

Lecture 6 – First-Order Functions

COSE212: Programming Languages

Jihyeok Park



2023 Fall

- **VAE – AE with variables**
 - Evaluation with Environments
 - Interpreter and Natural Semantics

- In this lecture, we will learn **first-order functions**.

- **F1VAE – VAE with first-order functions**
 - Concrete and Abstract Syntax
 - Interpreter and Natural Semantics

1. First-Order Functions
2. F1VAE – VAE with First-Order Functions
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for F1VAE
 - Evaluation with Function Environments
 - Function Application
4. Static Scoping vs Dynamic Scoping

1. First-Order Functions

2. F1VAE – VAE with First-Order Functions

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for F1VAE

Evaluation with Function Environments

Function Application

4. Static Scoping vs Dynamic Scoping

Let's calculate the square of several numbers in Scala.

```
1 * 1      // 1
2 * 2      // 4
3 * 3      // 9
42 * 42     // 1764
2434 * 2434 // 5925796
```

With a **first-order function**, we can avoid the repetition of the code.

```
// A `square` function that takes an integer `n` and returns its square.
def square(n: Int): Int = n * n

square(1)      // 1
square(2)      // 4
square(3)      // 9
square(42)     // 1764
square(2434)   // 5924356
```

Most programming languages support **first-order functions**.

- Scala

```
def square(n: Int): Int = n * n
```

```
square(3) // 9
```

- Python

```
def square(n): return n * n
```

```
square(3) # 9
```

- C++

```
int square(int n) { return n * n; }
```

```
square(3) // 9
```

- Rust

```
fn square(n: i32) -> i32 { return n * n; }
```

```
square(3) // 9
```

- ...

1. First-Order Functions

2. F1VAE – VAE with First-Order Functions

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for F1VAE

Evaluation with Function Environments

Function Application

4. Static Scoping vs Dynamic Scoping

Now, we want to extend VAE into F1VAE with **first-order functions**.

```
/* F1VAE */  
def square(n) = n * n;  
square(3) + 2 // 11
```

```
/* F1VAE */  
def add3(n) = n + 3;  
def mul2(m) = m * 2;  
mul2(add3(4)) // 14
```

- An F1VAE **program** is a pair of
 - 1 a list of **function definitions**
 - 2 an **expression**
- We extend an F1VAE **expression** with **function calls**.

Let's define the **concrete syntax** of F1VAE in BNF:

- An F1VAE **program** is a pair of
 - 1 a list of **function definitions**
 - 2 an **expression**
- We extend an F1VAE **expression** with **function calls**.

```
// programs
<program> ::= <fdef>* <expr>

// function definitions
<fdef>      ::= "def" <id> "(" <id> ")" "=" <expr> ";"

// expressions
<expr>      ::= ...
               | "{" <expr> "}"
               | "val" <id> "=" <expr> ";" <expr>
               | <id>
               | <id> "(" <expr> ")"
```

Let's define the **abstract syntax** of F1VAE in BNF:

Programs	$\mathbb{P} \ni p$	$::= f^* e$	(Program)
Function Definitions	$\mathbb{F} \ni f$	$::= \text{def } x(x)=e$	(FunDef)
Expressions	$\mathbb{E} \ni e$	$::= \dots$	
		$\text{val } x=e; e$	(Val)
		x	(Id)
		$x(e)$	(App)

```
// programs
case class Program(fdefs: List[FunDef], expr: Expr)
// function definitions
case class FunDef(name: String, param: String, body: Expr)
enum Expr:
  ...
  case Val(name: String, init: Expr, body: Expr)
  case Id(name: String)
  // function application
  case App(fname: String, arg: Expr)
```

For example, let's **parse** the following F1VAE program:

```
/* F1VAE */  
def add3(n) = n + 3;  
def mul2(m) = m * 2;  
mul2(add3(4))
```

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

```
Program(  
  List(  
    FunDef("add3", "n", Add(Id("n"), Num(3))),  
    FunDef("mul2", "m", Mul(Id("m"), Num(2)))  
  ),  
  App("mul2", App("add3", Num(4)))  
)
```

1. First-Order Functions

2. F1VAE – VAE with First-Order Functions

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for F1VAE

Evaluation with Function Environments

Function Application

4. Static Scoping vs Dynamic Scoping

Let's evaluate the following F1VAE program:

```
/* F1VAE */  
def add3(n) = n + 3;  
def mul2(m) = m * 2;  
mul2(add3(4)) // 14
```

How to find the function definition of `add3` or `mul2`?

We need to construct a **function environment** that maps function names to function definitions from the **list of function definitions** in a program.

$$[\text{add3} \mapsto f_0, \text{mul2} \mapsto f_1]$$

where

$$\begin{aligned} f_0 &= \text{def add3}(n)=n+3 \\ f_1 &= \text{def mul2}(m)=m \times 2 \end{aligned}$$

For VAE, the interpreter takes an **expression** e with an **environment** σ and returns a number n as the result.

```
type Value = BigInt                                // values
type Env = Map[String, Value]                      // environments
def interp(expr: Expr, env: Env): Value = ... // interpreter
```

$$\sigma \vdash e \Rightarrow n$$

Now, we extend it to take a **function environment** Λ , a mapping from function names to function definitions, as well:

```
type Value = BigInt                                // values
type Env = Map[String, Value]                      // environments
type FEnv = Map[String, FunDef]                    // function environments
def interp(expr: Expr, env: Env, fenv: FEnv): Value = ... // interpreter
```

$$\sigma, \Lambda \vdash e \Rightarrow n$$

However, an F1VAE program only contains a **list of function definitions**. Then, how to construct a **function environment** from a **list of function definitions**?

```
def createFEnv(fdefs: List[FunDef]): FEnv = fdefs.foldLeft(Map.empty) {  
  case (m: FEnv, fdef: FunDef) =>  
    val fname: String = fdef.name  
    // check if the function name is already in the function environment  
    if (m.contains(fname)) error(s"duplicate function: $fname")  
    else m + (fname -> fdef)  
}
```

It will throw an error if there are **duplicate function names**:

```
createFEnv(List(  
  FunDef("add3", "n", Add(Id("n"), Num(3))),  
  FunDef("add3", "n", Add(Num(3), Id("n"))),  
)) // error: duplicate function: add3
```

For F1VAE, we need to 1) implement the **interpreter**:

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

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

$$\sigma, \Lambda \vdash e \Rightarrow n$$

Programs	$\mathbb{P} \ni p ::= f^* e$	(Program)
Function Definitions	$\mathbb{F} \ni f ::= \text{def } x(x)=e$	(FunDef)
Expressions	$\mathbb{E} \ni e ::= \dots$	
	$x(e)$	(App)

where

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


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

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\text{App} \frac{\text{???}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow \text{???}}$$

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
...
case App(f, e) =>
  val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
  ...
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\begin{array}{c} x_0 \in \text{Domain}(\Lambda) \quad \Lambda(x_0) = \text{def } x_0(x_1) = e_2 \\ \dots \\ \text{App} \frac{\quad}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow ???} \end{array}$$

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
...
case App(f, e) =>
  val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
  ... interp(e, env, fenv) ...
```

$$\sigma, \Lambda \vdash e \Rightarrow n$$

$$\text{App} \frac{\begin{array}{c} x_0 \in \text{Domain}(\Lambda) \quad \Lambda(x_0) = \text{def } x_0(x_1)=e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \quad \dots \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow ???}$$

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
...
case App(f, e) =>
  val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
  ... Map(fdef.param -> interp(e, env, fenv)) ...
```

$$\boxed{\sigma, \Lambda \vdash e \Rightarrow n}$$

$$\text{App} \frac{\begin{array}{l} x_0 \in \text{Domain}(\Lambda) \quad \Lambda(x_0) = \text{def } x_0(x_1) = e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \quad \dots [x_1 \mapsto n_1] \dots \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
...
case App(f, e) =>
  val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
  interp(fdef.body, Map(fdef.param -> interp(e, env, fenv)), fenv)
```

$$\sigma, \Lambda \vdash e \Rightarrow n$$

$$\text{App} \frac{\begin{array}{l} x_0 \in \text{Domain}(\Lambda) \quad \Lambda(x_0) = \text{def } x_0(x_1) = e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \quad [x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2 \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

We skip the other cases because they are only augmented with passing function environments. If you are interested, please refer to this spec:

<https://github.com/ku-plrg-classroom/docs/blob/main/cose212/f1vae/f1vae-spec.pdf>

1. First-Order Functions
2. F1VAE – VAE with First-Order Functions
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for F1VAE
 - Evaluation with Function Environments
 - Function Application
4. Static Scoping vs Dynamic Scoping

The current semantics is called **static scoping (or lexical scoping)** because a binding occurrence is determined statically without considering the function application but only the function definition.

```
/* F1VAE */  
def f(x) = x + y;           // y is a free variable  
{ val y = 2; f(1) } + { val y = 4; f(3) }
```

However, we can define the semantics of F1VAE in another way by using the **dynamic scoping** instead; a binding occurrence is determined dynamically when function application is executed:

```
/* F1VAE */  
def f(x) = x + y;           // y = 2 or y = 4 depending on the call  
{ val y = 2; f(1) } + { val y = 4; f(3) } // (1 + 2) + (3 + 4) = 10
```

We can design and implement the semantics of F1VAE with the **dynamic scoping** by changing the definition of the function application:

```
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
  ...
  case App(f, e) =>
    val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
    interp(fdef.body, env + (fdef.param -> interp(e, env, fenv)), fenv)
```

$$\sigma, \Lambda \vdash e \Rightarrow n$$

$$\text{App} \frac{\begin{array}{l} x_0 \in \text{Domain}(\Lambda) \quad \Lambda(x_0) = \text{def } x_0(x_1) = e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \quad \sigma[x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2 \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

However, we will use the **static scoping** by default in this course.

Programs	$\mathbb{P} \ni p ::= f^* e$	(Program)
Function Definitions	$\mathbb{F} \ni f ::= \text{def } x(x)=e$	(FunDef)
Expressions	$\mathbb{E} \ni e ::= \dots$	
	$\quad x(e)$	(App)

```

type FEnv = Map[String, FunDef]
def interp(expr: Expr, env: Env, fenv: FEnv): Value = expr match
...
case App(f, e) =>
  val fdef = fenv.getOrElse(f, error(s"unknown function: $f"))
  interp(fdef.body, Map(fdef.param -> interp(e, env, fenv)), fenv)

```

$$\sigma, \Lambda \vdash e \Rightarrow n$$

$$\text{App} \frac{\begin{array}{l} x_0 \in \text{Domain}(\Lambda) \quad \Lambda(x_0) = \text{def } x_0(x_1)=e_2 \\ \sigma, \Lambda \vdash e_1 \Rightarrow n_1 \quad [x_1 \mapsto n_1], \Lambda \vdash e_2 \Rightarrow n_2 \end{array}}{\sigma, \Lambda \vdash x_0(e_1) \Rightarrow n_2}$$

- Please see this document¹ 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.

¹<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/f1vae>.

- First-Class Functions

Jihyeok Park

jihyeok_park@korea.ac.kr

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