

Lecture 7 – First-Class Functions

COSE212: Programming Languages

Jihyeok Park



2024 Fall

- **F1VAE – VAE with first-order functions**
 - Concrete and Abstract Syntax
 - Evaluation with Function Environments
 - Interpreters and Natural Semantics
 - Static Scoping vs Dynamic Scoping
- In this lecture, we will learn **first-class functions**.
- **FVAE – VAE with first-class functions**
 - Concrete and Abstract Syntax
 - Interpreter and Natural Semantics

1. First-Class Functions

2. FVAE – VAE with First-Class Functions

- Concrete Syntax

- Abstract Syntax

3. Interpreter and Natural Semantics for FVAE

- Closures – Functions as Values

- Addition and Multiplication

- Anonymous Functions

- Function Application

- Function Application (Dynamic Scoping)

1. First-Class Functions

2. FVAE – VAE with First-Class Functions

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for FVAE

Closures – Functions as Values

Addition and Multiplication

Anonymous Functions

Function Application

Function Application (Dynamic Scoping)

In a programming language, an entity is said to be **first-class citizen** if it is treated as a **value**. In other words, it can be

- ① **assigned** to a **variable**,
- ② **passed** as an **argument** to a function, and
- ③ **returned** from a function.

For example, an integer is obviously a first-class citizen in Scala:

```
// 1. We can assign an integer to a variable.
val n: Int = 3

// 2. We can pass an integer as an argument to a function.
def square(n: Int): Int = n * n
square(3)           // 3 * 3 = 9

// 3. We can return an integer from a function.
def square(n: Int): Int = n * n
square(3)           // 3 * 3 = 9
```

In Scala, **functions** are also **first-class citizens**, and we call them **first-class functions**.

```
def inc(n: Int): Int = n + 1

// 1. We can assign a function to a variable.
val f: Int => Int = inc
val g: Int => Int = x => x + 1           // anonymous (lambda) function

// 2. We can pass a function as an argument to a function.
def twice(f: Int => Int, n: Int): Int = f(f(n))
twice(inc, 3)                         // inc(inc(3)) = 3 + 1 + 1 = 5
List(1, 2, 3).map(inc)                 // List(2, 3, 4)

// 3. We can return a function from a function.
def addN(n: Int): Int => Int = m => n + m
def addN(n: Int)(m: Int): Int = n + m // currying
addN(3)(5)                            // 3 + 5 = 8
val add3: Int => Int = addN(3)
add3(5)                               // 3 + 5 = 8
```

Programming languages supporting **functional programming** paradigm treat functions as first-class citizens (i.e., **first-class functions**).

- Scala

```
List(1, 2, 3).map(x => x * 2)           // List(2, 4, 6)
```

- Python

```
list(map(lambda x: x * 2, [1, 2, 3]))  # [2, 4, 6]
```

- Rust

```
[1, 2, 3].iter().map(|x| x * 2).collect() // [2, 4, 6]
```

- Haskell

```
map (\x -> x * 2) [1, 2, 3]           -- [2, 4, 6]
```

- ...

1. First-Class Functions

2. FVAE – VAE with First-Class Functions

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for FVAE

Closures – Functions as Values

Addition and Multiplication

Anonymous Functions

Function Application

Function Application (Dynamic Scoping)

Now, we want to extend VAE into FVAE with **first-class functions** rather than **first-order functions** in F1VAE.

```
/* FVAE */  
val addN = n => m => n + m;  
val add3 = addN(3);  
add3(5) // 3 + 5 = 8
```

```
/* FVAE */  
val inc = x => x + 1;  
val twice = f => n => f(f(n));  
twice(inc)(5) // 5 + 1 + 1 = 7
```

For FVAE, we need to extend **expressions** of VAE with

- 1 anonymous (lambda) functions
- 2 function applications

For FVAE, we need to extend **expressions** of VAE with

- ① **anonymous (lambda) functions**
- ② **function applications**

Let's define the **concrete syntax** of FVAE in BNF:

```
// expressions
<expr> ::= ...
        | <id> "=>" <expr>
        | <expr> "(" <expr> ")"
```

Why not the following function application syntax?

```
| <id> "(" <expr> ")"
```

We cannot support curried function applications with the above syntax:

addN(3)(5)

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

Expressions $e ::= \dots$

	<code>val x = e; e</code>	(Val)
	<code>x</code>	(Id)
	<code>λx.e</code>	(Fun)
	<code>e(e)</code>	(App)

```
enum Expr:
  ...
  case Val(name: String, init: Expr, body: Expr)
  case Id(name: String)
  // anonymous (lambda) functions
  case Fun(param: String, body: Expr)
  // function applications
  case App(fun: Expr, arg: Expr)
```

For example, let's **parse** the following FVAE program:

```
/* FVAE */  
val addN = n => m => n + m;  
val add3 = addN(3);  
add3(5)
```

Then, the following **abstract syntax tree (AST)** is produced:

```
Val("addN",  
  Fun("n",  
    Fun("m",  
      Add(Id("n"), Id("m"))  
    )  
  ),  
  Val("add3",  
    App(Id("addN"), Const(3)),  
    App(Id("add3"), Const(5))  
  )  
)
```

1. First-Class Functions

2. FVAE – VAE with First-Class Functions

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for FVAE

Closures – Functions as Values

Addition and Multiplication

Anonymous Functions

Function Application

Function Application (Dynamic Scoping)

Let's evaluate the following FVAE program:

```
/* FVAE */  
val addN = n => m => n + m;  
val add3 = addN(3);  
add3(5) // 3 + 5 = 8
```

How to evaluate the function applications `addN(3)` and `add3(5)`?

$$\begin{aligned} [\text{addN} \mapsto v_0] &\vdash \text{addN}(3) \Rightarrow v_1 \\ [\text{addN} \mapsto v_0, \text{add3} \mapsto v_1] &\vdash \text{add3}(5) \Rightarrow v_2 \end{aligned}$$

What's values of `addN` and `add3` inside the environments?

Functions! Let's define **values** as either **numbers** or **functions**:

$$\text{Values } \mathbb{V} \ni v ::= n \mid \lambda x.e$$

However, it is **NOT** what exactly we want to do. Why?

```
/* FVAE */
val addN = n => m => n + m;
val add3 = addN(3);
add3(5) // 3 + 5 = 8
```

$$\begin{aligned} [\text{addN} \mapsto v_0] & \vdash \text{addN}(3) \Rightarrow \lambda m. (n + m) \\ [\text{addN} \mapsto v_0, \text{add3} \mapsto v_1] & \vdash \text{add3}(5) \Rightarrow v_2 \end{aligned}$$

where $v_0 = \lambda n. \lambda m. (n + m)$ and $v_1 = \lambda m. (n + m)$.

We know that m represents 5, but **what about** n ?

Let's define **closures** as pairs of **functions** and its **environments**:

Values $\mathbb{V} \ni v ::= n \mid \langle \lambda x. e, \sigma \rangle$

$$\begin{aligned} [\text{addN} \mapsto v_0] & \vdash \text{addN}(3) \Rightarrow \lambda m. (n + m) \\ [\text{addN} \mapsto v_0, \text{add3} \mapsto v_1] & \vdash \text{add3}(5) \Rightarrow 8 \end{aligned}$$

where $v_0 = \langle \lambda n. \lambda m. (n + m), \emptyset \rangle$ and $v_1 = \langle \lambda m. (n + m), [n \mapsto 3] \rangle$.

For VAE or F1VAE, a **value** is a **number** n .

```
type Value = BigInt
```

For FVAE, a **value** is either 1) a **number** n or 2) a **closure** $\langle \lambda x.e, \sigma \rangle$,

```
enum Value:  
  case NumV(n: BigInt)  
  case CloV(param: String, body: Expr, env: Env)
```

$$\begin{array}{ll} \text{Values } \mathbb{V} \ni v ::= n & (\text{NumV}) \\ \quad \mid \langle \lambda x.e, \sigma \rangle & (\text{CloV}) \end{array}$$

and the interpreter takes an **expression** e with an **environment** σ and returns a **value** v (either a number or a closure):

```
def interp(expr: Expr, env: Env): Value = ???
```

$$\sigma \vdash e \Rightarrow v$$

For FVAE, we need to 1) implement the **interpreter**:

```
def interp(expr: Expr, env: Env): Value = ???
```

and 2) define the **natural semantics** with environments:

$$\sigma \vdash e \Rightarrow v$$

Expressions	$e ::=$...
		$\lambda x.e$ (Fun)
		$e(e)$ (App)
Values	$\mathbb{V} \ni v ::=$	n (NumV)
		$\langle \lambda x.e, \sigma \rangle$ (CloV)

where

Environments	$\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}$	(Env)
Integers	$n \in \mathbb{Z}$	(BigInt)
Identifiers	$x \in \mathbb{X}$	(String)

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => interp(l, env) + interp(r, env)

  case Mul(l, r) => interp(l, env) * interp(r, env)
```

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

$$\text{Add} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 + e_2 \Rightarrow v_1 + v_2}$$

$$\text{Mul} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 * e_2 \Rightarrow v_1 \times v_2}$$

Is it correct?

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => interp(l, env) + interp(r, env)

  case Mul(l, r) => interp(l, env) * interp(r, env)
```

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

$$\text{Add} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 + e_2 \Rightarrow v_1 + v_2}$$

$$\text{Mul} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 * e_2 \Rightarrow v_1 \times v_2}$$

Is it correct? **No!**

We can only add or multiply **numbers** rather than arbitrary **values**.

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => (interp(l, env), interp(r, env)) match
    case (NumV(l), NumV(r)) => NumV(l + r)
    case (l, r) => error(s"invalid operation: ${l.str} + ${r.str}")
  case Mul(l, r) => (interp(l, env), interp(r, env)) match
    case (NumV(l), NumV(r)) => NumV(l * r)
    case (l, r) => error(s"invalid operation: ${l.str} * ${r.str}")
```

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

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

$$\text{Mul} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 * e_2 \Rightarrow n_1 \times n_2}$$

Let's refactor the code to avoid duplication using a helper function.

Let's define the following helper function that

- takes 1) a binary operation on **numbers** and 2) its name, and
- returns a binary operation on **values**:

```
type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] = (l, r) =>
  (l, r) match
    case (NumV(l), NumV(r)) => NumV(op(l, r))
    case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
```

Let's define the following helper function that

- takes 1) a binary operation on **numbers** and 2) its name, and
- returns a binary operation on **values**:

```
type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] = (_, _) match
  case (NumV(l), NumV(r)) => NumV(op(l, r))
  case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
```

Let's define the following helper function that

- takes 1) a binary operation on **numbers** and 2) its name, and
- returns a binary operation on **values**:

```
type BOp[T] = (T, T) => T
def numBOP(op: BOp[BigInt], x: String): BOp[Value] =
  case (NumV(l), NumV(r)) => NumV(op(l, r))
  case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")
```

Then, we can define the addition and multiplication on values as follows:

```
val numAdd: BOp[Value] = numBOP(_ + _, "+")
val numMul: BOp[Value] = numBOP(_ * _, "*")
```

Let's refactor the interpreter using the above helper functions.

```

type BOp[T] = (T, T) => T
def numBOp(op: BOp[BigInt], x: String): BOp[Value] =
  case (NumV(l), NumV(r)) => NumV(op(l, r))
  case (l, r) => error(s"invalid operation: ${l.str} $x ${r.str}")

val numAdd: BOp[Value] = numBOp(_ + _, "+")
val numMul: BOp[Value] = numBOp(_ * _, "*")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
  case Mul(l, r) => numMul(interp(l, env), interp(r, env))
  
```

$$\sigma \vdash e \Rightarrow v$$

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

$$\text{Mul} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 * e_2 \Rightarrow n_1 \times n_2}$$


```
def interp(expr: Expr, env: Env): Value = expr match
...
case Fun(p, b) => ???
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{Fun} \frac{\text{???}}{\sigma \vdash \lambda x.e \Rightarrow \text{???}}$$

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Fun(p, b) => CloV(p, b, env)
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{Fun} \frac{}{\sigma \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle}$$

Construct a **closure** $\langle \lambda x.e, \sigma \rangle$ from the function $\lambda x.e$ with the current environment σ for **static scoping**.

```
def interp(expr: Expr, env: Env): Value = expr match
...
case App(f, e) => ???
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{App} \frac{\text{???}}{\sigma \vdash e_0(e_1) \Rightarrow \text{???}}$$

```
def interp(expr: Expr, env: Env): Value = expr match
...
case App(f, e) => interp(f, env) match
  case CloV(p, b, fenv) => ...
  case v                => error(s"not a function: ${v.str}")
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{App} \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x. e_2, \sigma' \rangle \quad \dots}{\sigma \vdash e_0(e_1) \Rightarrow ???}$$

First, evaluate **function expression** e_0 , check it is a **closure** $\langle \lambda x. e_2, \sigma' \rangle$.

The **environment** σ' in the closure is the captured at the **definition site** of the function for **static scoping**.

And, we use the metavariable σ' rather than σ because they may be different in general.

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case App(f, e) => interp(f, env) match
    case CloV(p, b, fenv) => ... interp(e, env) ...
    case v                => error(s"not a function: ${v.str}")
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{App} \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x. e_2, \sigma' \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \dots}{\sigma \vdash e_0(e_1) \Rightarrow ???}$$

Then, evaluate the **argument expression** e_1 and let **value** v_1 be its result.

```
def interp(expr: Expr, env: Env): Value = expr match
...
case App(f, e) => interp(f, env) match
  case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(e, env)))
  case v                => error(s"not a function: ${v.str}")
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{App} \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x. e_2, \sigma' \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \sigma'[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

Finally, evaluate the **body expression** e_2 in the environment $\sigma'[x \mapsto v_1]$ extended from the environment σ' captured at the **definition site** of the function for **static scoping**.

```
def interp(expr: Expr, env: Env): Value = expr match
...
case App(f, e) => interp(f, env) match
  case CloV(p, b, fenv) => interp(b, env + (p -> interp(e, env)))
  case v                => error(s"not a function: ${v.str}")
```

$$\sigma \vdash e \Rightarrow v$$

$$\text{App} \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x. e_2, \sigma' \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

We can define **dynamic scoping** by using the current environment σ at the **call site** of the function instead of the environment σ' captured at the definition site of the function.

```
/* FVAE (static scoping) */
val x = 3;
val f = y => x * y;
val x = 4;
f(5) // 3 * 5 = 15
```

```
/* FVAE (dynamic scoping) */
val x = 3;
val f = y => x * y;
val x = 4;
f(5) // 4 * 5 = 20
```

$$\text{APP} \frac{\sigma_1 \vdash f \Rightarrow \langle \lambda y. (x * y), \sigma_0 \rangle \quad \sigma_1 \vdash 5 \Rightarrow 5 \quad \sigma_0[y \mapsto 5] \vdash x * y \Rightarrow 15}{\sigma_1 \vdash f(5) \Rightarrow 15}$$

$$\text{APP} \frac{\sigma_1 \vdash f \Rightarrow \langle \lambda y. (x * y), \sigma_0 \rangle \quad \sigma_1 \vdash 5 \Rightarrow 5 \quad \sigma_1[y \mapsto 5] \vdash x * y \Rightarrow 20}{\sigma_1 \vdash f(5) \Rightarrow 20}$$

where

$$\begin{aligned} \sigma_0 &= [x \mapsto 3] \\ \sigma_1 &= [x \mapsto 4, f \mapsto \langle \lambda y. (x * y), \sigma_0 \rangle] \end{aligned}$$

1. First-Class Functions

2. FVAE – VAE with First-Class Functions

- Concrete Syntax

- Abstract Syntax

3. Interpreter and Natural Semantics for FVAE

- Closures – Functions as Values

- Addition and Multiplication

- Anonymous Functions

- Function Application

- Function Application (Dynamic Scoping)

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

- Please see above document on GitHub:
 - Implement `interp` function.
 - Implement `interpDS` function.
- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

- Lambda Calculus

Jihyeok Park

jihyeok_park@korea.ac.kr

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