Lecture 10 – Mutable Data Structures

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 Fall

Recall

- Recursion
  - Recursion in F1VAE and FVAE
  - `mkRec` helper function
  - RFAE – FAE with recursion and conditionals

Recall

- Recursion
  - Recursion in F1VAE and FVAE
  - `mkRec` helper function
  - RFAE – FAE with recursion and conditionals

- In this lecture, we will learn **mutable data structures** (boxes)

Recall

- Recursion
  - Recursion in F1VAE and FVAE
  - `mkRec` helper function
  - RFAE – FAE with recursion and conditionals

- In this lecture, we will learn **mutable data structures** (boxes)

- **BFAE – FAE with mutable boxes**
  - Concrete and Abstract Syntax
  - Interpreter and Natural Semantics

Contents

Contents

Mutable Data Structures

So far, our languages are **purely functional**:

- All functions are **pure** (no side effects)
- All data structures and variables are **immutable** (no mutation)

Mutable Data Structures

So far, our languages are **purely functional**:

- All functions are **pure** (no side effects)
- All data structures and variables are **immutable** (no mutation)

However, **mutation** is widely used in practice, especially in **imperative languages** (e.g., C, C++, Java, Python, etc.).

Mutable Data Structures

So far, our languages are **purely functional**:

- All functions are **pure** (no side effects)
- All data structures and variables are **immutable** (no mutation)

However, **mutation** is widely used in practice, especially in **imperative languages** (e.g., C, C++, Java, Python, etc.).

Mutation makes it possible to **change the state** of a program by **updating the contents** of a data structure or a variable after its creation.

- **Mutable data structures** (e.g., `mutable.Map` in Scala)
- **Mutable variables** (e.g., `var` in Scala)

Mutable Data Structures

So far, our languages are **purely functional**:

- All functions are **pure** (no side effects)
- All data structures and variables are **immutable** (no mutation)

However, **mutation** is widely used in practice, especially in **imperative languages** (e.g., C, C++, Java, Python, etc.).

Mutation makes it possible to **change the state** of a program by **updating the contents** of a data structure or a variable after its creation.

- **Mutable data structures** (e.g., `mutable.Map` in Scala)
- **Mutable variables** (e.g., `var` in Scala)

While mutation helps us write more **efficient** programs, it also makes programs **harder to reason** about and **error-prone**.

Mutable Data Structures

So far, our languages are **purely functional**:

- All functions are **pure** (no side effects)
- All data structures and variables are **immutable** (no mutation)

However, **mutation** is widely used in practice, especially in **imperative languages** (e.g., C, C++, Java, Python, etc.).

Mutation makes it possible to **change the state** of a program by **updating the contents** of a data structure or a variable after its creation.

- **Mutable data structures** (e.g., `mutable.Map` in Scala)
- **Mutable variables** (e.g., `var` in Scala)

While mutation helps us write more **efficient** programs, it also makes programs **harder to reason** about and **error-prone**.

In this lecture, we will learn **mutable data structures**.

Mutable Data Structures

A **mutable data structure** is a data structure whose **contents** can be **modified** after its creation.

Mutable Data Structures

A **mutable data structure** is a data structure whose **contents** can be **modified** after its creation. Let's define them in Scala:

```
// immutable map
val imap = Map("x" -> 1, "y" -> 2)
imap + ("x" -> 3)    // Map(x -> 3, y -> 2)
imap                 // Map(x -> 1, y -> 2)
// mutable map
val mmap = scala.collection.mutable.Map("x" -> 1, "y" -> 2)
mmap.update("x", 3)
mmap                 // mutable.Map(x -> 3, y -> 2)
```

Mutable Data Structures

PLRG

A **mutable data structure** is a data structure whose **contents** can be **modified** after its creation. Let's define them in Scala:

```
// immutable map
val imap = Map("x" -> 1, "y" -> 2)
imap + ("x" -> 3)    // Map(x -> 3, y -> 2)
imap                 // Map(x -> 1, y -> 2)
// mutable map
val mmap = scala.collection.mutable.Map("x" -> 1, "y" -> 2)
mmap.update("x", 3)
mmap                 // mutable.Map(x -> 3, y -> 2)
```

We can define our own **mutable data structure** – a **Box**:

```
// mutable box
case class Box(var content: Int)
val box: Box = Box(5)
box.content            // 5
box.content = 8
box.content            // 8
```

Contents

BFAE – FAE with Mutable Boxes

Now, let's extend FAE into BFAE to support **mutable boxes**.

```
/* BFAE */
val box = Box(5);
box.get;          // 5
box.set(8);
box.get           // 8
```

```
/* BFAE */
val box = Box(1);
val f   = x => x + box.get;
f(3);             // 3 + 1 = 4
box.set(2);
f(3);             // 3 + 2 = 5
```

(We support variable definitions (`val`) as syntactic sugar.)

BFAE – FAE with Mutable Boxes

PLRG

Now, let's extend FAE into BFAE to support **mutable boxes**.

```
/* BFAE */
val box = Box(5);
box.get;          // 5
box.set(8);
box.get           // 8
```

```
/* BFAE */
val box = Box(1);
val f   = x => x + box.get;
f(3);             // 3 + 1 = 4
box.set(2);
f(3);             // 3 + 2 = 5
```

(We support variable definitions (`val`) as syntactic sugar.)

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

1. **box creation** (`Box`)
2. **box operations**: content getter (`get`) and setter (`set`)
3. **sequence** of expressions

Concrete Syntax

```
// expressions
<expr> ::= ...
         | "Box" "(" <expr> ")"
         | <expr> "." "get"
         | <expr> "." "set" "(" <expr> ")"
         | <expr> ";" <expr>
```

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

1. **box creation**
2. **box operations**: `get` and `set`
3. **sequence** of expressions

Abstract Syntax

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

$$
\begin{array}{llll}
\text{Expressions} & \mathbb{E} \ni e ::= \ldots & & \\
& & \mid \texttt{Box}(e) & (\texttt{NewBox}) \\
& & \mid e\texttt{.get} & (\texttt{GetBox}) \\
& & \mid e\texttt{.set}(e) & (\texttt{SetBox}) \\
& & \mid e;\ e & (\texttt{Seq})
\end{array}
$$

```
enum Expr:
  ...
  // box creation
  case NewBox(expr: Expr)
  // box content getter
  case GetBox(box: Expr)
  // box content setter
  case SetBox(box: Expr, expr: Expr)
  // sequence
  case Seq(left: Expr, right: Expr)
```

Contents

Evaluation with Memories

How to evaluate the following BFAE expression?

```
/* BFAE */
val box = Box(5);
box.get;    // 5
box.set(8);
box.get     // 8
```

Evaluation with Memories

How to evaluate the following BFAE expression?

```
/* BFAE */
val box = Box(5);
box.get;    // 5
box.set(8);
box.get     // 8
```

Let's evaluate it with a **memory** M, which is a finite **mapping** from **addresses** to their **values**.

$$M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$$

A **box** allocates a **memory cell** to store a **value** in the **memory**.

- **box creation** allocates a memory cell and stores the value
- **box content getter** reads the value from the memory cell
- **box content setter** writes the value to the memory cell

Evaluation with Memories

How to evaluate the following BFAE expression?

```
/* BFAE */                *
val box = Box(5);
box.get;
box.set(8);
box.get
```

$\sigma = [$

$]$

$\mathbb{A}$	:	a_0	a_1	a_2	$\dots$
M	=				$\dots$

Let's evaluate it with a **memory** M, which is a **mapping** from **addresses** to **values**.

$$M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$$

A **box** allocates a **memory cell** to store a **value** in the **memory**.

- **box creation** allocates a memory cell and stores the value
- **box content getter** reads the value from the memory cell
- **box content setter** writes the value to the memory cell

Evaluation with Memories

PLRG

How to evaluate the following BFAE expression?

```
/* BFAE */
val box = Box(5);   *
box.get;
box.set(8);
box.get
```

$\sigma = [$

$\quad \texttt{box} \mapsto a_0$

$]$

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

Let's evaluate it with a **memory** M, which is a **mapping** from **addresses** to **values**.

$$M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$$

A **box** allocates a **memory cell** to store a **value** in the **memory**.

- **box creation** allocates a memory cell and stores the value
- **box content getter** reads the value from the memory cell
- **box content setter** writes the value to the memory cell

Evaluation with Memories

PLRG

How to evaluate the following BFAE expression?

```
/* BFAE */
val box = Box(5);
box.get;   /* 5 */   *
box.set(8);
box.get
```

$\sigma = [$
  $\texttt{box} \mapsto a_0$
$]$

$\mathbb{A}$	:	a_0	a_1	a_2	$\dots$
M	=	5			$\dots$

Let's evaluate it with a **memory** M, which is a **mapping** from **addresses** to **values**.

$$M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$$

A **box** allocates a **memory cell** to store a **value** in the **memory**.

- **box creation** allocates a memory cell and stores the value
- **box content getter** reads the value from the memory cell
- **box content setter** writes the value to the memory cell

Evaluation with Memories

PLRG

How to evaluate the following BFAE expression?

```
/* BFAE */
val box = Box(5);
box.get;   /* 5 */
box.set(8);          *
box.get
```

$$\sigma = [\ \texttt{box} \mapsto a_0\ ]$$

$\mathbb{A}$	:	a_0	a_1	a_2	...
M	=	8			...

Let's evaluate it with a **memory** M, which is a **mapping** from **addresses** to **values**.

$$M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$$

A **box** allocates a **memory cell** to store a **value** in the **memory**.

- **box creation** allocates a memory cell and stores the value
- **box content getter** reads the value from the memory cell
- **box content setter** writes the value to the memory cell

Evaluation with Memories

How to evaluate the following BFAE expression?

```
/* BFAE */
val box = Box(5);
box.get;   /* 5 */
box.set(8);
box.get    /* 8 */  *
```

$\sigma = [$
  $\texttt{box} \mapsto a_0$
$]$

$\mathbb{A}$	:	a_0	a_1	a_2	$\ldots$
M	=	8			$\ldots$

Let's evaluate it with a **memory** M, which is a **mapping** from **addresses** to **values**.

$$M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$$

A **box** allocates a **memory cell** to store a **value** in the **memory**.

- **box creation** allocates a memory cell and stores the value
- **box content getter** reads the value from the memory cell
- **box content setter** writes the value to the memory cell

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */                    *
val a = Box(1);
val f = x => x + a.get;
f(5);

a.set(2);
f(5);

val b = Box(a);
b.get.set(3);
f(5);
```

$\sigma = [$

$]$

$\mathbb{A}$	:	a_0	a_1	a_2	...
M	=				...

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);        *
val f = x => x + a.get;
f(5);

a.set(2);
f(5);

val b = Box(a);
b.get.set(3);
f(5);
```

$$\sigma = [\ \texttt{a} \mapsto a_0\ ]$$

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

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);
val f = x => x + a.get;   *
f(5);

a.set(2);
f(5);

val b = Box(a);
b.get.set(3);
f(5);
```

$\sigma = [$
$\quad \texttt{a} \mapsto a_0$
$\quad \texttt{f} \mapsto \langle \lambda \texttt{x}.(\texttt{x + a.get}), [\texttt{a} \mapsto a_0] \rangle$
$]$

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

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);
val f = x => x + a.get;
f(5);   /* 5 + 1 = 6 */  *

a.set(2);
f(5);

val b = Box(a);
b.get.set(3);
f(5);
```

$$\sigma = [$$
$$\texttt{a} \mapsto a_0$$
$$\texttt{f} \mapsto \langle \lambda \texttt{x}.(\texttt{x + a.get}), [\texttt{a} \mapsto a_0] \rangle$$
$$]$$

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

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);
val f = x => x + a.get;
f(5);    /* 5 + 1 = 6 */

a.set(2);                    *
f(5);

val b = Box(a);
b.get.set(3);
f(5);
```

$$\sigma = [\ \mathtt{a} \mapsto a_0,\ \mathtt{f} \mapsto \langle \lambda \mathtt{x}.(\mathtt{x} + \mathtt{a.get}), [\mathtt{a} \mapsto a_0] \rangle\ ]$$

$\mathbb{A}$:	a_0	a_1	a_2	...
M =	2			...

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);
val f = x => x + a.get;
f(5);    /* 5 + 1 = 6 */

a.set(2);
f(5);    /* 5 + 2 = 7 */  *

val b = Box(a);
b.get.set(3);
f(5);
```

$$\sigma = [\; \texttt{a} \mapsto a_0 \quad \texttt{f} \mapsto \langle \lambda \texttt{x}.(\texttt{x + a.get}), [\texttt{a} \mapsto a_0] \rangle \;]$$

$\mathbb{A}$	:	a_0	a_1	a_2	...
M	=	2			...

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);
val f = x => x + a.get;
f(5);    /* 5 + 1 = 6 */

a.set(2);
f(5);    /* 5 + 2 = 7 */

val b = Box(a);            *
b.get.set(3);
f(5);
```

$$\sigma = [$$
$$\texttt{a} \mapsto a_0$$
$$\texttt{f} \mapsto \langle \lambda \texttt{x}.(\texttt{x + a.get}), [\texttt{a} \mapsto a_0] \rangle$$
$$\texttt{b} \mapsto a_1$$
$$]$$

$\mathbb{A}$	:	a_0	a_1	a_2	...
M	=	2	a_0		...

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);
val f = x => x + a.get;
f(5);   /* 5 + 1 = 6 */

a.set(2);
f(5);   /* 5 + 2 = 7 */

val b = Box(a);
b.get.set(3);          *
f(5);
```

$$\sigma = [\ \mathtt{a} \mapsto a_0,\ \ \mathtt{f} \mapsto \langle \lambda \mathtt{x}.(\mathtt{x}\ \mathtt{+}\ \mathtt{a.get}), [\mathtt{a} \mapsto a_0] \rangle,\ \ \mathtt{b} \mapsto a_1\ ]$$

$\mathbb{A}$	:	a_0	a_1	a_2	...
M	=	3	a_0		...

Evaluation with Memories

Here is another BFAE expression:

```
/* BFAE */
val a = Box(1);
val f = x => x + a.get;
f(5);    /* 5 + 1 = 6 */

a.set(2);
f(5);    /* 5 + 2 = 7 */

val b = Box(a);
b.get.set(3);
f(5);    /* 5 + 3 = 8 */   *
```

$$\sigma = [\ \mathtt{a} \mapsto a_0,\ \mathtt{f} \mapsto \langle \lambda \mathtt{x}.(\mathtt{x} + \mathtt{a.get}), [\mathtt{a} \mapsto a_0] \rangle,\ \mathtt{b} \mapsto a_1\ ]$$

$\mathbb{A}$	:	a_0	a_1	a_2	...
M	=	3	a_0		...

Interpreter and Natural Semantics

For BFAE, 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 Addr = Int
type Mem = Map[Addr, Value]
enum Value:
  ...
  case BoxV(addr: Addr)
```

Interpreter and Natural Semantics

For BFAE, 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 Addr = Int
type Mem = Map[Addr, Value]
enum Value:
  ...
  case BoxV(addr: Addr)
```

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

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

Addresses	a	$\in \mathbb{A}$	(`Addr`)
Memories	M	$\in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$	(`Mem`)
Values	$\mathbb{V} \ni v$	$::= \ldots \mid a$	(`BoxV`)

Addition

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Add(l, r) =>
    val (lv, lmem) = interp(l, env, mem)
    val (rv, rmem) = interp(r, env, lmem)
    (numAdd(lv, rv), rmem)
```

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

$$\texttt{Add}\ \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1\ \texttt{+}\ e_2 \Rightarrow n_1 + n_2, M_2}$$

Addition

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Add(l, r) =>
    val (lv, lmem) = interp(l, env, mem)
    val (rv, rmem) = interp(r, env, lmem)
    (numAdd(lv, rv), rmem)
```

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

$$\texttt{Add}\ \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2, M_2}$$

```
/* BFAE */
val x = Box(5);
{ x.set(8); 2 } + x.get; // 2 + 8 = 10 -- NOT 2 + 5 = 7
```

Box Creation

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case NewBox(c) =>
    val (cv, cmem) = interp(c, env, mem)
    val addr = malloc(cmem)
    (BoxV(addr), cmem + (addr -> cv))
```

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

$$\texttt{NewBox}\ \frac{\sigma, M \vdash e \Rightarrow v, M_1 \qquad a \notin \mathsf{Domain}(M_1)}{\sigma, M \vdash \mathsf{Box}(e) \Rightarrow a, M_1[a \mapsto v]}$$

Box Creation

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case NewBox(c) =>
    val (cv, cmem) = interp(c, env, mem)
    val addr = malloc(cmem)
    (BoxV(addr), cmem + (addr -> cv))
```

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

$$\texttt{NewBox}\ \frac{\sigma, M \vdash e \Rightarrow v, M_1 \qquad a \notin \mathsf{Domain}(M_1)}{\sigma, M \vdash \mathsf{Box}(e) \Rightarrow a, M_1[a \mapsto v]}$$

One way to implement `malloc` is to find the maximum address in the memory and increment it by one, 0 if the memory is empty:

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

Box Content Getter

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case GetBox(b) =>
    val (bv, bmem) = interp(b, env, mem)
    bv match
      case BoxV(addr) => (bmem(addr), bmem)
      case _          => error(s"not a box: ${bv.str}")
```

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

$$\texttt{GetBox}\ \frac{\sigma, M \vdash e \Rightarrow a, M_1}{\sigma, M \vdash e\,\texttt{.get} \Rightarrow M_1(a), M_1}$$

Box Content Setter

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case SetBox(b, c) =>
    val (bv, bmem) = interp(b, env, mem)
    bv match
      case BoxV(addr) =>
        val (cv, cmem) = interp(c, env, bmem)
        (cv, cmem + (addr -> cv))
      case _ =>
        error(s"not a box: ${bv.str}")
```

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

$$\texttt{SetBox}\ \frac{\sigma, M \vdash e_1 \Rightarrow a, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow v, M_2}{\sigma, M \vdash e_1\texttt{.set}(e_2) \Rightarrow v, M_2[a \mapsto v]}$$

Sequence

```
def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Seq(l, r) =>
    val (_, lmem) = interp(l, env, mem)
    interp(r, env, lmem)
```

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

$$\texttt{Seq}\ \frac{\sigma, M \vdash e_1 \Rightarrow \_, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1;\ e_2 \Rightarrow v_2, M_2}$$

Summary

1. Mutable Data Structures

2. BFAE – FAE with Mutable Boxes
   - Concrete Syntax
   - Abstract Syntax

3. Interpreter and Natural Semantics for BFAE
   - Evaluation with Memories
   - Interpreter and Natural Semantics
   - Addition
   - Box Creation
   - Box Content Getter
   - Box Content Setter
   - Sequence

Exercise #6

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

- Please see above document on GitHub:
  - Implement `interp` function.

- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

Next Lecture

- Mutable Variables

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