

# Lecture 18 – Subtyping

## AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Polymorphism
- Universal Types
  - Syntax and Dynamic Semantics
  - Well-Formedness
  - Type Equivalence
  - Typing Rules
  - Type Erasure
  - Examples
  - Normalization

## 1. Subtyping

## 2. STLC with Subtyping

- Typing Rules

- Subtype Relation

- Variance

- Algorithmic Typing Rules

## 3. Other Extensions

- Exceptions

- Records

- Conditional Expressions

## 4. Bounded Quantification

## 1. Subtyping

## 2. STLC with Subtyping

Typing Rules

Subtype Relation

Variance

Algorithmic Typing Rules

## 3. Other Extensions

Exceptions

Records

Conditional Expressions

## 4. Bounded Quantification

A **polymorphism** is to allow **multiple types** for a single term.

There are three representative polymorphisms:

- **Subtype polymorphism** defines a subtype relation between types, and allows a term of a type to be used as a term of its supertypes.
- **Ad-hoc polymorphism** allows different implementations for the same function name with different types (e.g., overloading, type classes).
- **Parametric polymorphism** defines a term with type variables, and allows the term to be instantiated with different types.

In this lecture, we will focus on **subtype polymorphism**.

Subtyping is a relation between types, denoted as  $S <: T$ , which means that type  $S$  is a **subtype** of type  $T$ .

If  $S <: T$ , then a term of type  $S$  can be used in any context where a term of type  $T$  is expected.

For example, if we have a type hierarchy where `Dog <: Animal`, then we can use a term of type `Dog` wherever a term of type `Animal` is expected.

```
trait Animal:
  def speak: String

case class Dog(name: String) extends Animal:
  def speak: String = name + " says woof."

val dog: Dog = Dog("Buddy")
val animal: Animal = dog // Upcasting from Dog to Animal
animal.speak             // Output: "Buddy says woof."
```

## 1. Subtyping

## 2. STLC with Subtyping

- Typing Rules

- Subtype Relation

- Variance

- Algorithmic Typing Rules

## 3. Other Extensions

- Exceptions

- Records

- Conditional Expressions

## 4. Bounded Quantification

Expressions  $e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e$

Types  $\tau ::= \text{num} \mid \tau \rightarrow \tau \mid \perp \mid \top$

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

$$\frac{}{\Gamma \vdash n : \text{num}} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{\Gamma \vdash e_1 : \text{num} \quad \Gamma \vdash e_2 : \text{num}}{\Gamma \vdash e_1 + e_2 : \text{num}}$$

$$\frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x:\tau. e : \tau \rightarrow \tau'} \quad \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

We have one more rule called the **subsumption** rule for subtyping:

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

Expressions  $e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e$

Types  $\tau ::= \text{num} \mid \tau \rightarrow \tau \mid \perp \mid \top$

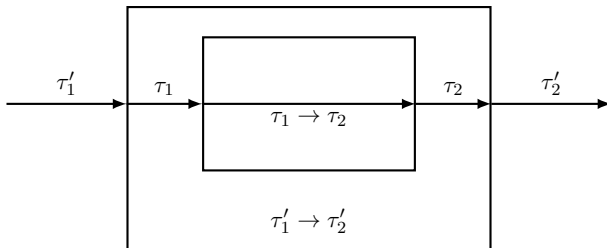
$\tau <: \tau$

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \qquad \frac{}{\perp <: \tau} \qquad \frac{}{\tau <: \top}$$

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

```
val x: Any = 42 // Int <: Any (Top)
def f(x: Nothing): String = x // Nothing (Bottom) <: String
```

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



```
val f: Any => String = (x: Any) => x.toString
val g: Int => Any = f // (Any => String) <: (Int => Any)
```

In general, a type constructor can be:

- **Covariant** if it preserves the subtype relation.
- **Contravariant** if it reverses the subtype relation.
- **Invariant** if it does not preserve or reverse the subtype relation.

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

The subtype relation for function types is called:

- **contravariant** in the parameter type and
- **covariant** in the return type.

In Scala, we can define type parameters with variance annotations:

```
class X[+A] // Covariant type parameter
class Y[-A] // Contravariant type parameter
class Z[A]  // Invariant type parameter

val intX: X[Int] = new X[Int]
val botX: X[Nothing] = intX // Error: X[Int] !<: X[Nothing]
val anyX: X[Any] = intX // OK : X[Int] <: X[Any]

val intY: Y[Int] = new Y[Int]
val botY: Y[Nothing] = intY // OK : Y[Int] <: Y[Nothing]
val anyY: Y[Any] = intY // Error: Y[Int] !<: Y[Any]

val intZ: Z[Int] = new Z[Int]
val botZ: Z[Nothing] = intZ // Error: Z[Int] !<: Z[Nothing]
val anyZ: Z[Any] = intZ // Error: Z[Int] !<: Z[Any]
```

In Scala language, **immutable** List is a **covariant** type constructor, while **mutable** Array is an **invariant** type constructor. Why?

$$\frac{\tau_1 <: \tau_2}{\text{List}[\tau_1] <: \text{List}[\tau_2]} \qquad \frac{}{\text{Array}[\tau] <: \text{Array}[\tau]}$$

The covariance of List is safe:

```
val intList: List[Int] = List(1, 2, 3)
val anyList: List[Any] = intList           // List[Int] <: List[Any]
val x: Int = intList.head                 // 1
x + 1                                     // OK
```

If we assume that Array is covariant, it would be unsafe:

```
val intArray: Array[Int] = Array(1, 2, 3)
val anyArray: Array[Any] = intArray       // Array[Int] <: Array[Any]
anyArray(0) = "Hello"                    // String <: Any
val y: Int = intArray(0)                  // "Hello" is not an Int!
y + 1                                     // Runtime error: "Hello" + 1
```

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

The subsumption rule is **non-syntax-directed** because  $\tau'$  is not determined by the syntax of  $e$ .

Let's define an **algorithmic** typing rule without the subsumption rule:

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

$$\frac{}{\Gamma \vdash n : \text{num}} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

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

$$\frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau'' \quad \tau'' <: \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

## 1. Subtyping

## 2. STLC with Subtyping

Typing Rules

Subtype Relation

Variance

Algorithmic Typing Rules

## 3. Other Extensions

Exceptions

Records

Conditional Expressions

## 4. Bounded Quantification

Expressions  $e ::= \dots \mid \text{raise}$

$$\overline{E[\text{raise}] \rightarrow \text{raise}}$$

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

$$\overline{\Gamma \vdash \text{raise} : \perp}$$

```
def f: Int = throw new Error // Nothing <: Int
def g: Boolean = throw new Error // Nothing <: Boolean
def h: String = throw new Error // Nothing <: String
```

Expressions  $e ::= \dots \mid \{x_1=e_1, \dots, x_n=e_n\} \mid e.x$

Values  $v ::= \dots \mid \{x_1=v_1, \dots, x_n=v_n\}$

Types  $\tau ::= \dots \mid \{x_1 : \tau_1, \dots, x_n : \tau_n\}$

$$\boxed{e \rightarrow e}$$

$$\frac{}{\{x_1=v_1, \dots, x_n=v_n\}.x_i \rightarrow v_i}$$

$E ::= \dots \mid \{x_1=E, \dots, x_n=e_n\} \mid \dots \mid \{x_1=v_1, \dots, x_n=E\} \mid E.x$

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

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

$$\frac{\Gamma \vdash e : \{\dots, x_i : \tau_i, \dots\}}{\Gamma \vdash e.x_i : \tau_i}$$

Types  $\tau ::= \dots \mid \{x_1 : \tau_1, \dots, x_n : \tau_n\}$

$$\tau <: \tau$$

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

$$\frac{}{\{x_1 : \tau_1, \dots, x_n : \tau_n, x_{n+1} : \tau_{n+1}\} <: \{x_1 : \tau_1, \dots, x_n : \tau_n\}}$$

$$\frac{\{k_1, \dots, k_n\} \text{ is a permutation of } \{1, \dots, n\}}{\{x_1 : \tau_1, \dots, x_n : \tau_n\} <: \{x_{k_1} : \tau_{k_1}, \dots, x_{k_n} : \tau_{k_n}\}}$$

Expressions  $e ::= \dots \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e$

Values  $v ::= \dots \mid \text{true} \mid \text{false}$

Types  $\tau ::= \dots \mid \text{bool}$

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

We can use the following rule with the subsumption rule:

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$
$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

Then, how to define algorithmic typing rules without the subsumption rule?

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_2 \sqcup \tau_3}$$

where  $\tau \sqcup \tau'$  (or  $\tau \sqcap \tau'$ ) is the least upper bound (or greatest lower bound) of  $\tau$  and  $\tau'$  with respect to the subtype relation:

$$\tau \sqcup \tau' = \begin{cases} \tau' & \text{if } \tau <: \tau' \\ \tau & \text{if } \tau >: \tau' \\ (\tau_1 \sqcap \tau'_1) \rightarrow (\tau_2 \sqcup \tau'_2) & \text{if } \tau = \tau_1 \rightarrow \tau_2 \text{ and } \tau' = \tau'_1 \rightarrow \tau'_2 \\ \top & \text{otherwise} \end{cases}$$

$$\tau \sqcap \tau' = \begin{cases} \tau & \text{if } \tau <: \tau' \\ \tau' & \text{if } \tau >: \tau' \\ (\tau_1 \sqcup \tau'_1) \rightarrow (\tau_2 \sqcap \tau'_2) & \text{if } \tau = \tau_1 \rightarrow \tau_2 \text{ and } \tau' = \tau'_1 \rightarrow \tau'_2 \\ \perp & \text{otherwise} \end{cases}$$

Scala supports union types ( $A \mid B$ ) and intersection types ( $A \& B$ ) between types  $A$  and  $B$ .

Using union and intersection types, Scala supports a more precise type for the conditional expression:

```
if (true) (x: Int) => x else (y: String) => y
// (Int & String) => (Int | String)
```

Without union and intersection types, the type of the above expression would be `Nothing => Any`, which is less precise than `(Int & String) => (Int | String)`.

## 1. Subtyping

## 2. STLC with Subtyping

Typing Rules

Subtype Relation

Variance

Algorithmic Typing Rules

## 3. Other Extensions

Exceptions

Records

Conditional Expressions

## 4. Bounded Quantification

We can extend the **system**  $F$  with subtyping, called **kernel**  $F_{<}$ , by adding **bounded quantification**:

Expressions  $e ::= n \mid x \mid e + e \mid \lambda x:\tau. e \mid e e \mid \Lambda\alpha <:\tau. e \mid e[\tau]$

Types  $\tau ::= \text{num} \mid \tau \rightarrow \tau \mid \alpha \mid \forall\alpha <:\tau. \tau$

$$\frac{\boxed{\Gamma \vdash e : \tau} \quad \alpha \notin \text{dom}(\Gamma) \quad \Gamma \vdash \tau \quad \Gamma[\alpha <:\tau] \vdash e : \tau'}{\Gamma \vdash \Lambda\alpha <:\tau. e : \forall\alpha <:\tau. \tau'}$$

$$\frac{\Gamma \vdash e : \forall\alpha <:\tau'. \tau'' \quad \Gamma \vdash \tau <:\tau'}{\Gamma \vdash e[\tau] : \tau''[\alpha \mapsto \tau]}$$

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

We need to define **subtype relation** with type environment  $\Gamma$ :

Types  $\tau ::= \text{num} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha <: \tau. \tau$

$$\begin{array}{c}
 \boxed{\Gamma \vdash \tau <: \tau} \\
 \\
 \frac{}{\Gamma \vdash \tau <: \tau} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 <: \tau_3} \\
 \\
 \frac{}{\Gamma \vdash \perp <: \tau} \qquad \frac{}{\Gamma \vdash \tau <: \top} \qquad \frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma \vdash \tau'_1 <: \tau'_2}{\Gamma \vdash \tau_1 \rightarrow \tau'_1 <: \tau_2 \rightarrow \tau'_2} \\
 \\
 \frac{(\alpha <: \tau) \in \Gamma}{\Gamma \vdash \alpha <: \tau} \qquad \frac{\Gamma[\alpha <: \tau] \vdash \tau_1 <: \tau_2}{\Gamma \vdash (\forall \alpha <: \tau. \tau_1) <: (\forall \alpha <: \tau. \tau_2)}
 \end{array}$$

The **kernel**  $F_{<}$ : does not allow the bound of the type variable to be different in the subtyping rule for universal types:

$$\frac{\Gamma[\alpha <: \tau] \vdash \tau_1 <: \tau_2}{\Gamma \vdash (\forall \alpha <: \tau. \tau_1) <: (\forall \alpha <: \tau. \tau_2)}$$

For example, we cannot derive the following subtyping relation in kernel  $F_{<}$ : because the bounds of  $\alpha$  are different (`num` and `⊤`):

$$\vdash (\forall \alpha <: \top. \alpha) <: (\forall \alpha <: \text{num}. \alpha)$$

We can support **full**  $F_{<}$ : by allowing different bounds in the subtyping rule for universal types:

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma[\alpha <: \tau_2] \vdash \tau'_1 <: \tau'_2}{\Gamma \vdash (\forall \alpha <: \tau_1. \tau'_1) <: (\forall \alpha <: \tau_2. \tau'_2)}$$

Note that the bound position (i.e.,  $\tau_1$  and  $\tau_2$ ) is **contravariant**, while the body position (i.e.,  $\tau'_1$  and  $\tau'_2$ ) is **covariant**.

Now, we can derive the following subtyping relation in full  $F_{<}$ :

$$\frac{\vdash \mathbf{num} <: \top \quad [\alpha <: \mathbf{num}] \vdash \alpha <: \alpha}{\vdash (\forall \alpha <: \top. \alpha) <: (\forall \alpha <: \mathbf{num}. \alpha)}$$

However, since full  $F_{<}$ : is **undecidable** but kernel  $F_{<}$ : is **decidable**, kernel  $F_{<}$ : is used in practice.

- Existential Types

Jihyeok Park  
jihyeok\_park@korea.ac.kr  
<https://plrg.korea.ac.kr>