

Lecture 9 – Recursive Functions

COSE212: Programming Languages

Jihyeok Park



2024 Fall

- Syntactic Sugar
 - FAE – Removing `val` from FVAE
 - Syntactic Sugar and Desugaring
- Lambda Calculus (LC)
 - Church Encodings
 - Church-Turing Thesis
- In this lecture, we will learn **recursion** and **conditionals**.
- **RFAE – FAE with recursive functions**
 - Concrete and Abstract Syntax
 - Interpreter and Natural Semantics

1. Recursion and Conditionals
 - Recursion in F1VAE
 - Recursion in FAE
2. Recursion without New Syntax in FAE
 - mkRec: Helper Function for Recursion
3. RFAE – FAE with Recursion and Conditionals
 - Concrete Syntax
 - Abstract Syntax
4. Interpreter and Natural Semantics for RFAE
 - Interpreter and Natural Semantics
 - Arithmetic Comparison Operators
 - Conditionals
 - Recursive Function Definitions

1. Recursion and Conditionals
 - Recursion in F1VAE
 - Recursion in FAE
2. Recursion without New Syntax in FAE
 - mkRec: Helper Function for Recursion
3. RFAE – FAE with Recursion and Conditionals
 - Concrete Syntax
 - Abstract Syntax
4. Interpreter and Natural Semantics for RFAE
 - Interpreter and Natural Semantics
 - Arithmetic Comparison Operators
 - Conditionals
 - Recursive Function Definitions

A **recursive function** is a function that calls itself, and it is useful for **iterative processes** on **inductive data structures**.

Let's define a **recursive function** `sum` that computes the sum of integers from 1 to n in Scala:

```
/* Scala */
def sum(n: Int): Int =
  if (n < 1) 0           // base case
  else n + sum(n - 1)   // recursive case

sum(10) // 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0 = 55
```

For recursive functions, we need **conditionals** to define 1) **base cases** and 2) **recursive cases**.

Most programming languages support **recursive functions**:

- Scala

```
def sum(n: Int): Int = if (n < 1) 0 else n + sum(n - 1)
```

- C++

```
int sum(int n) { return n < 1 ? 0 : n + sum(n - 1); }
```

- Python

```
def sum(n): return 0 if n < 1 else n + sum(n - 1)
```

- Rust

```
fn sum(n: i32) -> i32 { if n < 1 {0} else {n + sum(n-1)} }
```

- ...

The F1VAE language already supports **recursive functions**:

```
/* F1VAE */  
def sum(n) = n + sum(n + -1);  
sum(10)
```

Why? The **function environment** Λ stores all the function definitions before evaluating the expressions.

$$\Lambda = [\text{sum} \mapsto \text{def sum}(n) = n + \text{sum}(n + -1)]$$

We can lookup and invoke the function `sum` in its body.

However, is it enough to support recursive functions?

No! We need **conditionals** to define 1) **base cases** and 2) **recursive cases** for recursive functions. The above example causes an **infinite loop**.

If we only add **conditionals** to F1VAE, we can define recursive functions in F1VAE without any more extensions for recursion.

Programs $\mathbb{P} \ni p ::= f^* e$ (Program)

Function Definitions $\mathbb{F} \ni f ::= \text{def } x(x) = e$ (FunDef)

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

| $e < e$ (Lt)

| $\text{if } (e) e \text{ else } e$ (If)

Values $\mathbb{V} \ni v ::= n \mid \bar{b}$

Function Environments $\Lambda \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{F}$ (FEnv)

Boolean $b \in \mathbb{B} = \{\text{true}, \text{false}\}$ (Boolean)

```
/* F1VAE + conditionals */
def sum(n) = if (n < 1) 0 else n + sum(n + -1);
sum(10) // 55
```

$\Lambda = [\text{sum} \mapsto \text{def sum}(n) = \text{if } (n < 1) 0 \text{ else } n + \text{sum}(n + -1)]$


```
/* FAE + conditionals */  
val sum = n => {  
  if (n < 1) 0  
  else n + sum(n + -1)  
};  
sum(10)
```

What happens if we add **conditionals** to FAE? Is the following FAE expression a recursive function? **No!** `sum` is a **free identifier!** Why?

We use **static scoping** for function definitions in FAE. At the definition site, the variable `sum` is not defined in the environment.

Then, how to support recursive functions in FAE? There are two ways:

- 1 **Without new syntax** – using `mkRec` to define recursive functions
- 2 **With new syntax** – extending FAE with recursive function definitions

1. Recursion and Conditionals
 - Recursion in F1VAE
 - Recursion in FAE
2. Recursion without New Syntax in FAE
 - mkRec: Helper Function for Recursion**
3. RFAE – FAE with Recursion and Conditionals
 - Concrete Syntax
 - Abstract Syntax
4. Interpreter and Natural Semantics for RFAE
 - Interpreter and Natural Semantics
 - Arithmetic Comparison Operators
 - Conditionals
 - Recursive Function Definitions

```
/* FAE + conditionals */  
val sum = n => {  
  if (n < 1) 0  
  else n + sum(n + -1)  
};  
sum(10)
```

How to let `sum` know itself in its body?

Let's pass the function as an argument to itself!

```
/* FAE + conditionals */  
val sumX = sumY => {  
  n => {  
    if (n < 1) 0  
    else n + sumY(sumY)(n + -1)  
  }  
};  
sumX(sumX)(10)
```

However, it is annoying to always pass the function to itself!

Let's wrap this to get sum back!

```
/* FAE + conditionals */
val sum = n => {
  val sumX = sumY => {
    n => {
      if (n < 1) 0
      else n + sumY(sumY)(n + -1)
    }
  };
  sumX(sumX)(n)
};
sum(10)
```

We can simplify this using η -reduction¹:

$$e \quad \equiv \quad \lambda x.e(x) \quad \text{only if } x \text{ is NOT FREE in } e.$$

¹https://en.wikipedia.org/wiki/Lambda_calculus#%CE%B7-reduction

```
/* FAE + conditionals */  
val sum = {  
  val sumX = sumY => {  
    n => { // ALMOST the same as the original body  
      if (n < 1) 0  
      else n + sumY(sumY)(n + -1)  
    }  
  };  
  sumX(sumX)  
};  
sum(10)
```

The function body is almost the same as the original version except that we need to call the function as `sumY(sumY)` instead of `sum`.

Let's define a variable `sum` to be `sumY(sumY)`!

```
/* FAE + conditionals */
val sum = {
  val sumX = sumY => {
    val sum = sumY(sumY);
    n => { // EXACTLY the same as the original body
      if (n < 1) 0
      else n + sum(n + -1)
    }
  };
  sumX(sumX)
};
sum(10)
```

```

/* FAE + conditionals */
val sum = {
  val sumX = sumY => {
    val sum = sumY(sumY); // INFINITE LOOP
    n => { // EXACTLY the same as the original body
      if (n < 1) 0
      else n + sum(n + -1)
    }
  };
  sumX(sumX)
};
sum(10)

```

Unfortunately, this is an **infinite loop!**

We need to **delay** the evaluation of `sum` using the η -**expansion**:

$$e \quad \equiv \quad \lambda x.e(x) \quad \text{only if } x \text{ is } \mathbf{NOT \textit{FREE}} \text{ in } e.$$


```
/* FAE + conditionals */  
val sum = {  
  val sumX = sumY => {  
    val sum = x => sumY(sumY)(x);  
    n => { // EXACTLY the same as the original body  
      if (n < 1) 0  
      else n + sum(n + -1)  
    }  
  };  
  sumX(sumX)  
};  
sum(10)
```

Do we need to do this for every recursive function?

To avoid such boilerplate code, let's define a helper function `mkRec!`

```
/* FAE + conditionals */  
val sum = {  
  val fX = fY => {  
    val sum = x => fY(fY)(x);  
    n => {  
      if (n < 1) 0  
      else n + sum(n + -1)  
    }  
  };  
  fX(fX)  
};  
sum(10)
```

First, we rename `sumX` and `sumY` to `fX` and `fY`, respectively.

```
/* FAE + conditionals */
val sum = {
  val fX = fY => {
    val sum = x => fY(fY)(x);
    n => {
      if (n < 1) 0
      else n + sum(n + -1)
    }
  };
  fX(fX)
};
sum(10)
```

Then, let's desugar the inside variable definition `sum`.

```
/* FAE + conditionals */  
val sum = {  
  val fX = fY => {  
    (sum => n => {  
      if (n < 1) 0  
      else n + sum(n + -1)  
    }) (x => fY(fY)(x))  
  };  
  fX(fX)  
};  
sum(10)
```

```
/* FAE + conditionals */  
val sum = {  
  val fX = fY => {  
    (sum => n => {  
      if (n < 1) 0  
      else n + sum(n + -1)  
    })(x => fY(fY)(x))  
  };  
  fX(fX)  
};  
sum(10)
```

Finally, let's define a helper function `mkRec` that takes a body of a recursive function and returns a recursive function.

```
/* FAE + conditionals */
val mkRec = body => {
  val fX = fY => body(x => fY(fY)(x))
  fX(fX)
};
val sum = mkRec(sum => n => {
  if (n < 1) 0
  else n + sum(n + -1)
});
sum(10)
```

Now, we can also define other recursive functions using `mkRec`². For example, the following recursive function `fac` computes the factorial:

```
/* FAE + conditionals */
val mkRec = ...;
val fac = mkRec(fac => n => if (n < 1) 1 else n * fac(n + -1));
fac(5) // 5 * 4 * 3 * 2 * 1 = 120
```

²https://en.wikipedia.org/wiki/Fixed-point_combinator

1. Recursion and Conditionals
 - Recursion in F1VAE
 - Recursion in FAE
2. Recursion without New Syntax in FAE
 - mkRec: Helper Function for Recursion
3. RFAE – FAE with Recursion and Conditionals
 - Concrete Syntax
 - Abstract Syntax
4. Interpreter and Natural Semantics for RFAE
 - Interpreter and Natural Semantics
 - Arithmetic Comparison Operators
 - Conditionals
 - Recursive Function Definitions

The second way to support recursive functions in FAE is to extend FAE with **recursive function definitions**.

RFAE is an extension of FAE with **recursion** and **conditionals**.

```
/* RFAE */  
def sum(n) = if (n < 1) 0 else n + sum(n + -1);  
sum(10) // 55
```

For RFAE, we need to extend **expressions** of FAE with

- ① **arithmetic comparison operators**
- ② **conditionals**
- ③ **recursive function definitions**


```
/* RFAE */  
def sum(n) = if (n < 1) 0 else n + sum(n + -1);  
sum(10) // 55
```

A **recursive function definition** consists of four parts:

- a function name
- a parameter name
- a function body expression
- a scope expression

Note that a **recursive function definition** is also an expression can be used in any place where an expression is expected:

```
/* RFAE */  
2 * {  
  def sum(n) = if (n < 1) 0 else n + sum(n + -1);  
  sum(10) // 55  
} + 1 // 2 * 55 + 1 = 111
```

```
// expressions
<expr> ::= ...
    | <expr> "<" <expr>
    | "if" "(" <expr> ")" <expr> "else" <expr>
    | "def" <id> "(" <id> ")" "=" <expr> ";" <expr>
```

For RFAE, we need to extend **expressions** of FAE with

- ① **arithmetic comparison operators**
- ② **conditionals**
- ③ **recursive function definitions**

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

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

$e < e$	(Lt)
$\text{if } (e) e \text{ else } e$	(If)
$\text{def } x(x) = e; e$	(Rec)

```
enum Expr:
    ...
    // less-than
    case Lt(left: Expr, right: Expr)
    // conditionals
    case If(cond: Expr, thenExpr: Expr, elseExpr: Expr)
    // recursive function definition
    case Rec(name: String, param: String, body: Expr, scope: Expr)
```

1. Recursion and Conditionals
 - Recursion in F1VAE
 - Recursion in FAE
2. Recursion without New Syntax in FAE
 - mkRec: Helper Function for Recursion
3. RFAE – FAE with Recursion and Conditionals
 - Concrete Syntax
 - Abstract Syntax
4. Interpreter and Natural Semantics for RFAE
 - Interpreter and Natural Semantics
 - Arithmetic Comparison Operators
 - Conditionals
 - Recursive Function Definitions

Now, let's 1) implement the **interpreter**:

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

and 2) define the **natural semantics** for **recursive function definitions** and other new cases.

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

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

| $e < e$ (Lt)

| **if** $(e) e$ **else** e (If)

| **def** $x(x) = e; e$ (Rec)

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

```
enum Value:  
  case NumV(number: BigInt)  
  case BoolV(bool: Boolean)  
  case CloV(param: String, body: Expr, env: Env)
```

```

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

val numLt: BOp[Value] = numCOp(_ < _, "<")

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Lt(l, r) => numLt(interp(l, env), interp(r, env))

```

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

$$\text{Lt} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}$$

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case If(c, t, e) => interp(c, env) match
    case BoolV(true) => interp(t, env)
    case BoolV(false) => interp(e, env)
    case v => error(s"not a boolean: ${v.str}")
```

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

$$\text{If}_T \frac{\sigma \vdash e_0 \Rightarrow \text{true} \quad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \text{if } (e_0) e_1 \text{ else } e_2 \Rightarrow v_1}$$

$$\text{If}_F \frac{\sigma \vdash e_0 \Rightarrow \text{false} \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{if } (e_0) e_1 \text{ else } e_2 \Rightarrow v_2}$$

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(n, p, b, s) =>
    val newEnv: Env = ???
    interp(s, newEnv)
```

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

$$\text{Rec} \frac{\sigma' = ??? \quad \sigma' \vdash e_3 \Rightarrow v_3}{\sigma \vdash \text{def } x_0(x_1) = e_2; e_3 \Rightarrow v_3}$$


```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(n, p, b, s) =>
    val newEnv: Env = env + (n -> CloV(p, b, ???))
    interp(s, newEnv)
```

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

$$\text{Rec} \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda x_1. e_2, ??? \rangle] \quad \sigma' \vdash e_3 \Rightarrow v_3}{\sigma \vdash \text{def } x_0(x_1) = e_2; e_3 \Rightarrow v_3}$$

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(n, p, b, s) =>
    val newEnv: Env = env + (n -> CloV(p, b, newEnv)) // not working
    interp(s, newEnv)
```

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

$$\text{Rec} \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda x_1. e_2, \sigma' \rangle] \quad \sigma' \vdash e_3 \Rightarrow v_3}{\sigma \vdash \text{def } x_0(x_1) = e_2; e_3 \Rightarrow v_3}$$

While it makes sense in the natural semantics, the above Scala code doesn't work because `newEnv` is not yet defined.

Let's **delay** the evaluation of `newEnv` using the η -**expansion** again:

$$e \quad \equiv \quad \lambda x. e(x) \quad \text{only if } x \text{ is } \mathbf{NOT \textit{FREE}} \text{ in } e.$$

We augment the closure value with an **environment factory** $(\ () \Rightarrow \text{Env})$ rather than an **environment** (Env) :

```
enum Value:
  ...
  case CloV(param: String, body: Expr, env: () => Env)

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Func(p, b) => CloV(p, b, () => env)
  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}")
  case Rec(n, p, b, s) =>
    val newEnv: Env = env + (n -> CloV(p, b, () => newEnv)) // error
    interp(s, newEnv)
```

It still doesn't work because `newEnv` is not yet defined.

Let's use a **lazy value** (`lazy val`) to delay the evaluation of `newEnv`.

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(n, p, b, s) =>
    lazy val newEnv: Env = env + (n -> CloV(p, b, () => newEnv))
    interp(s, newEnv)
```

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

$$\text{Rec} \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda x_1. e_2, \sigma' \rangle] \quad \sigma' \vdash e_3 \Rightarrow v_3}{\sigma \vdash \text{def } x_0(x_1) = e_2; e_3 \Rightarrow v_3}$$

We will learn more about **lazy values** in the later lectures in this course.

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

- Please see above document on GitHub:
 - Implement `interp` 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. Recursion and Conditionals
 - Recursion in F1VAE
 - Recursion in FAE
2. Recursion without New Syntax in FAE
 - mkRec: Helper Function for Recursion
3. RFAE – FAE with Recursion and Conditionals
 - Concrete Syntax
 - Abstract Syntax
4. Interpreter and Natural Semantics for RFAE
 - Interpreter and Natural Semantics
 - Arithmetic Comparison Operators
 - Conditionals
 - Recursive Function Definitions

- Mutable Data Structures

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