Lecture 7 – First-Class Functions

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

- **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

Contents

Contents

First-Class Citizen

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
```

First-Class Functions

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

```scala
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
```

First-Class Functions

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

- Scala

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

- Python

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

- Rust

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

- Haskell

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

- ...

Contents

FVAE – VAE with First-Class Functions

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**

Concrete Syntax

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)
```

Abstract Syntax

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

$$
\begin{array}{lrcll}
\text{Expressions} & \mathbb{E} \ni e & ::= & \ldots & \\
& & | & \texttt{val}\ x\texttt{=}e\texttt{;}\ e & (\texttt{Val}) \\
& & | & x & (\texttt{Id}) \\
& & | & \lambda x.e & (\texttt{Fun}) \\
& & | & e(e) & (\texttt{App})
\end{array}
$$

```
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)
```

Abstract Syntax

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))
  )
)
```

Contents

Closures – Functions as Values

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)`?

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

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

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

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

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

Closures – Functions as Values

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

$$\begin{array}{ll} [\texttt{addN} \mapsto v_0] & \vdash \texttt{addN}(3) \Rightarrow \lambda m.(n+m) \\ [\texttt{addN} \mapsto v_0, \texttt{add3} \mapsto v_1] & \vdash \texttt{add3}(5) \Rightarrow \color{red}{v_2} \end{array}$$

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**:

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

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

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

Closures – Functions as Values

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

$$\text{Values} \quad \mathbb{V} \ni v ::= n \qquad (\texttt{NumV})$$
$$\mid \langle \lambda x.e, \sigma \rangle \quad (\texttt{CloV})$$

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

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

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

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

Interpreter and Natural Semantics for FVAE

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

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

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

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

$$
\begin{array}{llcll}
\text{Expressions} & \mathbb{E} \ni e & ::= & \ldots & \\
 & & | & \lambda x.e & (\texttt{Fun}) \\
 & & | & e(e) & (\texttt{App}) \\
\text{Values} & \mathbb{V} \ni v & ::= & n & (\texttt{NumV}) \\
 & & | & \langle \lambda x.e, \sigma \rangle & (\texttt{CloV})
\end{array}
$$

where

$$
\begin{array}{lll}
\text{Environments} & \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} & (\texttt{Env}) \\
\text{Integers} & n \in \mathbb{Z} & (\texttt{BigInt}) \\
\text{Identifiers} & x \in \mathbb{X} & (\texttt{String})
\end{array}
$$

Addition and Multiplication

```scala
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)
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 + e_2 \Rightarrow v_1 + v_2} \qquad \texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \times e_2 \Rightarrow v_1 \times v_2}$$

Is it correct?

Addition and Multiplication

```
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)
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 + e_2 \Rightarrow v_1 + v_2} \qquad \texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_1 \times e_2 \Rightarrow v_1 \times v_2}$$

Is it correct? No!

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

Addition and Multiplication

```
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}")
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2}$$

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

Addition and Multiplication

```
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}")
```

Let's define the following helper function:

```
type BOp = (BigInt, BigInt) => BigInt
// Type: (String, BOp) => ((Value, Value) => Value)
def numBOp(x: String, op: BOp): (Value, Value) => Value = ???
```

Let's curry the helper function:

```
// Type: String => BOp => (Value, Value) => Value
def numBOp(x: String)(op: BOp)(l: Value, r: Value): Value = ???
```

Addition and Multiplication

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

val numAdd: (Value, Value) => Value = numBOp("+")(_ + _)
val numMul: (Value, Value) => 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))
```

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

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2}$$

Anonymous Functions

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

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

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

Anonymous Functions

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

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

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

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

Function Application

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

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

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

Function Application

```scala
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}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \ldots}{\sigma \vdash e_0(e_1) \Rightarrow ???}$$

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

Function Application

```
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}")
```

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

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \ldots}{\sigma \vdash e_0(e_1) \Rightarrow ???}$$

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

Function Application

```scala
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}")
```

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

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

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

Function Application (Dynamic Scoping)

```scala
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}")
```

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

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

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.

Function Application (Dynamic Scoping)

```
/* 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
```

$$\textsc{App}\ \frac{\sigma_1 \vdash \mathtt{f} \Rightarrow \langle \lambda y.(x \times y), \sigma_0 \rangle \qquad \sigma_1 \vdash 5 \Rightarrow 5 \qquad \sigma_0[y \mapsto 5] \vdash x \times y \Rightarrow 15}{\sigma_1 \vdash \mathtt{f}(5) \Rightarrow 15}$$

$$\textsc{App}\ \frac{\sigma_1 \vdash \mathtt{f} \Rightarrow \langle \lambda y.(x \times y), \sigma_0 \rangle \qquad \sigma_1 \vdash 5 \Rightarrow 5 \qquad \sigma_1[y \mapsto 5] \vdash x \times y \Rightarrow 20}{\sigma_1 \vdash \mathtt{f}(5) \Rightarrow 20}$$

where

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

Summary

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)

Exercise #4

- Please see this document[1] 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/fvae.

Next Lecture

- Lambda Calculus

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