Lecture 10 – Mutable Data Structures COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

PLRG

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)

PLRG

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



- 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

Contents



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 Semantic 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" \rightarrow 1, "y" \rightarrow 2)
imap + ("x" -> 3) // Map(x -> 3, y -> 2)
                   // Map(x -> 1, y -> 2)
imap
// mutable map
import scala.collection.*
val mmap = mutable.Map("x" \rightarrow 1, "y" \rightarrow 2)
mmap.update("x", 3)
                      // mutable.Map(x -> 3, y -> 2)
mmap
// mutable box
case class Box(var content: Int)
val box = Box(5)
                       // 5
box.content
box.content = 8
                        // 8
box.content
```

Contents



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

BFAE - FAE with Mutable Boxes



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

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

BFAE – FAE with Mutable Boxes



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

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

For BFAE, we need to extend expressions of FAE with

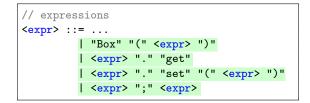
1 box creation

2 box operations: content getter and setter

3 sequence of expressions

Concrete Syntax





For BFAE, we need to extend expressions of FAE with

1 box creation

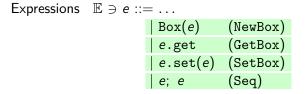
2 box operations: get and set

3 sequence of expressions

Abstract Syntax



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



enum	Expr:
• •	
//	box creation
ca	se NewBox(expr: Expr)
//	box content getter
ca	se GetBox(box: Expr)
//	box content setter
ca	<pre>se SetBox(box: Expr, expr: Expr)</pre>
//	sequence
ca	se Seq(left: Expr, right: Expr)

Contents



- 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?

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

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

- 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);*
$$\sigma = [$$
A : $a_0 \ a_1 \ a_2 \ \dots$ box.get;
box.set(8);
box.get...

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

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

- 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?



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

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

- 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?

$$\begin{array}{|c|c|c|c|c|c|c|c|c|} & /* & \text{BFAE } */ \\ \hline \text{val box = Box(5);} & \sigma = [& \mathbb{A} & : & a_0 & a_1 & a_2 & \dots \\ \hline \text{box.get;} & /* & 5 & */ & * \\ \hline \text{box.set(8);} & & \\ \hline \text{box.get} & & \end{bmatrix}$$

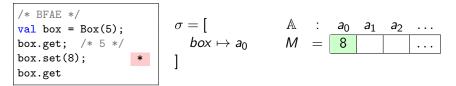
Let's evaluate it with a **memory** M, which is a **mapping** from **addresses** to **values**.

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

- 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?



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

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

- 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



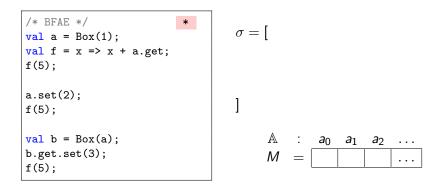
How to evaluate the following BFAE expression?

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

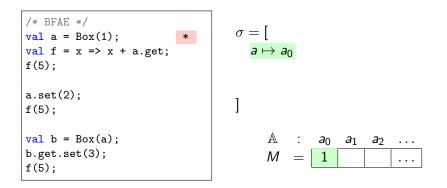
$$M \in \mathbb{A} \to \mathbb{V}$$

- 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

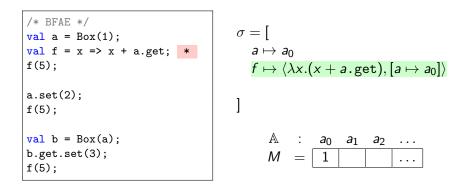




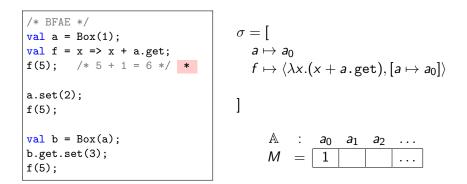




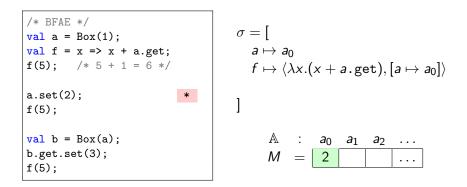








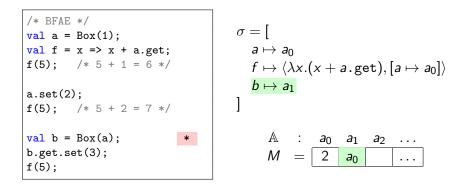




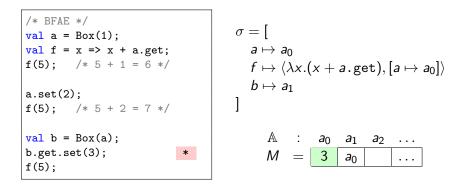


```
/* BFAE */
                                           \sigma = [
val a = Box(1);
val f = x \Rightarrow x + a.get;
                                              a \mapsto a_0
f(5): /* 5 + 1 = 6 */
                                              f \mapsto \langle \lambda x. (x + a.get), [a \mapsto a_0] \rangle
a.set(2);
f(5); /* 5 + 2 = 7 */ *
val b = Box(a);
                                                  A :
                                                            a_0
                                                                 a<sub>1</sub>
                                                                       a_2
                                                                             . . .
b.get.set(3);
                                                 Μ
                                                            2
                                                       =
                                                                             . . .
f(5);
```











$$\begin{array}{c} /* \ \text{BFAE } */ \\ \text{val } a = Box(1); \\ \text{val } f = x \Rightarrow x + a.get; \\ f(5); \ /* \ 5 + 1 = 6 \ */ \\ a.set(2); \\ f(5); \ /* \ 5 + 2 = 7 \ */ \\ \end{array} \\ \begin{array}{c} \sigma = [\\ a \mapsto a_0 \\ f \mapsto \langle \lambda x.(x + a.get), [a \mapsto a_0] \rangle \\ b \mapsto a_1 \\ \end{bmatrix} \\ b \mapsto a_1 \\ \end{array} \\ \begin{array}{c} M = \boxed{3 \ a_0 \ \dots} \\ \end{array}$$

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:

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

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

Addition



def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
...
case Add(1, r) =>
val (1v, lmem) = interp(1, env, mem)
val (rv, rmem) = interp(r, env, lmem)
(numAdd(1v, rv), rmem)

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

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

Addition



def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
...
case Add(1, r) =>
val (1v, lmem) = interp(1, env, mem)
val (rv, rmem) = interp(r, env, lmem)
(numAdd(1v, rv), rmem)

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

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

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

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))
```

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

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))
```

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

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}")
```

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

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

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}")
```

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

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

Sequence



def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
 ...
 case Seq(1, r) =>
 val (_, lmem) = interp(1, env, mem)
 interp(r, env, lmem)

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

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

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

Homework #2



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

¹https://github.com/ku-plrg-classroom/docs/tree/main/cose212/cobalt. COSE212 @ Korea University Lecture 10 - Mutable Data Structures October 9, 2023 34/35

Next Lecture



Mutable Variables

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