

# Lecture 20 – Typing Recursive Functions

## COSE212: Programming Languages

Jihyeok Park



2024 Fall

- **TFAE** – FAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics
  
- Let's learn how to apply **type system** to recursive functions.
  
- RFAE is an extension of FAE with
  - ① **recursive functions**
  - ② **conditional expressions**
  
- **TRFAE** – RFAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics

## 1. Types for Recursive Functions

Recall: `mkRec` and Recursive Functions  
`mkRec` in TFAE

## 2. TRFAE – RFAE with Type System

Concrete Syntax  
Abstract Syntax

## 3. Type Checker and Typing Rules

Arithmetic Comparison Operators  
Conditionals  
Recursive Function Definitions

## 1. Types for Recursive Functions

Recall: `mkRec` and Recursive Functions  
`mkRec` in TFAE

## 2. TRFAE – RFAE with Type System

Concrete Syntax  
Abstract Syntax

## 3. Type Checker and Typing Rules

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:

```
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
```

We learned two ways to support recursion functions:

① by introducing a **helper function** called `mkRec` in FAE as follows:

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

or ② by adding **new syntax** for recursive functions in RFAE:

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

Can we define `mkRec` in TFAE? **No!** Let's see why.

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

Let's fill out the parts of ??? for type annotations one by one.

```
/* TFAE */
val mkRec = (body: ???) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec((sum: ???) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.



```
/* TFAE */
val mkRec = (body: ???) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => fY(fY)(x);
    body(f)
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: ???) => fY(fY)(x);
    body(f)                                // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: Number) => fY(fY)(x);
    body(f) // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

```
/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ???) => {
    val f = (x: Number) => fY(fY)(x); // fY(fY): Number => Number
    body(f) // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

Let's fill out the parts of ??? for type annotations one by one.

```

/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T) => {
    val f = (x: Number) => fY(fY)(x);    // fY(fY): Number => Number
    body(f)                               // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)

```

Let's fill out the parts of ??? for type annotations one by one.

Let  $T$  be the type of  $fY$ .

Then,  $T$  should be equal to  $T \Rightarrow \text{Number} \Rightarrow \text{Number}$ .

```

/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T => Number => Number) => {
    val f = (x: Number) => fY(fY)(x);    // fY(fY): Number => Number
    body(f)                               // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)

```

Let's fill out the parts of ??? for type annotations one by one.

Let  $T$  be the type of  $fY$ .

Then,  $T$  should be equal to  $T \Rightarrow \text{Number} \Rightarrow \text{Number}$ .

```

/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: (T => Number => Number) => Number => Number) => {
    val f = (x: Number) => fY(fY)(x); // fY(fY): Number => Number
    body(f) // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)

```

Let's fill out the parts of ??? for type annotations one by one.

Let  $T$  be the type of  $fY$ .

Then,  $T$  should be equal to  $T \Rightarrow \text{Number} \Rightarrow \text{Number}$ .



```

/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: ((T => Number => Number) => Number => Number) => Number
    => Number) => {
    val f = (x: Number) => fY(fY)(x); // fY(fY): Number => Number
    body(f) // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)

```

Let's fill out the parts of `???` for type annotations one by one.

Let `T` be the type of `fY`.

Then, `T` should be equal to `T => Number => Number`.

```

/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: (((T => Number => Number) => Number => Number) => Number
    => Number) => Number => Number) => {
    val f = (x: Number) => fY(fY)(x); // fY(fY): Number => Number
    body(f) // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)

```

Let's fill out the parts of `???` for type annotations one by one.

Let `T` be the type of `fY`.

Then, `T` should be equal to `T => Number => Number`.

```

/* TFAE */
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: (((T => Number => Number) => Number => Number) =>
    Number => Number) => Number => Number) => Number => Number) => {
    val f = (x: Number) => fY(fY)(x); // fY(fY): Number => Number
    body(f) // f: Number => Number
  };
  fX(fX)
};
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)

```

Let's fill out the parts of **???** for type annotations one by one.

Let **T** be the type of **fY**.

Then, **T** should be equal to **T => Number => Number**.

We cannot define such **recursive type** in TFAE.

Then, is it possible to define mkRec in Scala?

**Yes!** Since Scala supports **recursive types**, we can define mkRec as:<sup>1</sup>

```
type Number = BigInt
case class T(self: T => Number => Number) // T = T => Number => Number
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T) => {
    val f = (x: Number) => fY.self(fY)(x);
    body(f)
  };
  fX(T(fX))
}
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

<sup>1</sup>This code is given by students 최민석 and 최용욱 in 2023 and slightly modified.

Then, is it possible to define mkRec in Scala?

**Yes!** Since Scala supports **recursive types**, we can define mkRec as:<sup>1</sup>

```
given Conversion[T, T => Number => Number] = _.self
given Conversion[T => Number => Number, T] = T(_)
type Number = BigInt
case class T(self: T => Number => Number) // T = T => Number => Number
val mkRec = (body: (Number => Number) => Number => Number) => {
  val fX = (fY: T) => {
    val f = (x: Number) => fY(fY)(x);
    body(f)
  };
  fX(fX)
}
val sum = mkRec((sum: Number => Number) => (n: Number) =>
  if (n < 1) 0
  else n + sum(n + -1));
sum(10)
```

<sup>1</sup>This code is given by students 최민석 and 최용욱 in 2023 and slightly modified.

## 1. Types for Recursive Functions

Recall: `mkRec` and Recursive Functions  
`mkRec` in TFAE

## 2. TRFAE – RFAE with Type System

Concrete Syntax  
Abstract Syntax

## 3. Type Checker and Typing Rules

Arithmetic Comparison Operators  
Conditionals  
Recursive Function Definitions

Before defining TRFAE, guess the type of the following RFAE expressions:

```
/* RFAE */ def f(n) = n; f
```

Without type annotation for parameter `n`, we cannot guess its type.

```
/* RFAE */ def f(n: Number) = n; f
```

With type annotation for parameter `n`, we can guess its type.

How about this?

```
/* RFAE */ def f(n: Number) = f(n); f
```

Unfortunately, its return type is not clear and actually can be any type.

So, we need **type annotation** for both parameters and return types.

```
/* RFAE */ def f(n: Number): Number = f(n); f
```

Now, let's extend RFAE into TRFAE with **type system**.

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

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

For TRFAE, we need to consider the **type system** of the following cases:

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



We need to add following concrete syntax from RFAE for TRFAE:

- 1 **type annotations** for **recursive function definitions**
- 2 **types** (number, boolean, and arrow types)

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

// types
<type> ::= "(" <type> ")" // only for precedence
  | "Number" // number type
  | "Boolean" // boolean type
  | <type> "=>" <type> // arrow type
```

Similarly, we can define the **abstract syntax** of TRFAE as follows:

Expressions	Types
$\mathbb{E} \ni e ::= \dots$	$\mathbb{T} \ni \tau ::= \text{num} \quad (\text{NumT})$
$e < e \quad (\text{Lt})$	$\text{bool} \quad (\text{BoolT})$
$\text{if } (e) e \text{ else } e \quad (\text{If})$	$\tau \rightarrow \tau \quad (\text{ArrowT})$
$\text{def } x(x:\tau):\tau = e; e \quad (\text{Rec})$	

We can define the abstract syntax of TRFAE in Scala as follows:

```
enum Expr:
  ...
  case Lt(left: Expr, right: Expr)
  case If(cond: Expr, thenExpr: Expr, elseExpr: Expr)
  case Rec(x: String, p: String, pty: Type, rty: Type, b: Expr, s: Expr)
enum Type:
  case NumT
  case BoolT
  case ArrowT(paramTy: Type, retTy: Type)
```

## 1. Types for Recursive Functions

Recall: `mkRec` and Recursive Functions  
`mkRec` in TFAE

## 2. TRFAE – RFAE with Type System

Concrete Syntax  
Abstract Syntax

## 3. Type Checker and Typing Rules

Arithmetic Comparison Operators  
Conditionals  
Recursive Function Definitions

Let's ① design **typing rules** of TRFAE to define when an expression is well-typed in the form of:

$$\Gamma \vdash e : \tau$$

and ② implement a **type checker** in Scala according to typing rules:

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = ???
```

The type checker returns the **type** of  $e$  if it is well-typed, or rejects it and throws a **type error** otherwise.

Similar to TFAE, we will keep track of the **variable types** using a **type environment**  $\Gamma$  as a mapping from variable names to their types.

Type Environments  $\Gamma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}$  (TypeEnv)

```
type TypeEnv = Map[String, Type]
```

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Lt(left, right) =>
    mustSame(typeCheck(left, tenv), NumT)
    mustSame(typeCheck(right, tenv), NumT)
    BoolT
```

$$\boxed{\Gamma \vdash e : \tau}$$

$$\tau\text{-Lt} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 < e_2 : \text{bool}}$$

Type checker should do

- ① check the types of  $e_1$  and  $e_2$  are `num` in  $\Gamma$
- ② return `bool` as the type of  $e_1 < e_2$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

$$\Gamma \vdash e : \tau$$

$$\tau\text{-If} \frac{\text{???}}{\Gamma \vdash \text{if } (e_0) \ e_1 \ \text{else } e_2 : \text{???}}$$

Let's think about the types of the following TRFAE expressions:

<code>if (true) 1 else 2</code>	should be	Number
<code>if (true) 1 else true</code>	might be	Number?
<code>(x: Boolean) =&gt; if (x) 1 else x</code>	cannot have a type	

**Type checker** cannot know the **actual value** of **condition expression**.

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) => ???
```

$$\Gamma \vdash e : \tau$$

$$\tau\text{-If} \frac{\text{???}}{\Gamma \vdash \text{if } (e_0) e_1 \text{ else } e_2 : \text{???}}$$

Let's think about the types of the following TRFAE:

<code>if (true) 1 else 2</code>	should be	<code>Number</code>
<code>if (true) 1 else true</code>		<code>REJECT</code>
<code>(x: Boolean) =&gt; if (x) 1 else x</code>		<code>REJECT</code>

**Type checker** cannot know the **actual value** of **condition expression**.

Let's accept only if **both types** of then- and else-expressions are **same**.

```

def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case If(cond, thenExpr, elseExpr) =>
    mustSame(typeCheck(cond, tenv), BoolT)
    val thenTy = typeCheck(thenExpr, tenv)
    val elseTy = typeCheck(elseExpr, tenv)
    mustSame(thenTy, elseTy)
    thenTy

```

$$\boxed{\Gamma \vdash e : \tau}$$

$$\tau\text{-If} \frac{\Gamma \vdash e_0 : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } (e_0) e_1 \text{ else } e_2 : \tau}$$

Type checker should do

- ① check the type of  $e_0$  is `bool` in  $\Gamma$
- ② check the types of  $e_1$  and  $e_2$  are equal in  $\Gamma$
- ③ return the type of  $e_1$  (or  $e_2$ )



```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    ???
```

$$\Gamma \vdash e : \tau$$
$$\tau\text{-Rec} \frac{\text{???}}{\Gamma \vdash \text{def } x_0(x_1:\tau_1):\tau_2 = e_2; e_3 : \text{???}}$$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    mustSame(typeCheck(body, ???), rty)
    ???
```

$$\Gamma \vdash e : \tau$$

$$\tau\text{-Rec} \frac{??? \vdash e_2 : \tau_2 \quad ???}{\Gamma \vdash \text{def } x_0(x_1 : \tau_1) : \tau_2 = e_2; e_3 : ???}$$

Type checker should do

- 1 check the type of  $e_2$  is  $\tau_2$  in ???
- 2 ???

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>

    mustSame(typeCheck(body, tenv + (p -> pty)), rty)
    ???
```

$$\Gamma \vdash e : \tau$$

$$\tau\text{-Rec} \frac{\Gamma[x_1 : \tau_1] \vdash e_2 : \tau_2 \quad ???}{\Gamma \vdash \text{def } x_0(x_1 : \tau_1) : \tau_2 = e_2; e_3 : ???}$$

Type checker should do

- 1 check the type of  $e_2$  is  $\tau_2$  in the type environment extended with type information for parameter  $(x_1 : \tau_1)$
- 2 ???

```

def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    val fty = ArrowT(pty, rty)
    mustSame(typeCheck(body, tenv + (f -> fty) + (p -> pty)), rty)
    ???

```

$$\boxed{\Gamma \vdash e : \tau}$$

$$\tau\text{-Rec} \frac{\Gamma[x_0 : \tau_1 \rightarrow \tau_2, x_1 : \tau_1] \vdash e_2 : \tau_2 \quad ???}{\Gamma \vdash \text{def } x_0(x_1 : \tau_1) : \tau_2 = e_2; e_3 : ???}$$

Type checker should do

- 1 check the type of  $e_2$  is  $\tau_2$  in the type environment extended with type information for function  $(x_0 : \tau_1 \rightarrow \tau_2)$  and parameter  $(x_1 : \tau_1)$
- 2 ???

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    val fty = ArrowT(pty, rty)
    mustSame(typeCheck(body, tenv + (f -> fty) + (p -> pty)), rty)
    typeCheck(scope, ???)
```

$$\boxed{\Gamma \vdash e : \tau}$$

$$\tau\text{-Rec} \frac{\Gamma[x_0 : \tau_1 \rightarrow \tau_2, x_1 : \tau_1] \vdash e_2 : \tau_2 \quad ??? \vdash e_3 : \tau_3}{\Gamma \vdash \text{def } x_0(x_1 : \tau_1) : \tau_2 = e_2; e_3 : \tau_3}$$

Type checker should do

- 1 check the type of  $e_2$  is  $\tau_2$  in the type environment extended with type information for function  $(x_0 : \tau_1 \rightarrow \tau_2)$  and parameter  $(x_1 : \tau_1)$
- 2 return the type of  $e_3$  in ???

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Rec(f, p, pty, rty, body, scope) =>
    val fty = ArrowT(pty, rty)
    mustSame(typeCheck(body, tenv + (f -> fty) + (p -> pty)), rty)
    typeCheck(scope, tenv + (f -> fty))
```

$$\boxed{\Gamma \vdash e : \tau}$$

$$\tau\text{-Rec} \frac{\Gamma[x_0 : \tau_1 \rightarrow \tau_2, x_1 : \tau_1] \vdash e_2 : \tau_2 \quad \Gamma[x_0 : \tau_1 \rightarrow \tau_2] \vdash e_3 : \tau_3}{\Gamma \vdash \text{def } x_0(x_1 : \tau_1) : \tau_2 = e_2; e_3 : \tau_3}$$

Type checker should do

- ① check the type of  $e_2$  is  $\tau_2$  in the type environment extended with type information for function  $(x_0 : \tau_1 \rightarrow \tau_2)$  and parameter  $(x_1 : \tau_1)$
- ② return the type of  $e_3$  in the type environment extended with type information for function  $(x_0 : \tau_1 \rightarrow \tau_2)$

## 1. Types for Recursive Functions

Recall: `mkRec` and Recursive Functions  
`mkRec` in TFAE

## 2. TRFAE – RFAE with Type System

Concrete Syntax  
Abstract Syntax

## 3. Type Checker and Typing Rules

Arithmetic Comparison Operators  
Conditionals  
Recursive Function Definitions

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

- Please see above document on GitHub:
  - Implement `typeCheck` function.
  - 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.



- Algebraic Data Types (1)

Jihyeok Park

[jihyeok\\_park@korea.ac.kr](mailto:jihyeok_park@korea.ac.kr)

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