# Lecture 20 – Type Operators

## AAA551: Programming Language Theory

Jihyeok Park


2026 Spring

# Recall

- Existential Types
  - Syntax and Dynamic Semantics
  - Well-Formedness and Typing Rules
  - Type Equivalence
  - Examples (Counter ADT)
  - Type Erasure
- Subtyping for Existential Types
- Encoding Existential Types via Universal Types

# Contents

# Contents

# Motivation

So far, every **type abstraction** we have seen ($\forall\alpha.\tau$, $\exists\alpha.\tau$) abstracts over a **proper type** $\alpha$ – a type that classifies values.

In Scala, **type constructors** that take types and produce types:

```
List[Int]          Map[String, Int]          Option[Bool]
```

Here `List`, `Map`, and `Option` are **not** types themselves – they are **functions on types**:

$$\begin{aligned} \texttt{List} &: \text{Type} \to \text{Type} \\ \texttt{Map} &: \text{Type} \to \text{Type} \to \text{Type} \\ \texttt{Option} &: \text{Type} \to \text{Type} \end{aligned}$$

**Idea:** introduce **type-level functions**, with their own notion of application and reduction.

# Type Operators

A **type operator** is a function from types to types. For example:

$$\begin{aligned} \texttt{Id} &\triangleq \lambda\alpha.\ \alpha \\ \texttt{KNum} &\triangleq \lambda\alpha.\ \texttt{num} \\ \texttt{Pair} &\triangleq \lambda\alpha.\ \lambda\beta.\ \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma \end{aligned}$$

Just as types classify **values**, we need **kinds** – the "types of types" – to classify type operators:

$$\texttt{Id} :: * \to * \qquad \texttt{KNum} :: * \to * \qquad \texttt{Pair} :: * \to * \to *$$

# Kinds

A **kind** $K$ classifies type expressions:

$$\begin{aligned} \text{Kinds} \quad K ::= {} & * \quad \text{(kind of proper types)} \\ \mid {} & K \to K \quad \text{(kind of type operators)} \end{aligned}$$

Read $K_1 \to K_2$ as: "type operator that takes a type of kind $K_1$ and produces a type of kind $K_2$".

Examples:

| | | |
|---|---|---|
| `num`, `bool`, `num` $\to$ `num` | :: | $*$ |
| `List`, `Option`, `Id`, `KNum` | :: | $* \to *$ |
| `Pair`, `Map`, `Either` | :: | $* \to * \to *$ |
| `Container` (later in this lecture) | :: | $(* \to *) \to *$ |

$\Gamma \vdash e : \tau$ makes sense only when the type $\tau$ is a **proper type** of kind $*$.

# Kinding Judgement

We replace the well-formedness judgement $\Gamma \vdash \tau$ with the more general **kinding judgement**:

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

read as "in environment $\Gamma$, type $\tau$ has kind $K$".

The type environment $\Gamma$ now binds each type variable to its **kind**:

$$\Gamma ::= \varnothing \mid \Gamma[\alpha :: K] \mid \Gamma[x : \tau]$$

Previously we wrote $\Gamma[\alpha]$ for "$\alpha$ is a proper type"; this is now shorthand for $\Gamma[\alpha :: *]$.

# Syntax of System $F_\omega$

We extend System F with **type-level abstraction** and **type-level application**:

$$
\begin{array}{lll}
\text{Expressions} & e & ::= n \mid x \mid e + e \mid \lambda x{:}\tau.\ e \mid e\ e \mid \Lambda\alpha :: K\ .\ e \mid e[\tau] \\
\text{Types} & \tau & ::= \texttt{num} \mid \tau \to \tau \mid \alpha \mid \forall\alpha :: K\ .\tau \\
 & & \quad \mid \quad \lambda\alpha :: K.\tau \quad \mid \quad \tau[\tau] \\
\text{Kinds} & K & ::= \quad * \mid K \to K
\end{array}
$$

Two new type forms:

- $\lambda\alpha :: K.\tau$ – **type-level abstraction**
- $\tau_1[\tau_2]$ – **type-level application**

The **term-level** syntax and dynamic semantics are essentially unchanged: all new structure lives at the **type level**. $F_\omega$ is sometimes described as "STLC on top of System F".

# Kinding Rules

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

$$\frac{}{\Gamma \vdash \texttt{num} :: *} \qquad \frac{\Gamma \vdash \tau_1 :: * \qquad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \tau_1 \to \tau_2 :: *} \qquad \frac{(\alpha :: K) \in \Gamma}{\Gamma \vdash \alpha :: K}$$

$$\frac{\Gamma[\alpha :: K] \vdash \tau :: *}{\Gamma \vdash \forall \alpha :: K.\tau :: *}$$

$$\frac{\Gamma[\alpha :: K_1] \vdash \tau :: K_2}{\Gamma \vdash \lambda \alpha :: K_1.\tau :: K_1 \to K_2} \qquad \frac{\Gamma \vdash \tau_1 :: K_1 \to K_2 \qquad \Gamma \vdash \tau_2 :: K_1}{\Gamma \vdash \tau_1[\tau_2] :: K_2}$$

The last two rules are the **STLC typing rules**, lifted to the type level. $\lambda$ at the type level is the introduction form for $K_1 \to K_2$, and type application is the elimination form.

# Type Equivalence

With type-level $\lambda$ and application, two syntactically different types may denote the same thing:

$$(\lambda\alpha :: *.\alpha \to \alpha)[\texttt{num}] \quad \text{and} \quad \texttt{num} \to \texttt{num}$$

We extend the equivalence relation with a **type-level $\beta$ rule**:

$$\boxed{\tau \equiv \tau} \qquad \frac{}{(\lambda\alpha :: K.\tau)[\tau'] \equiv \tau[\alpha \mapsto \tau']}$$

Plus the usual congruence rules so $\equiv$ remains a congruence under every type constructor:

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1[\tau_2] \equiv \tau_1'[\tau_2']} \qquad \frac{\tau \equiv \tau'}{\lambda\alpha :: K.\tau \equiv \lambda\alpha :: K.\tau'} \qquad \cdots$$

Type-level reduction is **strongly normalizing** (it is just STLC). So $\equiv$ is **decidable**: reduce both sides to normal form and compare syntactically (up to $\alpha$-equivalence).

# Typing Rules

Most typing rules are unchanged from System F. The two essential upgrades are:

- replace well-formedness $\Gamma \vdash \tau$ with kinding $\Gamma \vdash \tau :: *$;
- use $\equiv$ (which now includes type-level $\beta$) wherever types are compared.

$$\frac{\Gamma \vdash \tau_1 :: * \qquad \Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_3 \qquad \tau_1 \equiv \tau_3}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma[\alpha :: K\ ] \vdash e : \tau}{\Gamma \vdash \Lambda\alpha :: K\ .\ e : \forall\alpha :: K\ .\tau} \qquad \frac{\Gamma \vdash e : \forall\alpha :: K.\tau' \qquad \Gamma \vdash \tau :: K}{\Gamma \vdash e[\tau] : \tau'[\alpha \mapsto \tau]}$$

Notice how the kind annotation on $\forall$ **controls what kind of type** can instantiate it – so we can now write, e.g., $\forall F :: * \to *. \ldots$.

# Contents

# Type Lambdas

PLRG

Scala 3 supports **type-level abstraction** directly via **type lambdas**:

```
type Id   = [X] =>> X                          // identity operator
type KInt = [X] =>> Int                        // constant Int
type Pair = [X, Y] =>> (X, Y)                  // binary product
type Copy = [X] =>> (X, X)                     // copying operator
type Comp[F[_], G[_]] = [X] =>> F[G[X]]        // composition
```

The syntax `[X] =>> ...` corresponds directly to $\lambda\alpha :: *. \ldots$, and `[X, Y] =>> ...` to a curried type-level $\lambda$.

```
val n: Id[Int]       = 42
val m: KInt[Bool]    = 42
val p: Pair[Int, String] = (1, "two")
// Comp[Copy, Copy][Int] = Copy[Copy[Int]] = ((Int, Int), (Int, Int))
val c: Comp[Copy, Copy][Int] = ((1, 2), (3, 4))
```

# Higher-Kinded Type Parameters

The kind $(* \to *) \to *$ corresponds to a **higher-kinded type parameter**, written `F[_]` in Scala. For example, a generic `Container` abstraction:

```
trait Container[F[_]]:                                  // F :: * -> *
  def empty[A]: F[A]
  def size[A](fa: F[A]): Int
```

Here `Container[F[_]]` is parameterized by a **type operator** $F :: * \to *$.

```
given Container[List] with
  def empty[A]: List[A] = Nil
  def size[A](xs: List[A]): Int = xs.length

given Container[Option] with
  def empty[A]: Option[A] = None
  def size[A](o: Option[A]): Int = if o.isEmpty then 0 else 1
```

# Using a Higher-Kinded Type Class

We use a `Container` instance through Scala 3's **context parameter** (`using` clause):

```
def isEmpty[F[_], A](fa: F[A])(using C: Container[F]): Boolean =
  C.size(fa) == 0

isEmpty(List(1, 2, 3))           // false  (Container[List]   used)
isEmpty(Option.empty[Int])       // true   (Container[Option] used)
isEmpty(Option(true))            // false  (Container[Option] used)
```

The compiler **infers** `F` from the argument (e.g., `F = List`) and then **synthesizes** the matching `Container[F]` instance.

Equivalent shorthand using a **context bound** `F[_]: Container`:

```
def isEmpty[F[_]: Container, A](fa: F[A]): Boolean =
  summon[Container[F]].size(fa) == 0
```

# HKT and Type Lambdas Together

What if we want a `Container` instance for `Either[String, _]`? `Either` has kind $* \to * \to *$, but `Container` wants something of kind $* \to *$.

Solution: **partially apply** `Either` via a type lambda to get something of kind $* \to *$:

```
given Container[[X] =>> Either[String, X]] with
  def empty[A]: Either[String, A] = Left("empty")
  def size[A](e: Either[String, A]): Int = if e.isRight then 1 else 0
```

In the formal system, this is exactly:

$$\texttt{Container}[\lambda\alpha :: *.\, \texttt{Either}[\texttt{String}][\alpha]]$$

– a type operator passed as an argument to another type operator.

# Contents

# System $F^{\omega}_{<:}$

Combining $F_{\omega}$ (type operators) with $F_{<:}$ (bounded quantification) gives $F^{\omega}_{<:}$. The subtype relation must be lifted to **operators of higher kinds**.

Subtyping is **kind-indexed**: $\Gamma \vdash \tau_1 <: \tau_2 :: K$ means $\tau_1$ and $\tau_2$ both have kind $K$, and $\tau_1$ is a subtype of $\tau_2$.

At an arrow kind, operators are compared **pointwise on the body**:

$$\frac{\Gamma[\alpha :: K_1] \vdash \tau_1 <: \tau_2 :: K_2}{\Gamma \vdash \lambda\alpha :: K_1.\tau_1 \;<:\; \lambda\alpha :: K_1.\tau_2 \;::\; K_1 \to K_2}$$

Subtyping is also compatible with type-level application:

$$\frac{\Gamma \vdash \tau_1 <: \tau_2 :: K_1 \to K_2 \qquad \Gamma \vdash \tau :: K_1}{\Gamma \vdash \tau_1[\tau] <: \tau_2[\tau] :: K_2}$$

The argument type $\tau$ should be the same (invariant) on both sides because we don't know what variance the operator has in its parameter.

# Variance Revisited

PLRG

For a unary type operator $F$, we say $F$ is

- **covariant** if $\tau_1 <: \tau_2 \implies F[\tau_1] <: F[\tau_2]$
- **contravariant** if $\tau_1 <: \tau_2 \implies F[\tau_2] <: F[\tau_1]$
- **invariant** otherwise

These are properties of how the operator's **body** uses its parameter:

$$\lambda\alpha :: *.\texttt{List}[\alpha] \quad \text{(covariant)} \qquad \lambda\alpha :: *.\alpha \to \texttt{num} \quad \text{(contravariant)}$$

Scala turns variance into an **annotation** on the operator's parameter:

```
trait List[+A]                  // covariant:   List[Dog] <: List[Animal]
trait Function1[-A, +B]         // contravariant in A, covariant in B
trait Array[A]                  // invariant
```

In $F^{\omega}_{<:}$, variance is **derivable** from how $\alpha$ appears in the body; Scala's annotations are kind-level constraints the compiler checks against the body's structure.

# Next Lecture

- Curry-Howard Isomorphism

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