

# Lecture 19 – Typed Languages

## COSE212: Programming Languages

Jihyeok Park



2024 Fall

- Safe Language Systems
  - Dynamic vs Static Analysis for Detecting Run-Time Errors
  - Soundness vs Completeness of Analysis
- Type Systems
  - Types
  - Type Errors
  - Type Checking
  - Type Soundness

- Safe Language Systems
  - Dynamic vs Static Analysis for Detecting Run-Time Errors
  - Soundness vs Completeness of Analysis
- Type Systems
  - Types
  - Type Errors
  - Type Checking
  - Type Soundness
- In this lecture, we will define our first **typed language**.

- Safe Language Systems
  - Dynamic vs Static Analysis for Detecting Run-Time Errors
  - Soundness vs Completeness of Analysis
- Type Systems
  - Types
  - Type Errors
  - Type Checking
  - Type Soundness
- In this lecture, we will define our first **typed language**.
- **TFAE** – FAE with **type system**.
  - **Type Checker** and **Typing Rules**
  - Interpreter and Natural Semantics

## 1. TFAE – FAE with Type System

- Concrete Syntax

- Abstract Syntax

## 2. Type Checker and Typing Rules

- Recall: Type Checking

- Type Environment

- Numbers

- Addition and Multiplication

- Immutable Variable Definition and Identifier Lookup

- Function Definition and Application

- Examples

## 3. Interpreter with Type Checker

- Interpreter and Natural Semantics

- Operational Semantics vs Typing Rules

## 1. TFAE – FAE with Type System

Concrete Syntax

Abstract Syntax

## 2. Type Checker and Typing Rules

Recall: Type Checking

Type Environment

Numbers

Addition and Multiplication

Immutable Variable Definition and Identifier Lookup

Function Definition and Application

Examples

## 3. Interpreter with Type Checker

Interpreter and Natural Semantics

Operational Semantics vs Typing Rules

Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say its type is **Number**.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type?



Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say its type is **Number**.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type? **Yes!**  
It should take a **number** type argument and return a **number**.

Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say its type is **Number**.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type? **Yes!**

It should take a **number** type argument and return a **number**.

Let's say its type is **Number** => **Number** called **arrow type**.

Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say its type is **Number**.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type? **Yes!**

It should take a **number** type argument and return a **number**.

Let's say its type is **Number** => **Number** called **arrow type**.

```
/* FAE */ x => x
```

Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say its type is **Number**.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type? **Yes!**

It should take a **number** type argument and return a **number**.

Let's say its type is **Number** => **Number** called **arrow type**.

```
/* FAE */ x => x
```

How about this?

Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say its type is **Number**.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type? **Yes!**

It should take a **number** type argument and return a **number**.

Let's say its type is **Number** => **Number** called **arrow type**.

```
/* FAE */ x => x
```

How about this? There is no information on the parameter  $x$ .

Before defining TFAE, guess the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say its type is **Number**.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type? **Yes!**

It should take a **number** type argument and return a **number**.

Let's say its type is **Number** => **Number** called **arrow type**.

```
/* FAE */ x => x
```

How about this? There is no information on the parameter  $x$ .

One simple solution is to explicitly add **type annotations**!

Let's extend FAE into TFAE with **type annotations** to specify the types of function parameters:

```
/* TFAE */  
(x: Number) => x           // x is `Number` type  
(f: Number => Number) => f // f is `Number => Number` type
```

Let's extend FAE into TFAE with **type annotations** to specify the types of function parameters:

```
/* TFAE */  
(x: Number) => x           // x is `Number` type  
(f: Number => Number) => f // f is `Number => Number` type
```

If we define immutable variable definitions as **syntactic sugar**, it requires the type annotations:  $\mathcal{D}[\text{val } x:\tau = e; e'] = (\lambda x:\tau. \mathcal{D}[e']) (\mathcal{D}[e])$

```
/* TFAE */  
val x: Number = 42; x + 1 // == `((x: Number) => x + 1)(42)`
```



Let's extend FAE into TFAE with **type annotations** to specify the types of function parameters:

```
/* TFAE */  
(x: Number) => x           // x is `Number` type  
(f: Number => Number) => f // f is `Number => Number` type
```

If we define immutable variable definitions as **syntactic sugar**, it requires the type annotations:  $\mathcal{D}[\text{val } x:\tau = e; e'] = (\lambda x:\tau. \mathcal{D}[e']) (\mathcal{D}[e])$

```
/* TFAE */  
val x: Number = 42; x + 1 // == `((x: Number) => x + 1)(42)`
```

However, if we **explicitly define** them rather than syntactic sugar, we can guess variable types from their initial values:

```
/* TFAE */  
val x = 42; x + 1 // x is `Number` type because of `42`
```

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

- ① **function definitions** with **type annotations**
- ② **immutable variable definitions** without **type annotations**
- ③ **types**

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

- 1 **function definitions** with **type annotations**
- 2 **immutable variable definitions** without **type annotations**
- 3 **types**

We can extend the **concrete syntax** of FAE as follows:

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

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

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

- 1 **function definitions** with **type annotations**
- 2 **immutable variable definitions** without **type annotations**
- 3 **types**

We can extend the **concrete syntax** of FAE as follows:

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

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

Since functions are first-class values, the parameter and return types could be recursively arrow types. And,  $\Rightarrow$  is **right-associative**.

We can extend the **abstract syntax** of FAE for TFAE as follows:

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

|  $\lambda x:\tau.e$  (Fun)

|  $\text{val } x = e; e$  (Val)

Types  $\mathbb{T} \ni \tau ::= \text{num}$  (NumT)

|  $\tau \rightarrow \tau$  (ArrowT)

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

```
enum Expr:
  ...
  case Fun(param: String, ty: Type, body: Expr)
  case Val(name: String, init: Expr, body: Expr)

enum Type:
  case NumT
  case ArrowT(paramTy: Type, retTy: Type)
```

## 1. TFAE – FAE with Type System

Concrete Syntax

Abstract Syntax

## 2. Type Checker and Typing Rules

Recall: Type Checking

Type Environment

Numbers

Addition and Multiplication

Immutable Variable Definition and Identifier Lookup

Function Definition and Application

Examples

## 3. Interpreter with Type Checker

Interpreter and Natural Semantics

Operational Semantics vs Typing Rules

If the following conditions hold, we say “**the expression  $e$  has type  $\tau$** ”:

- $e$  does not cause any type error, and
- $e$  evaluates to a value of type  $\tau$  or does not terminate.

If so, we use the following notation and say that  $e$  is **well-typed**:

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

### Definition (Type Checking)

**Type checking** is a kind of static analysis checking whether a given expression  $e$  is **well-typed**. A **type checker** returns the **type** of  $e$  if it is well-typed, or rejects it and reports the detected **type error** otherwise.

If the following conditions hold, we say “**the expression  $e$  has type  $\tau$** ”:

- $e$  does not cause any type error, and
- $e$  evaluates to a value of type  $\tau$  or does not terminate.

If so, we use the following notation and say that  $e$  is **well-typed**:

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

### Definition (Type Checking)

**Type checking** is a kind of static analysis checking whether a given expression  $e$  is **well-typed**. A **type checker** returns the **type** of  $e$  if it is well-typed, or rejects it and reports the detected **type error** otherwise.

We need to

- ① design **typing rules** to define when an expression is well-typed
- ② implement a **type checker** in Scala according to typing rules



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

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

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

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

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

```
def typeCheck(expr: Expr): Type = ???
```

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

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

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

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

```
def typeCheck(expr: Expr): Type = ???
```

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

In addition, we need to keep track of the **variable types**.

Let's ① design **typing rules** of TFAE 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.

In addition, we need to keep track of the **variable types**.

Let's define a **type environment**  $\Gamma$  as a mapping from variable names to their types and pass it to the type checker.

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

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

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  case Num(_) => ???
  ...
```

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

$$\tau\text{-Num} \frac{}{\Gamma \vdash n : ???}$$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  case Num(_) => NumT
  ...
```

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

$$\tau\text{-Num} \frac{}{\Gamma \vdash n : \text{num}}$$

The number literal  $n$  has `num` type in any type environment  $\Gamma$ .

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Add(left, right) =>
    ???
```

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

$$\tau\text{-Add} \frac{\text{???}}{\Gamma \vdash e_1 + e_2 : \text{???}}$$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Add(left, right) =>
    typeCheck(left, tenv)
    ???
```

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

$$\tau\text{-Add} \frac{\Gamma \vdash e_1 : \tau \quad ???}{\Gamma \vdash e_1 + e_2 : ???}$$

Type checker should do

- 1 get the type of  $e_1$  in  $\Gamma$



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

def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type mismatch: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{-Add} \frac{\Gamma \vdash e_1 : \text{num} \quad ???}{\Gamma \vdash e_1 + e_2 : ???}$$

Type checker should do

- 1 check the type of  $e_1$  is `num` in  $\Gamma$

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

def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type mismatch: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{-Add} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : ???}$$

Type checker should do

- 1 check the types of  $e_1$  and  $e_2$  are num in  $\Gamma$

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

def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type mismatch: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{-Add} \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}}$$

Type checker should do

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

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

def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type mismatch: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{-Mul} \quad \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 * e_2 : \text{num}}$$

Type checker should do

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

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
...
case Val(x, init, body) =>
  val initTy = typeCheck(init, tenv)
  typeCheck(body, tenv + (x -> initTy))
```

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

$$\tau\text{-Val} \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x : \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \text{val } x = e_1; e_2 : \tau_2}$$

This rule stores the type of  $x$  in  $\Gamma$  inferred from the initial value.

```
/* TFAE */ val x = 1; x + 2      // `x: Number` in `tenv` <- `1: Number`
```

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Id(x) =>
    tenv.getOrElse(x, error(s"free identifier: $x"))
```

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

$$\tau\text{-Id} \frac{x \in \text{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

This rule looks up the type of  $x$  in  $\Gamma$ .

```
/* TFAE */ val x = 1; x + 2      // `x: Number` in `tenv` <- `1: Number`
```

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Fun(param, paramTy, body) =>
    val retTy = typeCheck(body, tenv + (param -> paramTy))
    ArrowT(paramTy, retTy)
```

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

$$\tau\text{-Fun} \frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'}$$

We can check the body of a function with the its parameter type.

```
/* TFAE */ (x: Number) => x      // Number => Number
```

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
...
case App(fun, arg) => typeCheck(fun, tenv) match
  case ArrowT(paramTy, retTy) =>
    mustSame(typeCheck(arg, tenv), paramTy)
    retTy
  case ty => error(s"not a function type: ${ty.str}")
```

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

$$\tau\text{-App} \frac{\Gamma \vdash e_0 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0(e_1) : \tau_2}$$

We don't have to check the type of the function body because it is already checked when the function is defined.

```
/* TFAE */ ((x: Number) => x)(1) // Number
```



```
/* TFAE */ val x = 1; x + 2           // 3: Number
```

```
/* TFAE */ val x = 1; x + 2           // 3: Number
```

$$\frac{\frac{\frac{}{\emptyset \vdash 1 : \text{num}}}{x \in \text{Domain}([x : \text{num}])} \quad \frac{[x : \text{num}] \vdash x : \text{num} \quad [x : \text{num}] \vdash 2 : \text{num}}{[x : \text{num}] \vdash x + 2 : \text{num}}}{\emptyset \vdash \text{val } x = 1; x + 2 : \text{num}}$$

```
/* TFAE */ val x = 1; x + 2           // 3: Number
```

$$\frac{\frac{\frac{}{\emptyset \vdash 1 : \text{num}}}{x \in \text{Domain}([x : \text{num}])} \quad \frac{\frac{[x : \text{num}] \vdash x : \text{num}}{[x : \text{num}] \vdash 2 : \text{num}}}{[x : \text{num}] \vdash x + 2 : \text{num}}}{\emptyset \vdash \text{val } x = 1; x + 2 : \text{num}}$$

```
/* TFAE */ ((x: Number) => x)(2) * 3   // 3: Number
```

```
/* TFAE */ val x = 1; x + 2           // 3: Number
```

$$\frac{\frac{\frac{}{\emptyset \vdash 1 : \text{num}}}{x \in \text{Domain}([x : \text{num}])} \quad \frac{\frac{}{[x : \text{num}] \vdash x : \text{num}} \quad \frac{}{[x : \text{num}] \vdash 2 : \text{num}}}{[x : \text{num}] \vdash x + 2 : \text{num}}}{\emptyset \vdash \text{val } x = 1; x + 2 : \text{num}}$$

```
/* TFAE */ ((x: Number) => x)(2) * 3   // 3: Number
```

$$\frac{\frac{\frac{\frac{}{\emptyset \vdash \lambda x : \text{num}. x : \text{num} \rightarrow \text{num}}}{x \in \text{Domain}([x : \text{num}])} \quad \frac{}{\emptyset \vdash 2 : \text{num}}}{\emptyset \vdash (\lambda x : \text{num}. x)(2) : \text{num}} \quad \frac{}{\emptyset \vdash 3 : \text{num}}}{\emptyset \vdash (\lambda x : \text{num}. x)(2) * 3 : \text{num}}$$

## 1. TFAE – FAE with Type System

Concrete Syntax

Abstract Syntax

## 2. Type Checker and Typing Rules

Recall: Type Checking

Type Environment

Numbers

Addition and Multiplication

Immutable Variable Definition and Identifier Lookup

Function Definition and Application

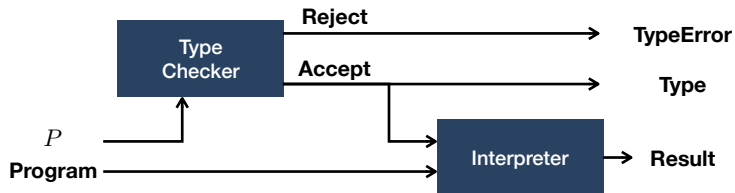
Examples

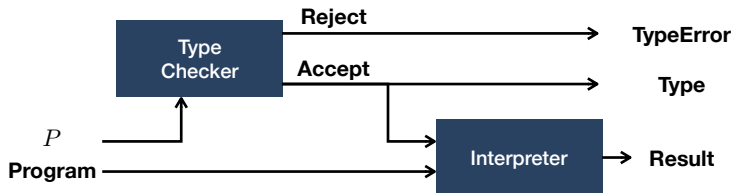
## 3. Interpreter with Type Checker

Interpreter and Natural Semantics

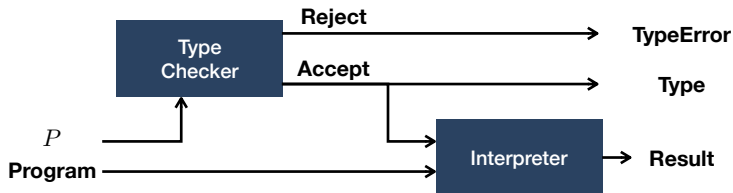
Operational Semantics vs Typing Rules

# Interpreter with Type Checker





```
def eval(str: String): String =  
  val expr = Expr(str)  
  val ty = typeCheck(expr, Map.empty)  
  val v = interp(expr, Map.empty)  
  s"${v.str}: ${ty.str}"
```



```
def eval(str: String): String =  
  val expr = Expr(str)  
  val ty = typeCheck(expr, Map.empty)  
  val v = interp(expr, Map.empty)  
  s"${v.str}: ${ty.str}"
```

```
eval("val x = 1; x + 2")           // 3: Number  
eval("((x: Number) => x)(2) * 3") // 6: Number  
eval("1 + (x: Number) => x")      // a type error thrown by `typeCheck`
```



For interpreter and natural semantics for TFAE, it is just enough to extend the those for function definitions in FAE.

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

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

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

The type annotation is ignored in the interpreter and natural semantics.

What is the difference between **operational semantics** and **typing rules**?

$$\boxed{\sigma \vdash e \Rightarrow v} \quad \text{vs} \quad \boxed{\Gamma \vdash e : \tau}$$

What is the difference between **operational semantics** and **typing rules**?

$$\boxed{\sigma \vdash e \Rightarrow v} \quad \text{vs} \quad \boxed{\Gamma \vdash e : \tau}$$

See the table below for the comparison.

	<b>Operational Semantics</b>	<b>Typing Rules</b>
<b>Mathematical Notation</b>	$\sigma \vdash e \Rightarrow v$	$\Gamma \vdash e : \tau$
<b>Dynamic/Static</b>	Dynamic	Static
<b>Concrete/Abstract</b>	Concrete	Abstract
<b>Purpose</b>	Evaluation	Type Checking
<b>Implementation</b>	Interpreter	Type Checker
<b>Result</b>	Value	Type

## 1. TFAE – FAE with Type System

- Concrete Syntax

- Abstract Syntax

## 2. Type Checker and Typing Rules

- Recall: Type Checking

- Type Environment

- Numbers

- Addition and Multiplication

- Immutable Variable Definition and Identifier Lookup

- Function Definition and Application

- Examples

## 3. Interpreter with Type Checker

- Interpreter and Natural Semantics

- Operational Semantics vs Typing Rules

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

- 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.

- Typing Recursive Functions

Jihyeok Park

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

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