\
\langle n_1, \rho, ([\cdot] \texttt{ + } (e_2, \rho')) \ :: \ \kappa \rangle & \to & \langle e_2, \rho', (n_1 \texttt{ + } [\cdot]) \ :: \ \kappa \rangle \\
\langle n_2, \rho', (n_1 \texttt{ + } [\cdot]) \ :: \ \kappa \rangle & \to & \langle n_1 + n_2, \varnothing, \kappa \rangle \\
\langle e_1 \ e_2, \rho, \kappa \rangle & \to & \langle e_1, \rho, ([\cdot] \ (e_2, \rho)) \ :: \ \kappa \rangle \\
\langle \lambda x.e, \rho, ([\cdot] \ (e', \rho')) \ :: \ \kappa \rangle & \to & \langle e', \rho', ((\lambda x.e, \rho) \ [\cdot]) \ :: \ \kappa \rangle \\
\langle v, ((\lambda x.e, \rho) \ [\cdot]) \ :: \ \kappa \rangle & \to & \langle e, \rho[x \mapsto v], \kappa \rangle
\end{array}
$$

where $\rho[x \mapsto v]$ is the environment obtained by extending ρ with the mapping from x to v.

Since we capture the environment when creating a closure, we can support **static scoping** with the CEK machine.

CEK Machine: Example

PLRG

$$
\begin{array}{lllll}
 & \langle((\lambda x.\lambda y.x + y)\ 1)\ (2 + 3) & , \varnothing & & , [\cdot]\ \rangle \\
\rightarrow & \langle(\lambda x.\lambda y.x + y)\ 1 & , \varnothing & , ([\cdot]\ (2 + 3, \varnothing)) :: & [\cdot]\ \rangle \\
\rightarrow & \langle\lambda x.\lambda y.x + y & , \varnothing & , ([\cdot]\ (1, \varnothing)) :: ([\cdot]\ (2 + 3, \varnothing)) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 1 & , \varnothing & , ((\lambda x.\lambda y.x + y, \varnothing)\ [\cdot]) :: ([\cdot]\ (2 + 3, \varnothing)) :: & [\cdot]\ \rangle \\
\rightarrow & \langle\lambda y.x + y & , \rho_0 & , ([\cdot]\ (2 + 3, \varnothing)) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 2 + 3 & , \varnothing & , ((\lambda y.x + y, \rho_0)\ [\cdot]) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 2 & , \varnothing & , ([\cdot] + (3, \varnothing)) :: ((\lambda y.x + y, \rho_0)\ [\cdot]) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 3 & , \varnothing & , (2 + [\cdot]) :: ((\lambda y.x + y, \rho_0)\ [\cdot]) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 5 & , \varnothing & , ((\lambda y.x + y, \rho_0)\ [\cdot]) :: & [\cdot]\ \rangle \\
\rightarrow & \langle x + y & , \rho_1 & & , [\cdot]\ \rangle \\
\rightarrow & \langle x & , \rho_1 & , ([\cdot] + (y, \rho_1)) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 1 & , \varnothing & , ([\cdot] + (y, \rho_1)) :: & [\cdot]\ \rangle \\
\rightarrow & \langle y & , \rho_1 & , (1 + [\cdot]) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 5 & , \varnothing & , (1 + [\cdot]) :: & [\cdot]\ \rangle \\
\rightarrow & \langle 6 & , \varnothing & & , [\cdot]\ \rangle
\end{array}
$$

where $\rho_0 = [x \mapsto 1]$ and $\rho_1 = [x \mapsto 1, y \mapsto 5]$.

CESK Machine

A **CESK machine** extends the CEK machine with a **store** to represent the mapping from locations to values with the following form of states:

$$\sigma = \langle e, \rho, s, \kappa \rangle$$

- an expression e represents the current **control** (C).
- An environment (E) ρ for the mapping from variables to locations.
- A store (S) s for the mapping from locations to values:

$$s \in \mathbb{L} \rightharpoonup \mathbb{V}$$

- A stack κ represents the current **continuation** (K).

Using a CESK machine, we can support **mutations** or **side effects** in the language by updating the values of locations in the store.

Variants of CEK Machine

There are many variants of the CEK machine.

One possible variant is to use not only a **continuation stack** but also a **value stack** to represent the current state:

$$\sigma = \langle \kappa \mid\mid s \rangle$$

$$\begin{array}{lll} \text{Continuations} & \kappa ::= & \square \\ & & \mid (\rho \vdash e) :: \kappa \\ & & \mid (\texttt{+}) :: \kappa \\ & & \mid (\texttt{@}) :: \kappa \end{array}$$

$$\text{Value Stacks} \qquad s ::= \blacksquare \mid v :: s$$

Variants of CEK Machine

PLRG

The transition rules for this variant are as follows:

$$
\begin{array}{lcl}
\langle (\rho \vdash n) :: \kappa \mid\mid s \rangle & \to & \langle \kappa \mid\mid n :: s \rangle \\
\langle (\rho \vdash x) :: \kappa \mid\mid s \rangle & \to & \langle \kappa \mid\mid \rho(x) :: s \rangle \\
\langle (\rho \vdash e_1 \texttt{ + } e_2) :: \kappa \mid\mid s \rangle & \to & \langle (\rho \vdash e_1) :: (\rho \vdash e_2) :: (\texttt{+}) :: \kappa \mid\mid s \rangle \\
\langle (\texttt{+}) :: \kappa \mid\mid n_2 :: n_1 :: s \rangle & \to & \langle \kappa \mid\mid (n_1 + n_2) :: s \rangle \\
\langle (\rho \vdash \lambda x.e) :: \kappa \mid\mid s \rangle & \to & \langle \kappa \mid\mid \langle \lambda x.e, \rho \rangle :: s \rangle \\
\langle (\rho \vdash e_1(e_2)) :: \kappa \mid\mid s \rangle & \to & \langle (\rho \vdash e_1) :: (\rho \vdash e_2) :: (\texttt{@}) :: \kappa \mid\mid s \rangle \\
\langle (\texttt{@}) :: \kappa \mid\mid v_2 :: \langle \lambda x.e, \rho \rangle :: s \rangle & \to & \langle (\rho[x \mapsto v_2] \vdash e) :: \kappa \mid\mid s \rangle
\end{array}
$$

For example, a simple 1 + 2 can be evaluated as follows:

$$
\begin{array}{rl}
 & \langle (\varnothing \vdash 1 \texttt{ + } 2) :: \square \mid\mid \blacksquare \rangle \\
\to & \langle (\varnothing \vdash 1) :: (\varnothing \vdash 2) :: (\texttt{+}) :: \square \mid\mid \blacksquare \rangle \\
\to & \langle (\varnothing \vdash 2) :: (\texttt{+}) :: \square \mid\mid 1 :: \blacksquare \rangle \\
\to & \langle (\texttt{+}) :: \square \mid\mid 2 :: 1 :: \blacksquare \rangle \\
\to & \langle \square \mid\mid 3 :: \blacksquare \rangle
\end{array}
$$

Homework #1

https://github.com/ku-plrg-classroom/docs/tree/main/aaa551/abs-machine

- Please see above document on GitHub:
  1. Implement `reduce` function.
- The due date is 23:59 on Apr. 16 (Thu.).
- Please only submit **`Implementation.scala`** file to **LMS**.

Next Lecture

- Axiomatic Semantics

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