

Lecture 9 – Abstract Machines

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- 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

1. Continuations

A-Normal Form

2. CC Machine

Recall: Evaluation Contexts and λ -Calculus

CC Machine

SCC Machine

3. CK, CEK, CESK Machines

CK Machine

CEK Machine

CESK Machine

Variants of CEK Machine

A **continuation** is a representation of the “rest” of the computation.

A **continuation** is a representation of the “rest” of the computation.

For example, consider the following expression:

$$(1 + 3) * 5$$

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:

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: $4 * 5 = 20$) |

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.

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.

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: $4 * 5 = 20$) |

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: $4 * 5 = 20$)

```
{
  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: $4 * 5 = 20$)

```
{
  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: $4 * 5 = 20$)

```
{
  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: $4 * 5 = 20$)

```
{
  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
```

```

{
  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:

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

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

```
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)
```

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

```
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**.

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

```

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?

1. Continuations

A-Normal Form

2. CC Machine

Recall: Evaluation Contexts and λ -Calculus

CC Machine

SCC Machine

3. CK, CEK, CESK Machines

CK Machine

CEK Machine

CESK Machine

Variants of CEK Machine

Lambda calculus is the simplest functional language:

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

The following rule defines the **full β -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.

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

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

with the following evaluation context:

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

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

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

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

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

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

with the following inference rules:

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

and the evaluation context:

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

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

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)

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

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

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

$$\langle e_1 + e_2, E \rangle \quad \rightarrow \quad \langle e_1, E[[\cdot] + e_2] \rangle$$

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

$$\langle e_1 + e_2, E \rangle \quad \rightarrow \quad \langle e_1, E[[\cdot] + e_2] \rangle$$

$$\langle n_1, E[[\cdot] + e_2] \rangle \quad \rightarrow \quad \langle n_1 + e_2, E \rangle$$

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

$$\langle e_1 + e_2, E \rangle \rightarrow \langle e_1, E[[\cdot] + e_2] \rangle$$

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

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

$$\langle n_2, E[n_1 + [\cdot]] \rangle \rightarrow \langle n_1 + n_2, E \rangle$$

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

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

$$\langle e_1 + e_2, E \rangle \rightarrow \langle e_1, E[[\cdot] + e_2] \rangle$$

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

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

$$\langle n_2, E[n_1 + [\cdot]] \rangle \rightarrow \langle n_1 + n_2, E \rangle$$

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

$$\langle e_1 e_2, E \rangle \rightarrow \langle e_1, E[[\cdot] e_2] \rangle$$

$$\langle v, E[[\cdot] e] \rangle \rightarrow \langle v e, E \rangle$$

$$\langle v e, E \rangle \rightarrow \langle e, E[v [\cdot]] \rangle$$

$$\langle v_2, E[v_1 [\cdot]] \rangle \rightarrow \langle v_1 v_2, E \rangle$$

$$\langle (\lambda x.e) v, E \rangle \rightarrow \langle e[x \mapsto v], E \rangle$$

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

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

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

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

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

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

into the following single rule:

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

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

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

into the following single rule:

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

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

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

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

| | | |
|---|---|---|
| $\langle ((\lambda x. \lambda y. x + y) 1) (2 + 3) \rangle$ | , | $\langle \cdot \rangle$ |
| $\rightarrow \langle (\lambda x. \lambda y. x + y) 1 \rangle$ | , | $\langle \cdot \rangle (2 + 3) \rangle$ |
| $\rightarrow \langle \lambda x. \lambda y. x + y \rangle$ | , | $\langle [\cdot] 1 \rangle (2 + 3) \rangle$ |
| $\rightarrow \langle 1 \rangle$ | , | $\langle (\lambda x. \lambda y. x + y) [\cdot] \rangle (2 + 3) \rangle$ |
| $\rightarrow \langle \lambda y. 1 + y \rangle$ | , | $\langle \cdot \rangle (2 + 3) \rangle$ |
| $\rightarrow \langle 2 + 3 \rangle$ | , | $\langle \lambda y. 1 + y \rangle [\cdot] \rangle$ |
| $\rightarrow \langle 2 \rangle$ | , | $\langle \lambda y. 1 + y \rangle ([\cdot] + 3) \rangle$ |
| $\rightarrow \langle 3 \rangle$ | , | $\langle \lambda y. 1 + y \rangle (2 + [\cdot]) \rangle$ |
| $\rightarrow \langle 5 \rangle$ | , | $\langle \lambda y. 1 + y \rangle [\cdot] \rangle$ |
| $\rightarrow \langle 1 + 5 \rangle$ | , | $\langle \cdot \rangle$ |
| $\rightarrow \langle 1 \rangle$ | , | $\langle \cdot \rangle + 5 \rangle$ |
| $\rightarrow \langle 5 \rangle$ | , | $1 + [\cdot] \rangle$ |
| $\rightarrow \langle 6 \rangle$ | , | $\langle \cdot \rangle$ |

1. Continuations

A-Normal Form

2. CC Machine

Recall: Evaluation Contexts and λ -Calculus

CC Machine

SCC Machine

3. CK, CEK, CESK Machines

CK Machine

CEK Machine

CESK Machine

Variants of CEK Machine

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

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

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)

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

The transition rules for the CK machine are as follows:

$$\langle e_1 + e_2, \kappa \rangle \quad \rightarrow \quad \langle e_1, ([\cdot] + e_2) :: \kappa \rangle$$

$$\langle n_1, ([\cdot] + e_2) :: \kappa \rangle \quad \rightarrow \quad \langle e_2, (n_1 + [\cdot]) :: \kappa \rangle$$

$$\langle n_2, (n_1 + [\cdot]) :: \kappa \rangle \quad \rightarrow \quad \langle n_1 + n_2, \kappa \rangle$$

$$\langle e_1 e_2, \kappa \rangle \quad \rightarrow \quad \langle e_1, ([\cdot] e_2) :: \kappa \rangle$$

$$\langle \lambda x.e, ([\cdot] e) :: \kappa \rangle \quad \rightarrow \quad \langle e, ((\lambda x.e) [\cdot]) :: \kappa \rangle$$

$$\langle v, ((\lambda x.e) [\cdot]) :: \kappa \rangle \quad \rightarrow \quad \langle e[x \mapsto v], \kappa \rangle$$

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

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

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

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:

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

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

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

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

The transition rules for the CEK machine are as follows:

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

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.

| | | |
|---|---------------|---|
| $\langle ((\lambda x. \lambda y. x + y) 1) (2 + 3) \rangle$ | $, \emptyset$ | $, [\cdot]$ |
| $\rightarrow \langle (\lambda x. \lambda y. x + y) 1 \rangle$ | $, \emptyset$ | $, ([\cdot] (2 + 3, \emptyset)) :: [\cdot]$ |
| $\rightarrow \langle \lambda x. \lambda y. x + y \rangle$ | $, \emptyset$ | $, ([\cdot] (1, \emptyset)) :: ([\cdot] (2 + 3, \emptyset)) :: [\cdot]$ |
| $\rightarrow \langle 1 \rangle$ | $, \emptyset$ | $, ((\lambda x. \lambda y. x + y, \emptyset) [\cdot]) :: ([\cdot] (2 + 3, \emptyset)) :: [\cdot]$ |
| $\rightarrow \langle \lambda y. x + y \rangle$ | $, \rho_0$ | $, ([\cdot] (2 + 3, \emptyset)) :: [\cdot]$ |
| $\rightarrow \langle 2 + 3 \rangle$ | $, \emptyset$ | $, ((\lambda y. x + y, \rho_0) [\cdot]) :: [\cdot]$ |
| $\rightarrow \langle 2 \rangle$ | $, \emptyset$ | $, ([\cdot] + (3, \emptyset)) :: ((\lambda y. x + y, \rho_0) [\cdot]) :: [\cdot]$ |
| $\rightarrow \langle 3 \rangle$ | $, \emptyset$ | $, (2 + [\cdot]) :: ((\lambda y. x + y, \rho_0) [\cdot]) :: [\cdot]$ |
| $\rightarrow \langle 5 \rangle$ | $, \emptyset$ | $, ((\lambda y. x + y, \rho_0) [\cdot]) :: [\cdot]$ |
| $\rightarrow \langle x + y \rangle$ | $, \rho_1$ | $, [\cdot]$ |
| $\rightarrow \langle x \rangle$ | $, \rho_1$ | $, ([\cdot] + (y, \rho_1)) :: [\cdot]$ |
| $\rightarrow \langle 1 \rangle$ | $, \emptyset$ | $, ([\cdot] + (y, \rho_1)) :: [\cdot]$ |
| $\rightarrow \langle y \rangle$ | $, \rho_1$ | $, (1 + [\cdot]) :: [\cdot]$ |
| $\rightarrow \langle 5 \rangle$ | $, \emptyset$ | $, (1 + [\cdot]) :: [\cdot]$ |
| $\rightarrow \langle 6 \rangle$ | $, \emptyset$ | $, [\cdot]$ |

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

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:

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

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

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:

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

- 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.

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:

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

$$\begin{array}{l} \text{Continuations} \quad \kappa ::= \square \\ \quad \quad \quad \quad | (\rho \vdash e) :: \kappa \\ \quad \quad \quad \quad | (+) :: \kappa \\ \quad \quad \quad \quad | (@) :: \kappa \end{array}$$

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

The transition rules for this variant are as follows:

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

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

$$\begin{aligned}
 &\langle (\emptyset \vdash 1 + 2) :: \square \parallel \blacksquare \rangle \\
 \rightarrow &\langle (\emptyset \vdash 1) :: (\emptyset \vdash 2) :: (+) :: \square \parallel \blacksquare \rangle \\
 \rightarrow &\langle (\emptyset \vdash 2) :: (+) :: \square \parallel 1 :: \blacksquare \rangle \\
 \rightarrow &\langle (+) :: \square \parallel 2 :: 1 :: \blacksquare \rangle \\
 \rightarrow &\langle \square \parallel 3 :: \blacksquare \rangle
 \end{aligned}$$

<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](#).

- Axiomatic Semantics

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