## Lecture 11 – Mutable Variables

COSE212: Programming Languages

Jihyeok Park



2025 Fall

#### Recall



- Mutation makes it possible to change the state of a program by updating the contents of a data structure or a variable.
  - 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

### Contents



- 1. Mutable Variables
- 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
- 4. Call-by-Value vs. Call-by-Reference

Assignment

## Contents



#### 1. Mutable Variables

MFAE – FAE with Mutable Variables
 Concrete Syntax
 Abstract Syntax

3. Interpreter and Natural Semantics for MFAE

Evaluation with Memories
Interpreter and Natural Semantic
Mutable Variable
Identifier Lookup
Function Application
Assignment

4. Call-by-Value vs. Call-by-Reference

### Mutable Variables



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: Tnt = 1
// 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.
v = 3
f(5)
             // 5 + 3 == 8 : Int
```

## Contents



- 1. Mutable Variables
- MFAE FAE with Mutable Variables
   Concrete Syntax
   Abstract Syntax
- 3. Interpreter and Natural Semantics for MF.
  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

- mutable variables (var) rather than immutable variables (val)
   (all variables, including parameters, are mutable in MFAE)
- **2** assignment (=) (right-associative: e.g., x = y = e is equivalent to x = (y = e))
- **3 sequence** of expressions

## Concrete Syntax



For MFAE, we need to extend expressions of FAE with

- mutable variables (var) rather than immutable variables (val) (all variables, including parameters, are mutable in MFAE)
- 2 assignment (=) (right-associative: e.g., x=y=e is equivalent to x=(y=e))
- 3 sequence of expressions





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

```
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)
```

## Contents



- 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 represent mutable variables by assigning different **addresses** to each variable in the environment and storing their values in the **memory**.

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

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





We can represent mutable variables by assigning different **addresses** to each variable in the environment and storing their values in the **memory**.

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

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

## **Evaluation with Memories**



We can represent mutable variables by assigning different **addresses** to each variable in the environment and storing their values in the **memory**.

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

 $\sigma = [$ 





We can represent mutable variables by assigning different **addresses** to each variable in the environment and storing their values in the **memory**.

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

 $\sigma = [$ 





We can represent mutable variables by assigning different **addresses** to each variable in the environment and storing their values in the **memory**.

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

```
\sigma = \begin{bmatrix} & & & \\ & \mathbf{x} \mapsto a_0 & & \\ & & \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \begin{bmatrix} 8 & & & & \\ & & & & \\ \end{bmatrix} \quad \dots
```





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





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

```
\sigma = \begin{bmatrix} \\ y \mapsto a_0 \end{bmatrix}
A : a_0 \ a_1 \ a_2 \ a_3 \ \dots
M = \boxed{1} \ \boxed{\dots}
```





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

```
\sigma = \begin{bmatrix} & & & & \\ & y \mapsto a_0 & \\ & f \mapsto a_1 & & \\ \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \begin{bmatrix} 1 & v & & & \dots \end{bmatrix}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





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

```
\sigma = \begin{bmatrix} y \mapsto a_0 \\ x \mapsto a_2 \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \boxed{1 \quad v \quad 5 \quad \dots}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





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

```
\sigma = \begin{bmatrix} y \mapsto a_0 \\ x \mapsto a_2 \end{bmatrix}

A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots

M = \boxed{1 \quad v \quad 6 \quad \dots}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





$$\sigma = \begin{bmatrix} y \mapsto a_0 \\ x \mapsto a_2 \end{bmatrix}$$

$$A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots$$

$$M = \boxed{1 \quad v \quad 6 \quad \dots}$$

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





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

```
\sigma = \begin{bmatrix} & & & & \\ & y \mapsto a_0 & \\ & f \mapsto a_1 & \\ & & \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \begin{bmatrix} 1 & v & 6 & & \dots \end{bmatrix}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





```
/* 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 = \boxed{3 \quad v \quad 6 \quad \dots}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





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

```
\sigma = \begin{bmatrix} & & & & \\ & y \mapsto a_0 & & \\ & x \mapsto a_3 & & \\ \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \begin{bmatrix} 3 & v & 6 & 5 & \dots \end{bmatrix}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





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

```
\sigma = \begin{bmatrix} y \mapsto a_0 \\ x \mapsto a_3 \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \boxed{3 \quad v \quad 6 \quad 8 \quad \dots}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





```
\sigma = \begin{bmatrix} y \mapsto a_0 \\ x \mapsto a_3 \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \boxed{3 \quad v \quad 6 \quad 8 \quad \dots}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





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

```
\sigma = \begin{bmatrix} & & & \\ & y \mapsto a_0 & \\ & f \mapsto a_1 & \\ & \end{bmatrix}
A : a_0 \quad a_1 \quad a_2 \quad a_3 \quad \dots
M = \boxed{3 \quad v \quad 6 \quad 8 \quad \dots}
```

where 
$$v = \langle \lambda \mathbf{x}.(\mathbf{x} = \mathbf{x} + \mathbf{y}; \mathbf{x} * \mathbf{x}), [\mathbf{y} \mapsto a_0] \rangle$$





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:

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

$$\begin{array}{lll} \text{Environments} & \sigma & \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{A} & (\texttt{Env}) \\ \text{Addresses} & a & \in \mathbb{A} & (\texttt{Addr}) \\ \text{Memories} & M & \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V} & (\texttt{Mem}) \end{array}$$

## Mutable Variable



```
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))
```

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

$$\operatorname{Var} \frac{ \begin{matrix} \sigma, M \vdash e_1 \Rightarrow v_1, M_1 \\ \sigma[x \mapsto a], M_1[a \mapsto v_1] \vdash e_2 \Rightarrow v_2, M_2 \end{matrix} }{ \sigma, M \vdash \operatorname{var} \ x = e_1; \ e_2 \Rightarrow v_2, M_2 }$$

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

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

## Identifier Lookup



```
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"))
```

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

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

## **Function Application**



```
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}")
```

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

$$\operatorname{App} \frac{ \begin{array}{ccc} \sigma, M \vdash e_1 \Rightarrow \langle \lambda x. e_3, \sigma' \rangle, M_1 & \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2 \\ a \notin \operatorname{Domain}(M_2) & \sigma'[x \mapsto a], M_2[a \mapsto v_2] \vdash e_3 \Rightarrow v_3, M_3 \\ \hline \sigma, M \vdash e_1(e_2) \Rightarrow v_3, M_3 \end{array} }$$

## Assignment



```
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))
```

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

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

## Contents



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



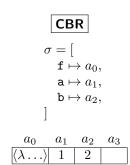
The current semantics of MFAE is based on the **call-by-value (CBV)** evaluation strategy, because the argument expression is always evaluated and the result **value** is passed to the parameter.



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.

```
 \begin{array}{c|c} \textbf{CBV} \\ \sigma = [\\ \textbf{f} \mapsto a_0,\\ \textbf{a} \mapsto a_1,\\ \textbf{b} \mapsto a_2,\\ \end{bmatrix} \\ \hline a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \\ \hline \langle \lambda \ldots \rangle \mid 1 \mid 2 \mid \qquad \mid \qquad \\ \hline \end{array}
```

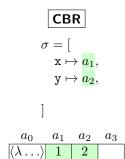
```
/* MFAE */
var f = x => y => {
  var t = x;
  x = y;
  y = t;
};
var a = 1;
var b = 2;
f(a)(b); a; b
```





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.

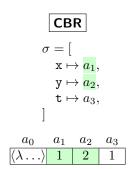
```
/* MFAE */
var f = x => y => { *
  var t = x;
  x = y;
  y = t;
};
var a = 1;
var b = 2;
f(a)(b); a; b
```





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.

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

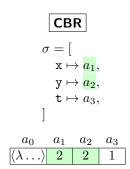




The current semantics of MFAE is based on the **call-by-value (CBV)** evaluation strategy, because the argument expression is always evaluated and the result **value** is passed to the parameter.

```
\begin{array}{c}
| \textbf{CBV} | \\
\sigma = [ \\
x \mapsto a_3, \\
y \mapsto a_4, \\
t \mapsto a_5, \\
] \\
\hline
a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \\
\hline
a_0 \quad a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5
\end{array}
```

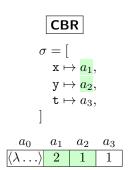
```
/* MFAE */
var f = x => y => {
  var t = x;
  x = y;
  y = t;
};
var a = 1;
var b = 2;
f(a)(b); a; b
```





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.

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





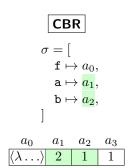
The current semantics of MFAE is based on the **call-by-value (CBV)** evaluation strategy, because the argument expression is always evaluated and the result **value** is passed to the parameter.

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**.

```
\sigma = [ \\ \mathbf{f} \mapsto a_0, \\ \mathbf{a} \mapsto a_1, \\ \mathbf{b} \mapsto a_2, \\ ]
\begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ \langle \lambda \dots \rangle & 1 & 2 & 2 & 1 & 1 \end{bmatrix}
```

**CBV** 

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







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)
    fy 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}")
    . . .
```

$$\operatorname{App}_{x} \frac{\sigma, M \vdash e_{1} \Rightarrow \langle \lambda x'. e_{2}, \sigma' \rangle, M_{1}}{\sigma'[x' \mapsto \sigma(x)], M_{1} \vdash e_{2} \Rightarrow v_{2}, M_{2}}{\sigma, M \vdash e_{1}(x) \Rightarrow v_{2}, M_{2}}$$

## Exercise #7



#### https://github.com/ku-plrg-classroom/docs/tree/main/cose212/mfae

- Please see above document 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.

# Summary



- 1. Mutable Variables
- 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

## Next Lecture



• Garbage Collection

Jihyeok Park
 jihyeok\_park@korea.ac.kr
https://plrg.korea.ac.kr