

Lecture 10 – Mutable Data Structures

COSE212: Programming Languages

Jihyeok Park



2023 Fall

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

- 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)

- 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

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

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

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

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

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.).

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 update 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)

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 update 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**.

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 update 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**.

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

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
import scala.collection.*
val mmap = mutable.Map("x" -> 1, "y" -> 2)
mmap.update("x", 3)
mmap                // mutable.Map(x -> 3, y -> 2)

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

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

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.)

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**
- 2 **box operations**: content getter and setter
- 3 **sequence** of expressions


```
// 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

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

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

	Box(<i>e</i>)	(NewBox)
	<i>e</i> .get	(GetBox)
	<i>e</i> .set(<i>e</i>)	(SetBox)
	<i>e</i> ; <i>e</i>	(Seq)

```
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)
```

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

How to evaluate the following BFAE expression?

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

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 **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

How to evaluate the following BFAE expression?

```
/* BFAE */
```

```
val box = Box(5);
```

```
box.get;
```

```
box.set(8);
```

```
box.get
```

*

$$\sigma = [$$

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

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

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

$$M \in \mathbb{A} \rightarrow \mathbb{V}$$

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

How to evaluate the following BFAE expression?

```
/* BFAE */
```

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

$$\sigma = \left[\begin{array}{l} \text{box} \mapsto a_0 \end{array} \right]$$
$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 5 & & & \dots \\ \hline \end{array} \end{array}$$

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

$$M \in \mathbb{A} \rightarrow \mathbb{V}$$

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

How to evaluate the following BFAE expression?

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

$$\sigma = [\quad]$$

$$box \mapsto a_0$$

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

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

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

$$M \in \mathbb{A} \rightarrow \mathbb{V}$$

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

How to evaluate the following BFAE expression?

```
/* BFAE */
```

```
val box = Box(5);
```

```
box.get; /* 5 */
```

```
box.set(8);
```

```
*
```

```
box.get
```

$$\sigma = \left[\begin{array}{l} \text{box} \mapsto a_0 \end{array} \right]$$

$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 8 & & & \dots \\ \hline \end{array} \end{array}$$

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

$$M \in \mathbb{A} \rightarrow \mathbb{V}$$

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

How to evaluate the following BFAE expression?

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

$$\sigma = \left[\begin{array}{l} \text{box} \mapsto a_0 \end{array} \right]$$
$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 8 & & & \dots \\ \hline \end{array} \end{array}$$

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

$$M \in \mathbb{A} \rightarrow \mathbb{V}$$

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

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);
```

*

$\sigma = [$

 $]$

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

			...
--	--	--	-----

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);
```

*

$$\sigma = [$$
$$a \mapsto a_0$$
$$]$$
$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 1 & & & \dots \\ \hline \end{array} \end{array}$$

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);
```

$$\sigma = [$$
$$a \mapsto a_0$$
$$f \mapsto \langle \lambda x. (x + a.get), [a \mapsto a_0] \rangle$$
$$]$$
$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 1 & & & \dots \\ \hline \end{array} \end{array}$$

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);
```

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

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

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);
```

*

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

$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 2 & & & \dots \\ \hline \end{array} \end{array}$$

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);
```

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

$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 2 & & & \dots \\ \hline \end{array} \end{array}$$

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);
```

*

$$\sigma = [$$
$$a \mapsto a_0$$
$$f \mapsto \langle \lambda x. (x + a.get), [a \mapsto a_0] \rangle$$
$$b \mapsto a_1$$
$$]$$
$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 2 & a_0 & & \dots \\ \hline \end{array} \end{array}$$

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);
```

*

$$\sigma = [$$
$$a \mapsto a_0$$
$$f \mapsto \langle \lambda x. (x + a.get), [a \mapsto a_0] \rangle$$
$$b \mapsto a_1$$
$$]$$
$$\begin{array}{lcl} \mathbb{A} & : & a_0 \quad a_1 \quad a_2 \quad \dots \\ M & = & \begin{array}{|c|c|c|c|} \hline 3 & a_0 & & \dots \\ \hline \end{array} \end{array}$$

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 */ *
```

$$\sigma = [$$
$$a \mapsto a_0$$
$$f \mapsto \langle \lambda x. (x + a.get), [a \mapsto a_0] \rangle$$
$$b \mapsto a_1$$
$$]$$
$$\mathbb{A} : a_0 \quad a_1 \quad a_2 \quad \dots$$
$$M = \begin{array}{|c|c|c|c|} \hline 3 & a_0 & & \dots \\ \hline \end{array}$$

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)
```

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 $\mathbb{V} \ni v ::= \dots \mid a$ (BoxV)

```
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)
```

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

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

```
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)
```

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

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

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

```
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))
```

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

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


```
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))
```

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

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

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)
```

```
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}")
```

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

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

```
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}")
```

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

$$\text{GetBox} \frac{\sigma, M \vdash e_1 \Rightarrow a, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v, M_2}{\sigma, M \vdash e_1.\text{set}(e_2) \Rightarrow v, M_2[a \mapsto v]}$$

```
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)
```

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

$$\text{GetBox} \frac{\sigma, M \vdash e_1 \Rightarrow -, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1; e_2 \Rightarrow v_2, M_2}$$

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

- Please see this document¹ on GitHub.
- The due date is Oct. 27 (Fri.).
- Please only submit `Implementation.scala` file to **Blackboard**.

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

- Mutable Variables

Jihyeok Park

jihyeok_park@korea.ac.kr

<https://plrg.korea.ac.kr>