Lecture 9 – Recursive Functions

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

- 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

Contents

Contents

Recursion

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**.

Recursion

Most programming languages support **recursive functions**:

- Scala

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

- C++

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

- Python

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

- Rust

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

- ...

Recursion in F1VAE

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

$$
\begin{array}{llcll}
\text{Programs} & \mathbb{P} \ni p & ::= & f^* \; e & (\texttt{Program}) \\
\text{Function Definitions} & \mathbb{F} \ni f & ::= & \texttt{def } x(x)\texttt{=}e & (\texttt{FunDef}) \\
\text{Expressions} & \mathbb{E} \ni e & ::= & \ldots & \\
 & & | & e < e & (\texttt{Lt}) \\
 & & | & \texttt{if } (e) \; e \texttt{ else } e & (\texttt{If}) \\
\text{Values} & \mathbb{V} \ni v & ::= & n \mid b \mid \langle \lambda x.e, \sigma \rangle &
\end{array}
$$

$$
\begin{array}{lll}
\text{Function Environments} & \Lambda \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{F} & (\texttt{FEnv}) \\
\text{Boolean} & b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\} & (\texttt{Boolean})
\end{array}
$$

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

$$\Lambda = [\texttt{sum} \mapsto f_0]$$

where $f_0 = \texttt{def sum(n)=if (n} < \texttt{1) 0 else n} + \texttt{sum(n} + -\texttt{1)}$

Recursion in FVAE

However, the following FVAE expression is not a recursive function:

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

Why?

`val` does not support recursive definitions. Thus, `sum` is **NOT** in the scope of the function body!

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

Recursion in FVAE

PLRG

```
/* FVAE */
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 as an argument to itself!

Let's wrap this to get `sum` back!

Recursion in FVAE

```
/* FVAE */
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-**reduction**:

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

Recursion in FVAE

```
/* FVAE */
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)`!

Recursion in FVAE

```
/* FVAE */
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**:

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

Recursion in FVAE

```
/* FVAE */
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`!

mkRec: Helper Function for Recursion

```
/* FVAE */
val mkRec = body => {
  val fX = fY => {
    val f = x => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec(sum => n => { // EXACTLY the same as the original body
  if (n < 1) 0
  else n + sum(n + -1)
});
sum(10)
```

For example, we can define factorial (`fac`) function using `mkRec`:

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

Contents

RFAE – FAE with Recursion and Conditionals

Now, let's extend FAE into RFAE with **recursion** and **conditionals**.

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

```
/* RFAE */
def fib(n) = {
  if (n < 2) n
  else fib(n + -1) + fib(n + -2)
};
fib(7) // 13
```

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

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

Concrete Syntax

```
// 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**

Abstract Syntax

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

$$
\begin{array}{llll}
\text{Expressions} & \mathbb{E} \ni e ::= \ldots & & \\
& \mid e < e & & (\texttt{Lt}) \\
& \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e & & (\texttt{If}) \\
& \mid \texttt{def}\ x(x)\texttt{=}e;\ e & & (\texttt{Rec})
\end{array}
$$

```
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)
```

Contents

Definition with Desugaring

There are two ways to define the semantics of **recursive function definitions** 1) using desugaring or 2) directly defining it.

The first way is to treat **recursive function definitions** as **syntactic sugar** and **desugar** them with mkRec:

$$\begin{array}{rcl}\mathcal{D}[\![\texttt{def}\ x_0(x_1)\texttt{=}e_0;\, e_1]\!] & = & \mathcal{D}[\![\texttt{val}\ x_0\texttt{=mkRec}(\lambda x_0.\lambda x_1.e_0);\ e_1]\!] \\ & = & (\lambda x_0.\mathcal{D}[\![e_1]\!])(\texttt{mkRec}(\lambda x_0.\lambda x_1.\mathcal{D}[\![e_0]\!]))\end{array}$$

```
/* RFAE */
def sum(n) = if (n<1) 0 else n+sum(n+-1); sum(10)
// will be desugared into
(sum => sum(10))(mkRec(sum => (n => if (n<1) 0 else n+sum(n+-1))))
```

```
/* RFAE */
def fib(n) = if(n<2) n else fib(n+-1)+fib(n+-2); fib(7)
// will be desugared into
(fib => fib(7))(mkRec(fib => (n => if(n<2) n else fib(n+-1)+fib(n+-2))))
```

Interpreter and Natural Semantics

The second way is to directly 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.

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

$$
\begin{array}{llll}
\text{Expressions} & \mathbb{E} \ni e ::= & \ldots & \\
& & \mid e < e & (\mathtt{Lt}) \\
& & \mid \mathtt{if}\ (e)\ e\ \mathtt{else}\ e & (\mathtt{If}) \\
& & \mid \mathtt{def}\ x(x)\mathtt{=}e;\ e & (\mathtt{Rec}) \\
\\
\text{Values} & \mathbb{V} \ni v ::= & n \mid b \mid \langle \lambda x.e, \sigma \rangle &
\end{array}
$$

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

Arithmetic Comparison Operators

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

val numLt: (Value, Value) => Value = numCOp("<")(_ < _)

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

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

$$\texttt{Lt}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}$$

Conditionals

```
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}")
```

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

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

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

Recursive Function Definitions

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

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

$$\texttt{Rec}\ \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda x_1.e_0, \sigma' \rangle] \qquad \sigma' \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{def}\ x_0(x_1)\texttt{=}e_0\texttt{;}\ e_1 \Rightarrow v_1}$$

Recursive Function Definitions

PLRG

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

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

$$\texttt{Rec}\ \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda x_1.e_0, \sigma' \rangle] \qquad \sigma' \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{def}\ x_0(x_1)\texttt{=}e_0\texttt{;}\ e_1 \Rightarrow v_1}$$

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

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

Recursive Function Definitions

PLRG

We augment the closure value with an **environment factory** (`() => 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 sill doesn't work because `newEnv` is not yet defined.

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

Recursive Function Definitions

```
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)
```

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

$$\texttt{Rec}\ \frac{\sigma' = \sigma[x_0 \mapsto \langle \lambda x_1.e_0, \sigma' \rangle] \qquad \sigma' \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{def}\ x_0(x_1)\texttt{=}e_0\texttt{;}\ e_1 \Rightarrow v_1}$$

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

Exercise #5

- Please see this document[1] 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] https://github.com/ku-plrg-classroom/docs/tree/main/cose212/rfae.

Summary

1. Recursion
   - Recursion in F1VAE
   - Recursion in FVAE
   - `mkRec`: Helper Function for Recursion
2. RFAE – FAE with Recursion and Conditionals
   - Concrete Syntax
   - Abstract Syntax
3. Interpreter and Natural Semantics for RFAE
   - Definition with Desugaring
   - Interpreter and Natural Semantics
   - Arithmetic Comparison Operators
   - Conditionals
   - Recursive Function Definitions

Next Lecture

- Mutable Data Structures

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