

Lecture 11 – Mutable Variables

COSE212: Programming Languages

Jihyeok Park



2023 Fall

- Mutation makes it possible to update the **contents** of a data structure or a variable after its creation.
 - **Mutable data structures**
 - **Mutable variables**
- **Mutable Data Structures** – Mutable Boxes
- BFAE – FAE with Mutable Boxes
 - Evaluation with Memories
- In this lecture, we will learn **Mutable Variables**
- **MFAE – FAE with Mutable Variables**
 - Concrete and Abstract Syntax
 - Interpreter and Natural Semantics

1. Mutable Variables
2. MFAE – FAE with Mutable Variables
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for MFAE
 - Evaluation with Memories
 - Interpreter and Natural Semantics
 - Mutable Variable
 - Identifier Lookup
 - Function Application
 - Assignment
4. Call-by-Value vs. Call-by-Reference

1. Mutable Variables
2. MFAE – FAE with Mutable Variables
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for MFAE
 - Evaluation with Memories
 - Interpreter and Natural Semantics
 - Mutable Variable
 - Identifier Lookup
 - Function Application
 - Assignment
4. Call-by-Value vs. Call-by-Reference

A **mutable variable** is a variable whose value can be changed after its initialization.

Let's define mutable variables in Scala:

```
// A mutable variable `x` of type `Int` with 1
var x: Int = 1
x + 2          // 1 + 2 == 3 : Int

// We can reassign a mutable variable `x`
x = 2          // x == 2
x + 2          // 2 + 2 == 4 : Int

// The function `f` is impure because it uses a mutable variable `y`
var y: Int = 1
def f(x: Int): Int = x + y
f(5)           // 5 + 1 == 6 : Int
y = 3
f(5)           // 5 + 3 == 8 : Int
```

1. Mutable Variables
2. MFAE – FAE with Mutable Variables
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for MFAE
 - Evaluation with Memories
 - Interpreter and Natural Semantics
 - Mutable Variable
 - Identifier Lookup
 - Function Application
 - Assignment
4. Call-by-Value vs. Call-by-Reference

Now, let's extend FAE into MFAE to support **mutable variables**.

```

/* MFAE */
var x = 5;
x;           // 5
x = 8;
x           // 8

```

```

/* MFAE */
var y = 1;
var f = x => { x = x + y; x * x };
f(5);       // (5 + 1) * (5 + 1) = 36
y = 3;
f(5);       // (5 + 3) * (5 + 3) = 64

```

For MFAE, we need to extend **expressions** of FAE with

- ① **mutable variables** (`var`) rather than immutable variables (`val`) (all variables, including parameters, are mutable in MFAE)
- ② **assignment** (`=`)
- ③ **sequence** of expressions (right-associative: e.g., $e_1; e_2; e_3 \equiv (e_1; (e_2; e_3))$)

```
// expressions
<expr> ::= ...
        | "var" <id> "=" <expr> ";" <expr>
        | <id> "=" <expr>
        | <expr> ";" <expr>
```

For MFAE, we need to extend **expressions** of FAE with

- ① **mutable variables** (`var`) rather than immutable variables (`val`)
(all variables, including parameters, are mutable in MFAE)
- ② **assignment** (`=`)
- ③ **sequence** of expressions
(right-associative: e.g., $e_1; e_2; e_3 \equiv (e_1; (e_2; e_3))$)

Let's define the **abstract syntax** of MFAE in BNF:

Expressions $\mathbb{E} \ni e ::= \dots$

var x=e; e	(Var)
x=e	(Assign)
e; e	(Seq)

```
enum Expr:
  ...
  // mutable variable definition
  case Var(name: String, init: Expr, body: Expr)
  // variable assignment
  case Assign(name: String, expr: Expr)
  // sequence
  case Seq(left: Expr, right: Expr)
```

1. Mutable Variables
2. MFAE – FAE with Mutable Variables
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for MFAE
 - Evaluation with Memories
 - Interpreter and Natural Semantics
 - Mutable Variable
 - Identifier Lookup
 - Function Application
 - Assignment
4. Call-by-Value vs. Call-by-Reference

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```
/* MFAE */ *
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);
y = 3;
f(5);
```

$$\sigma = [$$

$$]$$

$$A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$
$$M = \begin{array}{|c|c|c|c|c|} \hline & & & & \dots \\ \hline \end{array}$$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);
y = 3;
f(5);

```

*

$$\sigma = [$$

$$y \mapsto a_0$$

$$]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & & & & \dots \\ \hline \end{array}$$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);
y = 3;
f(5);

```

*

$$\sigma = [$$

$$y \mapsto a_0$$

$$f \mapsto a_1$$

$$]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & v & & & \dots \\ \hline \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);
y = 3;
f(5);
    
```

$$\sigma = [$$

$$y \mapsto a_0$$

$$x \mapsto a_2$$

$$]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & v & 5 & & \dots \\ \hline \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y; /* 5 + 1 */ *
  x * x
};
f(5);
y = 3;
f(5);
    
```

$$\sigma = \left[\begin{array}{l} y \mapsto a_0 \\ x \mapsto a_2 \end{array} \right]$$

$$\begin{array}{l} \mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots \\ M = \begin{array}{|c|c|c|c|c|} \hline 1 & v & 6 & & \dots \\ \hline \end{array} \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y; /* 5 + 1 */
  x * x      /* 6 * 6 */ *
};
f(5);
y = 3;
f(5);

```

$$\sigma = [$$

$$y \mapsto a_0$$

$$x \mapsto a_2$$

$$]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & v & 6 & & \dots \\ \hline \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);           /* 36 */ *
y = 3;
f(5);

```

$$\sigma = [$$

$$y \mapsto a_0$$

$$f \mapsto a_1$$

$$]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & v & 6 & & \dots \\ \hline \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);           /* 36 */
y = 3;          *
f(5);

```

$$\sigma = [\begin{array}{l} y \mapsto a_0 \\ f \mapsto a_1 \end{array}]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 3 & v & 6 & & \dots \\ \hline \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);           /* 36 */
y = 3;
f(5);
  
```

$$\sigma = [$$

$$y \mapsto a_0$$

$$x \mapsto a_3$$

$$]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 3 & v & 6 & 5 & \dots \\ \hline \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y; /* 5 + 3 */ *
  x * x
};
f(5);      /* 36 */
y = 3;
f(5);
    
```

$$\sigma = \left[\begin{array}{l} y \mapsto a_0 \\ x \mapsto a_3 \end{array} \right]$$

$$M = \begin{array}{c} \mathbb{A} \\ \text{---} \\ M \end{array} : \begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & \dots \\ \boxed{3} & \boxed{v} & \boxed{6} & \boxed{8} & \boxed{\dots} \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y; /* 5 + 3 */
  x * x      /* 8 * 8 */ *
};
f(5);      /* 36 */
y = 3;
f(5);
  
```

$$\sigma = [$$

$$y \mapsto a_0$$

$$x \mapsto a_3$$

$$]$$

$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \begin{array}{|c|c|c|c|c|} \hline 3 & v & 6 & 8 & \dots \\ \hline \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

Example

We can evaluate MFAE expressions with **memories** similar to BFAE.

Let's see how to evaluate the following MFAE expression:

```

/* MFAE */
var y = 1;
var f = x => {
  x = x + y;
  x * x
};
f(5);           /* 36 */
y = 3;
f(5);           /* 64 */

```

$$\sigma = [$$

$$y \mapsto a_0$$

$$f \mapsto a_1$$

$$]$$

$$M = \begin{array}{c} \mathbb{A} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \\ \text{---} \end{array} : \begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & \dots \\ \boxed{3} & \boxed{v} & \boxed{6} & \boxed{8} & \boxed{\dots} \end{array}$$

where $v = \langle \lambda x. x = x + y; x * x, [y \mapsto a_0] \rangle$

For MFAE, we need to 1) implement the **interpreter** with environments and **memories** by passing the updated memory in the result:

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = ???
```

```
type Env = Map[String, Addr]
type Addr = Int
type Mem = Map[Addr, Value]
```

```
enum Value:
  case NumV(n: BigInt)
  case CloV(p: String, b: Expr, e: Env)
```

and 2) define the **natural semantics** with environments and **memories** by passing the updated memory in the result:

$$\sigma, M \vdash e \Rightarrow v, M$$

Environments $\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{A}$ (Env)

Addresses $a \in \mathbb{A}$ (Addr)

Memories $M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$ (Mem)

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Var(name, init, body) =>
    val (iv, imem) = interp(init, env, mem)
    val addr = malloc(imem)
    interp(body, env + (name -> addr), imem + (addr -> iv))
```

$$\sigma, M \vdash e \Rightarrow v, M$$

$$\text{Var} \frac{\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \quad a \notin \text{Domain}(M_1) \quad \sigma[x \mapsto a], M_1[a \mapsto v_1] \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash \text{var } x=e_1; e_2 \Rightarrow v_2, M_2}$$

We learned one way to implement malloc in the previous lecture:

```
def malloc(mem: Mem): Addr = mem.keySet.maxOption.fold(0)(_ + 1)
```



```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Id(name) => (mem(lookupId(env, name)), mem)

def lookupId(env: Env, name: String): Addr =
  env.getOrElse(name, error(s"free identifier: $name"))
```

$$\sigma, M \vdash e \Rightarrow v, M$$

$$\text{Id} \frac{x \in \text{Domain}(\sigma)}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M}$$

```

def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case App(fun, arg) =>
    val (fv, fmem) = interp(fun, env, mem)
    fv match
      case CloV(param, body, fenv) =>
        val (av, amem) = interp(arg, env, fmem)
        val addr = malloc(amem)
        interp(body, fenv + (param -> addr), amem + (addr -> av))
      case _ =>
        error(s"not a function: ${fv.str}")

```

$$\sigma, M \vdash e \Rightarrow v, M$$

$$\text{App} \frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x. e_3, \sigma' \rangle, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2 \quad a \notin \text{Domain}(M_2) \quad \sigma'[x \mapsto a], M_2[a \mapsto v_2] \vdash e_3 \Rightarrow v_3, M_3}{\sigma, M \vdash e_1(e_2) \Rightarrow v_3, M_3}$$

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Assign(name, expr) =>
    val (ev, emem) = interp(expr, env, mem)
    (ev, emem + (lookupId(env, name) -> ev))
```

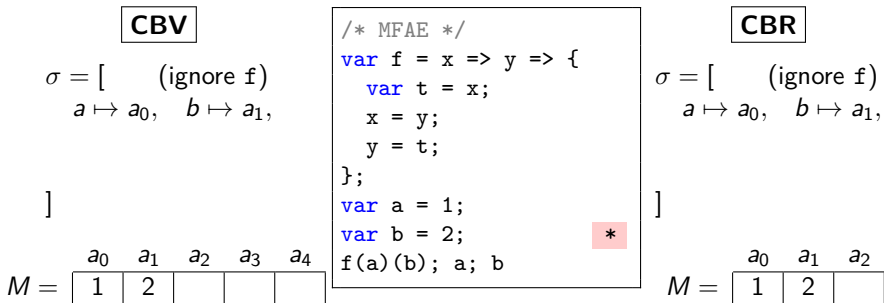
$$\sigma, M \vdash e \Rightarrow v, M$$

$$\text{Assign} \frac{\sigma, M \vdash e \Rightarrow v, M' \quad x \in \text{Domain}(\sigma)}{\sigma, M \vdash x=e \Rightarrow v, M'[\sigma(x) \mapsto v]}$$

1. Mutable Variables
2. MFAE – FAE with Mutable Variables
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for MFAE
 - Evaluation with Memories
 - Interpreter and Natural Semantics
 - Mutable Variable
 - Identifier Lookup
 - Function Application
 - Assignment
4. Call-by-Value vs. Call-by-Reference

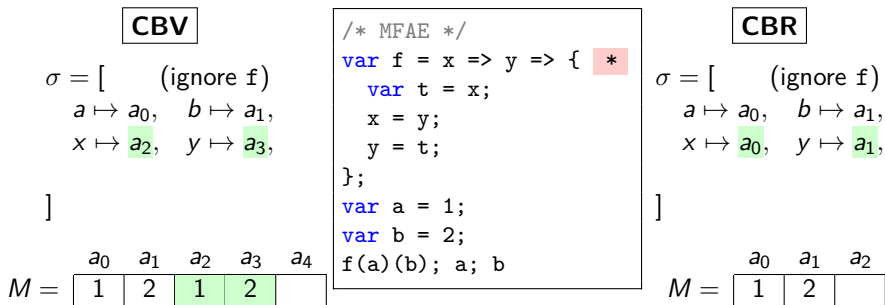
The current semantics of MFAE is based on the **call-by-value (CBV)** evaluation strategy, because the argument expression is always evaluated and the result **value** is passed to the parameter.

However, we can define the semantics of MFAE in another way by using the **call-by-reference (CBR)** evaluation strategy instead; if the argument expression is an identifier, the parameter points to its **address**.



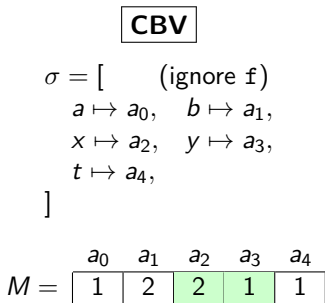
The current semantics of MFAE is based on the **call-by-value (CBV)** evaluation strategy, because the argument expression is always evaluated and the result **value** is passed to the parameter.

However, we can define the semantics of MFAE in another way by using the **call-by-reference (CBR)** evaluation strategy instead; if the argument expression is an identifier, the parameter points to its **address**.



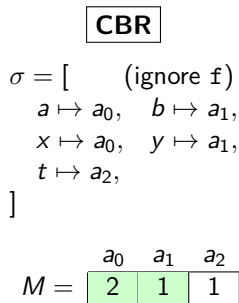
The current semantics of MFAE is based on the **call-by-value (CBV)** evaluation strategy, because the argument expression is always evaluated and the result **value** is passed to the parameter.

However, we can define the semantics of MFAE in another way by using the **call-by-reference (CBR)** evaluation strategy instead; if the argument expression is an identifier, the parameter points to its **address**.



```

/* MFAE */
var f = x => y => {
  var t = x;
  x = y;
  y = t;
};
var a = 1;
var b = 2;
f(a)(b); a; b
    
```



The current semantics of MFAE is based on the **call-by-value (CBV)** evaluation strategy, because the argument expression is always evaluated and the result **value** is passed to the parameter.

However, we can define the semantics of MFAE in another way by using the **call-by-reference (CBR)** evaluation strategy instead; if the argument expression is an identifier, the parameter points to its **address**.

<div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 10px;">CBV</div> $\sigma = [\quad \text{(ignore f)}$ $a \mapsto a_0, \quad b \mapsto a_1,$ $]$ $M =$ <table border="1" style="display: inline-table; border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 5px;">a_0</td> <td style="padding: 5px;">a_1</td> <td style="padding: 5px;">a_2</td> <td style="padding: 5px;">a_3</td> <td style="padding: 5px;">a_4</td> </tr> <tr> <td style="padding: 5px; background-color: #e0ffe0;">1</td> <td style="padding: 5px; background-color: #e0ffe0;">2</td> <td style="padding: 5px;">2</td> <td style="padding: 5px;">1</td> <td style="padding: 5px;">1</td> </tr> </table>	a_0	a_1	a_2	a_3	a_4	1	2	2	1	1	<div style="border: 1px solid black; padding: 10px; margin: 10px auto; width: 80%;"> <pre> /* MFAE */ var f = x => y => { var t = x; x = y; y = t; }; var a = 1; var b = 2; f(a)(b); a; b </pre> </div>	<div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 10px;">CBR</div> $\sigma = [\quad \text{(ignore f)}$ $a \mapsto a_0, \quad b \mapsto a_1,$ $]$ $M =$ <table border="1" style="display: inline-table; border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 5px;">a_0</td> <td style="padding: 5px;">a_1</td> <td style="padding: 5px;">a_2</td> </tr> <tr> <td style="padding: 5px; background-color: #e0ffe0;">2</td> <td style="padding: 5px; background-color: #e0ffe0;">1</td> <td style="padding: 5px; background-color: #e0ffe0;">1</td> </tr> </table>	a_0	a_1	a_2	2	1	1
a_0	a_1	a_2	a_3	a_4														
1	2	2	1	1														
a_0	a_1	a_2																
2	1	1																

We can define the semantics of MFAE with the **call-by-reference (CBR)** evaluation strategy by adding the following case:

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case App(fun, arg) =>
    val (fv, fmem) = interp(fun, env, mem)
    fv match
      case CloV(param, body, fenv) => arg match
        case Id(name) =>
          val addr = lookupId(env, name)
          interp(body, fenv + (param -> addr), fmem)
        case _ => ...
      case _ => error(s"not a function: ${fv.str}")
  ...
```

$$\text{App}_x \frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x'. e_2, \sigma' \rangle, M_1 \quad x \in \text{Domain}(\sigma) \quad \sigma'[x' \mapsto \sigma(x)], M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1(x) \Rightarrow v_2, M_2}$$

- Please see this document¹ on GitHub.
 - Implement `interp` function.
 - Implement `interpCBR` function.
- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

¹<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/mfae>.

1. Mutable Variables
2. MFAE – FAE with Mutable Variables
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for MFAE
 - Evaluation with Memories
 - Interpreter and Natural Semantics
 - Mutable Variable
 - Identifier Lookup
 - Function Application
 - Assignment
4. Call-by-Value vs. Call-by-Reference

- Garbage Collection

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