Lecture 11 – Mutable Variables

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

- Mutation makes it possible to update the **contents** of a data structure or a variable after its creation.
    - **Mutable data structures**
    - **Mutable variables**

Recall

- 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

Recall

- 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

Contents

Contents

Mutable Variables

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

Mutable Variables

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

Let's define mutable variables in Scala:

```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
```

Contents

MFAE – FAE with Mutable Variables

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 };
f(5);               // 5 + 1 = 6
y = 3;
f(5);               // 5 + 3 = 8
```

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

1. **mutable variables** (`var`) rather than immutable variables (`val`)
2. **assignment** (=)
3. **sequence** of expressions

Concrete Syntax

```
// 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`)
2. **assignment** (=)
3. **sequence** of expressions

Abstract Syntax

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

$$
\begin{array}{llll}
\text{Expressions} & \mathbb{E} \ni e ::= \ldots & & \\
& \mid \texttt{var}\ x\texttt{=}e\texttt{;}\ e & (\texttt{Var}) \\
& \mid x\texttt{=}e & (\texttt{Assign}) \\
& \mid e\texttt{;}\ e & (\texttt{Seq})
\end{array}
$$

```
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

Evaluation with Memories

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);
```

$\sigma = [$

$]$

$$\begin{array}{ccccccc} \mathbb{A} & : & a_0 & a_1 & a_2 & a_3 & \ldots \\ M & = & \boxed{} & \boxed{} & \boxed{} & \boxed{} & \boxed{\ldots} \end{array}$$

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	a_1	a_2	a_3	$\ldots$
M	$=$	1				$\ldots$

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	a_1	a_2	a_3	$\ldots$
M	$=$	1	v			$\ldots$

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

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);
y = 3;
f(5);
```

$$\sigma = [\ y \mapsto a_0,\ f \mapsto a_1,\ x \mapsto a_2\ ]$$

$\mathbb{A}$:	a_0	a_1	a_2	a_3	$\ldots$
M =	1	v	5		$\ldots$

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

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
};
f(5);
y = 3;
f(5);
```

$$\sigma = [\; y \mapsto a_0,\; f \mapsto a_1,\; x \mapsto a_2 \;]$$

$\mathbb{A}$:	a_0	a_1	a_2	a_3	...
M =	1	v	6		...

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

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);
```

$$\sigma = [\; y \mapsto a_0,\; f \mapsto a_1,\; x \mapsto a_2 \;]$$

$\mathbb{A}$:	a_0	a_1	a_2	a_3	$\ldots$
M =	1	v	6		$\ldots$

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

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);
```

$$\sigma = [\; y \mapsto a_0,\; f \mapsto a_1 \;]$$

$$\begin{array}{ccccccc} \mathbb{A} & : & a_0 & a_1 & a_2 & a_3 & \ldots \\ M & = & \boxed{1} & \boxed{v} & \boxed{6} & \boxed{} & \boxed{\ldots} \end{array}$$

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

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	a_1	a_2	a_3	...
M =	3	v	6		...

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

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);
```

$$\sigma = [\; y \mapsto a_0,\; f \mapsto a_1,\; x \mapsto a_3 \;]$$

$\mathbb{A}$	:	a_0	a_1	a_2	a_3	...
M	=	3	v	6	5	...

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

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
};
f(5);          /* 36 */
y = 3;
f(5);
```

$$\sigma = [\; y \mapsto a_0,\; f \mapsto a_1,\; x \mapsto a_3 \;]$$

$$\begin{array}{cccccccc} \mathbb{A} & : & a_0 & a_1 & a_2 & a_3 & \ldots \\ M & = & \boxed{3} & \boxed{v} & \boxed{6} & \boxed{8} & \boxed{\ldots} \end{array}$$

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

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);
```

$$\sigma = [\; y \mapsto a_0 \quad f \mapsto a_1 \quad x \mapsto a_3 \;]$$

$\mathbb{A}$	:	a_0	a_1	a_2	a_3	$\ldots$
M	$=$	3	v	6	8	$\ldots$

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

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 */      *
```

$$\sigma = [\; y \mapsto a_0 \;\; f \mapsto a_1 \;]$$

$$\begin{array}{ccccccc} \mathbb{A} & : & a_0 & a_1 & a_2 & a_3 & \ldots \\ M & = & \boxed{3} & \boxed{v} & \boxed{6} & \boxed{8} & \boxed{\ldots} \end{array}$$

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

Interpreter and Natural Semantics

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[Var, Addr]
type Addr = Int
type Mem  = Map[Addr, Value]
```

Interpreter and Natural Semantics

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[Var, Addr]
type Addr = Int
type Mem  = Map[Addr, Value]
```

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

$$\boxed{\sigma, M \vdash e \Rightarrow v, M}$$

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

Mutable Variable

```scala
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))
```

$$\boxed{\sigma, M \vdash e \Rightarrow v, M}$$

$$\texttt{Var}\ \frac{a \notin \mathrm{Domain}(M_1) \qquad \begin{array}{c}\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \\ \sigma[x \mapsto a], M_1[a \mapsto v_1] \vdash e_2 \Rightarrow v_2, M_2\end{array}}{\sigma, M \vdash \texttt{val}\ x\texttt{=}e_1\texttt{;}\ e_2 \Rightarrow v_2, M_2}$$

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

```scala
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"))
```

$$\boxed{\sigma, M \vdash e \Rightarrow v, M}$$

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

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}")
```

$$\boxed{\sigma, M \vdash e \Rightarrow v, M}$$

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

Function Application

PLRG

```
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))
```

$$\boxed{\sigma, M \vdash e \Rightarrow v, M}$$

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

Contents

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

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

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

The current semantics of MFAE is based on the **call-by-value (CBV)** 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)** strategy instead; if the argument expression is an identifier, the parameter points to its address.

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

The current semantics of MFAE is based on the **call-by-value (CBV)** 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)** strategy instead; if the argument expression is an identifier, the parameter points to its address.

CBV

$$\sigma = [\; a \mapsto a_0, \quad b \mapsto a_1, \;]$$

$$M = \begin{array}{c|c|c|c|c|} a_0 & a_1 & a_2 & a_3 & a_4 \\ \hline 1 & 2 & & & \end{array}$$

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

CBR

$$\sigma = [\; a \mapsto a_0, \quad b \mapsto a_1, \;]$$

$$M = \begin{array}{c|c|c|} a_0 & a_1 & a_2 \\ \hline 1 & 2 & \end{array}$$

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

The current semantics of MFAE is based on the **call-by-value (CBV)** 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)** strategy instead; if the argument expression is an identifier, the parameter points to its address.

CBV

$$\sigma = [\; a \mapsto a_0,\; b \mapsto a_1,\; x \mapsto a_2,\; y \mapsto a_3,\;]$$

$$M = \begin{array}{c|c|c|c|c|} \multicolumn{1}{c}{a_0} & \multicolumn{1}{c}{a_1} & \multicolumn{1}{c}{a_2} & \multicolumn{1}{c}{a_3} & \multicolumn{1}{c}{a_4} \\ \hline 1 & 2 & 1 & 2 & \\ \hline \end{array}$$

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

CBR

$$\sigma = [\; a \mapsto a_0,\; b \mapsto a_1,\; x \mapsto a_0,\; y \mapsto a_1,\;]$$

$$M = \begin{array}{c|c|c|} \multicolumn{1}{c}{a_0} & \multicolumn{1}{c}{a_1} & \multicolumn{1}{c}{a_2} \\ \hline 1 & 2 & \\ \hline \end{array}$$

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

The current semantics of MFAE is based on the **call-by-value (CBV)** 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)** strategy instead; if the argument expression is an identifier, the parameter points to its address.

CBV

$$\sigma = [\; a \mapsto a_0,\quad b \mapsto a_1,\quad x \mapsto a_2,\quad y \mapsto a_3,\quad t \mapsto a_4, \;]$$

	a_0	a_1	a_2	a_3	a_4
$M =$	1	2	2	1	1

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

CBR

$$\sigma = [\; a \mapsto a_0,\quad b \mapsto a_1,\quad x \mapsto a_0,\quad y \mapsto a_1,\quad t \mapsto a_2, \;]$$

	a_0	a_1	a_2
$M =$	2	1	1

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

The current semantics of MFAE is based on the **call-by-value (CBV)** 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)** strategy instead; if the argument expression is an identifier, the parameter points to its address.

CBV

$$\sigma = [\; a \mapsto a_0, \quad b \mapsto a_1, \;]$$

$M =$	a_0	a_1	a_2	a_3	a_4
	1	2	2	1	1

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

CBR

$$\sigma = [\; a \mapsto a_0, \quad b \mapsto a_1, \;]$$

$M =$	a_0	a_1	a_2
	2	1	1

Function Application (Call-by-Reference)

PLRG

We can design and implement the semantics of MFAE with the **call-by-reference** strategy by adding the following cases:

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

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

Exercise #6

- Please see this document[1] 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.

Summary

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

Next Lecture

- Garbage Collection

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