

Lecture 24 – Subtype Polymorphism

COSE212: Programming Languages

Jihyeok Park



2025 Fall

- **Polymorphism** is to use a single entity as **multiple types**, and there are various kinds of polymorphism:
 - **Parametric polymorphism**
 - **Subtype polymorphism**
 - **Ad-hoc polymorphism**
 - ...
- **Parametric polymorphism** is a form of polymorphism by introducing **type variables** and instantiating them with **type arguments**.
- **PTFAE** – TFAE with **parametric polymorphism**.
- In this lecture, we will learn **subtype polymorphism**.
- **STFAE** – TFAE with **subtype polymorphism**.
 - **Interpreter** and **Natural Semantics**
 - **Type Checker** and **Typing Rules**

Contents

1. Subtype Polymorphism
2. STFAE – TFAE with Subtype Polymorphism
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for STFAE
4. Type Checker and Typing Rules without Subtyping
 - Records
 - Field Accesses
 - Exit
 - Immutable Variable Definition
5. Subtype Relation
 - Bottom Type and Top Type
 - Record Types
 - Function Types
 - Typing Rules with Subsumption
 - Algorithmic Typing Rules without Subsumption

Contents

1. Subtype Polymorphism

2. STFAE – TFAE with Subtype Polymorphism

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for STFAE

4. Type Checker and Typing Rules without Subtyping

Records

Field Accesses

Exit

Immutable Variable Definition

5. Subtype Relation

Bottom Type and Top Type

Record Types

Function Types

Typing Rules with Subsumption

Algorithmic Typing Rules without Subsumption

Subtype Polymorphism

To easily explain **subtype polymorphism**, let's support **records** and **record types** to TFAE. (Also, optional type annotations for **val**.)

```
/* STFAE */  
// A record with two fields `a` and `b` whose types are `Number`  
val x: {a: Number, b: Number} = {a=1, b=2}  
x.a    // Access the field `a` of `x` and evaluate to `1`
```

Consider the following expression:

```
/* STFAE */  
val f = (x: ???) => x.a  
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

Unfortunately, we cannot assign any type to x because the type of x should be ① {a: Number}, ② {a: Number, b: Number}, and ③ {c: Number, a: Number}, simultaneously.

How can we resolve this problem? **Subtype Polymorphism!**

Definition (Subtype Polymorphism)

Subtype polymorphism is a form of polymorphism by introducing **subtype relations** between types.

```
/* STFAE */  
val f = (x: {a: Number}) => x.a      // Allow subtypes of `{a: Number}`  
f({a=1}) + f({a=2, b=3}) + f({c=4, a=5})
```

All the following types are **subtypes** of {a: Number}:

{a: Number}	{a: Number, b: Number}
{c: Number, a: Number}	...

It corresponds to the **subset relation** between sets in mathematics, and most programming languages support **subtype polymorphism**.

Subtype relations could be defined for other types (e.g., functions, lists, pairs, data types etc.) as well.

Contents

1. Subtype Polymorphism

2. STFAE – TFAE with Subtype Polymorphism

Concrete Syntax

Abstract Syntax

3. Interpreter and Natural Semantics for STFAE

4. Type Checker and Typing Rules without Subtyping

Records

Field Accesses

Exit

Immutable Variable Definition

5. Subtype Relation

Bottom Type and Top Type

Record Types

Function Types

Typing Rules with Subsumption

Algorithmic Typing Rules without Subsumption

Now, let's extend TFAE into STFAE to support **subtype polymorphism** with **records** and optional type annotations for **val**.

```
/* STFAE */
val x: {a: Number} = {a=1}           // A record with a field `a`
x.a                                // Access the field `a` of `x`
val x: {a: Top, b: Top} = {a=1, b=x} // `Top` is the top type
val x: Bot = exit                   // Exit the program
...                                 // Not evaluated after `exit`
```

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

- ① **Optional Type Annotations for val**
- ② **Records**
- ③ **Field Accesses**
- ④ **Exit** (to immediately exit the program)
- ⑤ **Record Types**
- ⑥ **Bottom Type** (corresponding to the empty set)
- ⑦ **Top Type** (corresponding to the universal set)

- ① Optional Type Annotations for `val`
- ② Records
- ③ Field Accesses
- ④ Exit (to immediately exit the program)
- ⑤ Record Types
- ⑥ Bottom Type (corresponding to the empty set)
- ⑦ Top Type (corresponding to the universal set)

```
<expr>      ::= ...
              | "val" <id> [ ":" <type> ]? "=" <expr> ";"? <expr>
              | "{" [<id> "=" <expr>]*{",","} "}"
              | <expr> "." <id>
              | "exit"
<type>      ::= ...
              | "{" [<id> ":" <type>]*{",","} "}"
              | "Bot"
              | "Top"
```

Duplicate field names are not allowed in records and record types.

Expressions	$\mathbb{E} \ni e ::= \dots$	$\mid \{[x = e]^*\} \text{ (Record)}$
	$\mid \text{val } x [:\tau]^? = e; e \text{ (Val)}$	$\mid e.x \text{ (Access)}$
		$\mid \text{exit} \text{ (Exit)}$
Types	$\mathbb{T} \ni \tau ::= \dots$	$\mid \perp \text{ (BotT)}$
	$\mid \{[x : \tau]^*\} \text{ (RecordT)}$	$\mid \top \text{ (TopT)}$

```

enum Expr:
    ...
    case Val(name: String, tyOpt: Option[Type], init: Expr, body: Expr)
    case Record(fields: List[(String, Expr)])
    case Access(record: Expr, field: String)
    case Exit

enum Type:
    ...
    case RecordT(fields: Map[String, Type])
    case BotT
    case TopT

```

Contents

1. Subtype Polymorphism
2. STFAE – TFAE with Subtype Polymorphism
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for STFAE
4. Type Checker and Typing Rules without Subtyping
 - Records
 - Field Accesses
 - Exit
 - Immutable Variable Definition
5. Subtype Relation
 - Bottom Type and Top Type
 - Record Types
 - Function Types
 - Typing Rules with Subsumption
 - Algorithmic Typing Rules without Subsumption

For STFAE, we need to 1) implement the **interpreter** with environments:

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

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

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

with a new kind of values called **record values**:

$$\begin{array}{ll} \text{Values} & \forall \exists v ::= n \quad (\text{NumV}) \\ & | \langle \lambda x.e, \sigma \rangle \quad (\text{CloV}) \\ & | \{[x = v]^*\} \quad (\text{RecordV}) \end{array}$$

```
enum Value:  
  case NumV(number: BigInt)  
  case CloV(param: String, body: Expr, env: Env)  
  case RecordV(fields: Map[String, Value])
```

```

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Record(fs) =>
    RecordV(fs.map { case (f, e) => (f, interp(e, env)) }.toMap)
  case Access(r, f) => interp(r, env) match
    case RecordV(fs) => fs.getOrElse(f, error(s"no such field: $f"))
    case v             => error(s"not a record: ${v.str}")
  
```

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

$$\text{Record} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \dots \quad \sigma \vdash e_n \Rightarrow v_n}{\sigma \vdash \{x_1 = e_1, \dots, x_n = e_n\} \Rightarrow \{x_1 = v_1, \dots, x_n = v_n\}}$$

$$\text{Access} \frac{\sigma \vdash e \Rightarrow \{\dots, x = v, \dots\}}{\sigma \vdash e.x \Rightarrow v}$$

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Exit => error("exit")
```

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

There is no rule for exit because it cannot produce any value.

We cannot draw the derivation tree for the following expression:

```
/* STFAE */ 1 + exit
```

However, we can draw the derivation tree for the following expression:

```
/* STFAE */ (x: Number) => 1 + exit
```

$$\text{Fun} \frac{}{\emptyset \vdash \lambda x:\text{num}.(1 + \text{exit}) \Rightarrow \langle \lambda x.(1 + \text{exit}), \emptyset \rangle}$$

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Val(name, _, expr, body) =>
    interp(body, env + (name -> interp(expr, env)))
```

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

$$\text{Val} \quad \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x = e_1; e_2 \Rightarrow v_2}$$

$$\text{Val}_\tau \quad \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x : \tau_0 = e_1; e_2 \Rightarrow v_2}$$

Contents

1. Subtype Polymorphism
2. STFAE – TFAE with Subtype Polymorphism
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for STFAE
4. Type Checker and Typing Rules without Subtyping
 - Records
 - Field Accesses
 - Exit
 - Immutable Variable Definition
5. Subtype Relation
 - Bottom Type and Top Type
 - Record Types
 - Function Types
 - Typing Rules with Subsumption
 - Algorithmic Typing Rules without Subsumption

Type Checker and Typing Rules without Subtyping

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

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

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** Γ 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]
```

Records

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Record(fields) =>
    RecordT(fields.map { case (f, e) => (f, typeCheck(e, tenv)) }.toMap)
```

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

$$\tau\text{-Record} \frac{\Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{x_1 = e_1, \dots, x_n = e_n\} : \{x_1 : \tau_1, \dots, x_n : \tau_n\}}$$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Access(record, f) => typeCheck(record, tenv) match
    case RecordT(fs) => fs.getOrElse(f, error(s"no such field: $f"))
    case ty           => error(s"not a record type: ${ty.str}")
```

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

$$\tau\text{-Access} \quad \frac{\Gamma \vdash e : \{\dots, x : \tau, \dots\}}{\Gamma \vdash e.x : \tau}$$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Exit => BotT
```

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

$$\tau\text{-Exit} \frac{}{\Gamma \vdash \text{exit} : \perp}$$

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Val(name, tyOpt, expr, body) =>
    val ty = typeCheck(expr, tenv)

    typeCheck(body, tenv + (name -> ty))
```

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

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

Immutable Variable Definition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Val(name, tyOpt, expr, body) =>
    val ty = typeCheck(expr, tenv)
    tyOpt.map(givenTy => mustEqual(ty, givenTy))
    val nameTy = tyOpt.getOrElse(ty)
    typeCheck(body, tenv + (name -> nameTy))
```

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

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

$$\tau\text{-Val}_\tau \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 = \tau_0 \quad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \text{val } x : \tau_0 = e_1; e_2 : \tau_2}$$

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

$$\tau\text{-Val}_\tau \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 = \tau_0 \quad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \text{val } x : \tau_0 = e_1; e_2 : \tau_2}$$

Consider the following example:

```
/* STFAE */
val x: {a: Number} = {a=2, b=3}; x.a
```

It fails to type check because:

$$\{a: \text{Number}, b: \text{Number}\} \neq \{a: \text{Number}\}$$

Let's apply **subtype polymorphism** to fix this problem by introducing a **subtype relation** ($<:$) between types.

Contents

1. Subtype Polymorphism
2. STFAE – TFAE with Subtype Polymorphism
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for STFAE
4. Type Checker and Typing Rules without Subtyping
 - Records
 - Field Accesses
 - Exit
 - Immutable Variable Definition
5. Subtype Relation
 - Bottom Type and Top Type
 - Record Types
 - Function Types
 - Typing Rules with Subsumption
 - Algorithmic Typing Rules without Subsumption

To support **subtype polymorphism**, we need to define a **subtype relation** $<:$ between types.

$$\boxed{\tau <: \tau}$$

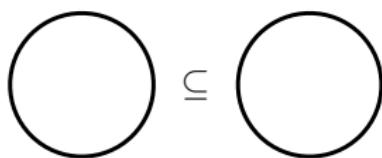
$\tau <: \tau'$ denotes τ is a **subtype** of τ' (or τ' is a **super type** of τ).

First, **subtype relation** is **reflexive** and **transitive**:

$$\frac{}{\tau <: \tau}$$

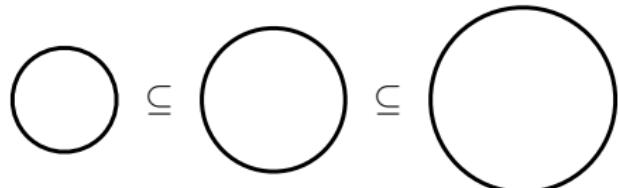
$$\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''}$$

$$\tau \quad <: \quad \tau$$



Reflexivity

$$\tau \quad <: \quad \tau' \quad <: \quad \tau''$$



Transitivity

Subtype Relation – Bottom Type and Top Type

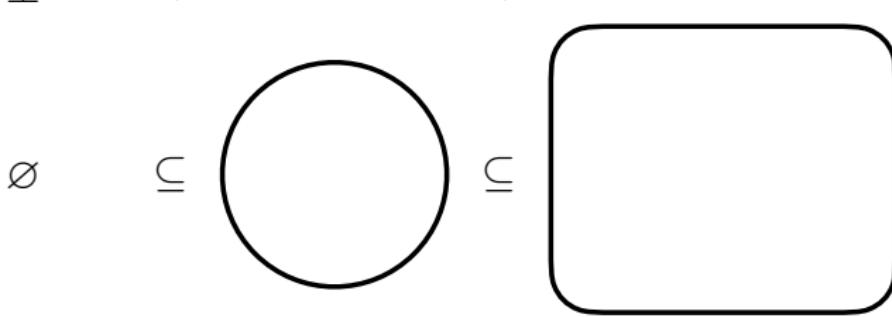
$$\tau <: \tau$$

The **bottom type** \perp and the **top type** \top represent the **empty set** of values and the **universal set** of values, respectively.

Thus, \perp is a subtype of any type, and any type is a subtype of \top :

$$\overline{\perp <: \tau} \quad \overline{\tau <: \top}$$

$$\perp <: \tau \quad \tau <: \top$$



Value Set \mathbb{V}

Subtype Relation – Record Types (1)

$$\tau <: \tau$$

Let's consider the subtype relation between **record types**.

```
/* STFAE */
val f = (x: {a: Number}) => x.a
val y: {a: Number, b: Number} = {a = 1, b = 2}
f(y)
```

If we **add** any new field to a record type, the resulting type should be a subtype of the original type.

$$\overline{\{x_1 : \tau_1, \dots, x_n : \tau_n, x : \tau\} <: \{x_1 : \tau_1, \dots, x_n : \tau_n\}}$$

Subtype Relation – Record Types (2)

$$\tau <: \tau$$

Let's consider the subtype relation between **record types**.

```
/* STFAE */  
val f = (x: {a: Top, b: Top}) => x  
val x: {a: Number, b: Number} = {a = 1, b = 2}  
f(x)
```

If all fields of a record type are **subtypes** of the corresponding fields of another record type, the resulting type should be a subtype of the other.

$$\frac{\tau_1 <: \tau'_1 \quad \dots \quad \tau_n <: \tau'_n}{\{x_1 : \tau_1, \dots, x_n : \tau_n\} <: \{x_1 : \tau'_1, \dots, x_n : \tau'_n\}}$$

$$\tau <: \tau$$

Let's consider the subtype relation between **record types**.

```
/* STFAE */  
val f = (x: {a: Number, b: Number}) => x  
val x: {b: Number, a: Number} = {a = 1, b = 2}  
f(x)
```

If the fields of a record type is a **permutation** of the fields of another record type, the resulting type should be a subtype of the other.

$$\frac{\{x_1 : \tau_1, \dots, x_n : \tau_n\} \text{ is a permutation of } \{x'_1 : \tau'_1, \dots, x'_n : \tau'_n\}}{\{x_1 : \tau_1, \dots, x_n : \tau_n\} <: \{x'_1 : \tau'_1, \dots, x'_n : \tau'_n\}}$$

Subtype Relation – Function Types

$$\tau <: \tau$$

Let's consider the subtype relation between **function types**.

```
val f = (g: Number => {a: Top}) => g(42).a
```

```
val h: Top => {a: Top} = (x: Top) => {a: x}
// (Top => {a: Top}) <: (Number => {a: Top})
f(h)
```

If the **parameter type** τ_1 is a **super type** of the **parameter type** τ'_1 ,
 $\tau_1 \rightarrow \tau_2$ is a subtype of $\tau'_1 \rightarrow \tau_2$.

```
val h: Number => {a: Number} = (x: Number) => {a: x}
// (Number => {a: Number}) <: (Number => {a: Top})
f(h)
```

Reversely, if the **return type** τ_2 is a **subtype** of the **return type** τ'_2 ,
 $\tau_1 \rightarrow \tau_2$ is a subtype of $\tau_1 \rightarrow \tau'_2$.

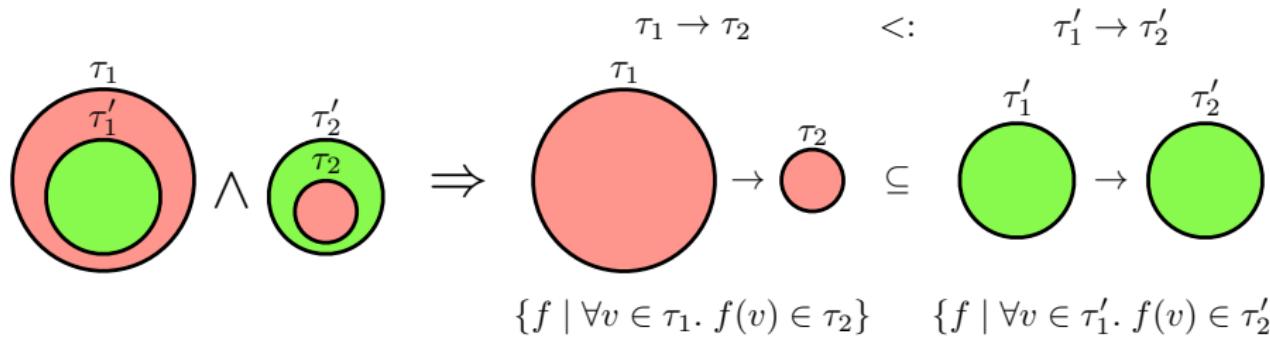
Subtype Relation – Function Types

$$\tau <: \tau$$

If the following conditions are satisfied, $\tau_1 \rightarrow \tau_2$ is a subtype of $\tau'_1 \rightarrow \tau'_2$.

- the **parameter type** τ_1 is a **super type** of the **parameter type** τ'_1 .
- the **return type** τ_2 is a **subtype** of the **return type** τ'_2 .

$$\frac{\tau_1 :> \tau'_1 \quad \tau_2 <: \tau'_2}{(\tau_1 \rightarrow \tau_2) <: (\tau'_1 \rightarrow \tau'_2)}$$



Typing Rules with Subsumption

One possible way to support subtype polymorphism is to add a general **subsumption** rule to the typing rules:

$$\frac{\Gamma \vdash e : \tau \quad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

With this rule, we can type the following expression.

```
/* STFAE */
val x: {a: Number} = {a=2, b=3}; x.a
```

$$\frac{\begin{array}{c} \dots \\ \overline{\emptyset \vdash \{a = 2, b = 3\} : \{a : \text{num}, b : \text{num}\}} \quad \overline{\{a : \text{num}, b : \text{num}\} <: \{a : \text{num}\}} \\ \dots \\ \overline{\emptyset \vdash \{a = 2, b = 3\} : \{a : \text{num}\}} \end{array}}{\overline{\emptyset \vdash \text{val } x : \{a : \text{num}\} = \{a = 2, b = 3\}; x.a : \text{num}}}$$

However, it is **not algorithmic** because we don't know which types are required as the result of subsumption.

Algorithmic Typing Rules without Subsumption

Another way is to **directly apply** the subtype relation to the each typing rule **without subsumption rule**, and it is **algorithmic**.

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

$$\tau\text{-Add} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{num} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 <: \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}}$$

$$\tau\text{-Mul} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \text{num} \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 <: \text{num}}{\Gamma \vdash e_1 * e_2 : \text{num}}$$

$$\tau\text{-Val}_\tau \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \tau_1 <: \tau_0 \quad \Gamma[x : \tau_0] \vdash e_2 : \tau_2}{\Gamma \vdash \text{val } x : \tau_0 = e_1; e_2 : \tau_2}$$

$$\tau\text{-App} \quad \frac{\Gamma \vdash e_0 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_3 \quad \tau_3 <: \tau_1}{\Gamma \vdash e_0(e_1) : \tau_2}$$

Summary

1. Subtype Polymorphism
2. STFAE – TFAE with Subtype Polymorphism
 - Concrete Syntax
 - Abstract Syntax
3. Interpreter and Natural Semantics for STFAE
4. Type Checker and Typing Rules without Subtyping
 - Records
 - Field Accesses
 - Exit
 - Immutable Variable Definition
5. Subtype Relation
 - Bottom Type and Top Type
 - Record Types
 - Function Types
 - Typing Rules with Subsumption
 - Algorithmic Typing Rules without Subsumption

Exercise #15

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

- 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.

Next Lecture

- Type Inference (1)

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