

Lecture 17 – Universal Types

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Algebraic data types
 - Product types
 - Sum types
 - Algebraic data types
- References and exceptions
 - References
 - Exceptions

1. Polymorphism

2. Universal Types

- Syntax and Dynamic Semantics

- Well-Formedness

- Type Equivalence

- Typing Rules

- Type Erasure

- Examples

- Normalization

1. Polymorphism

2. Universal Types

Syntax and Dynamic Semantics

Well-Formedness

Type Equivalence

Typing Rules

Type Erasure

Examples

Normalization

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 **parametric polymorphism** (also known as **universal types**).

1. Polymorphism

2. Universal Types

- Syntax and Dynamic Semantics

- Well-Formedness

- Type Equivalence

- Typing Rules

- Type Erasure

- Examples

- Normalization

Consider the following **twice** function for numbers:

$$\text{twiceNum} \triangleq \lambda f : \text{num} \rightarrow \text{num}. \lambda x : \text{num}. f (f (x))$$

However, if we want to apply this function to other types (e.g., bools, functions), we need to define separate functions for each type:

$$\text{twiceBool} \triangleq \lambda f : \text{bool} \rightarrow \text{bool}. \lambda x : \text{bool}. f (f (x))$$

$$\begin{aligned} \text{twiceNumFun} &\triangleq \lambda f : (\text{num} \rightarrow \text{num}) \rightarrow (\text{num} \rightarrow \text{num}). \\ &\quad \lambda x : (\text{num} \rightarrow \text{num}). f (f (x)) \end{aligned}$$

⋮

Let's introduce a **type variable** α and **instantiate** it with different types.

Let's extend STLC with universal types (also known as **System F**):

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

Values $v ::= \lambda x:\tau. e \mid n \mid \Lambda\alpha. e$

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

$$\frac{n = n_1 + n_2}{n_1 + n_2 \rightarrow n}$$

$$\frac{}{(\lambda x:\tau. e)(v) \rightarrow e[x \mapsto v]}$$

$$\frac{e_1 \rightarrow e_2}{E[e_1] \rightarrow E[e_2]}$$

$$\frac{}{(\Lambda\alpha. e)[\tau] \rightarrow e[\alpha \mapsto \tau]}$$

$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E \mid E[\tau]$

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

First, we need to check the **well-formedness** of types:

$\Gamma \vdash \tau$

$$\frac{}{\Gamma \vdash \text{num}} \quad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \rightarrow \tau_2} \quad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \quad \frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash \forall\alpha.\tau}$$

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

Two types often have different syntax but the same meaning. For example,

$$\forall \alpha. \alpha \rightarrow \alpha \quad \equiv \quad \forall \beta. \beta \rightarrow \beta$$

Let's define the **type equivalence** relation \equiv :

$$\boxed{\tau \equiv \tau}$$

$$\frac{}{\text{num} \equiv \text{num}} \qquad \frac{\tau_1 \equiv \tau'_1 \quad \tau_2 \equiv \tau'_2}{\tau_1 \rightarrow \tau_2 \equiv \tau'_1 \rightarrow \tau'_2} \qquad \frac{}{\alpha \equiv \alpha} \qquad \frac{\tau \equiv \tau'[\alpha \mapsto \beta]}{\forall \alpha. \tau \equiv \forall \beta. \tau'}$$

Now, let's define the typing rules:

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

$$\frac{}{\Gamma \vdash n : \mathbf{num}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma[x : \tau] \vdash e : \tau' \quad \Gamma \vdash \tau}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_3 \quad \tau_1 \equiv \tau_3}{\Gamma \vdash e_1 e_2 : \tau_2}$$

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

The type information in a program is only used for type checking, and does not affect the runtime behavior of the program.

Let's define the **type erasure** function $\text{erase}(e)$ to remove all type annotations from an expression e :

$$\text{erase}(n) = n$$

$$\text{erase}(x) = x$$

$$\text{erase}(e_1 + e_2) = \text{erase}(e_1) + \text{erase}(e_2)$$

$$\text{erase}(\lambda x:\tau. e) = \lambda x. \text{erase}(e)$$

$$\text{erase}(e_1 e_2) = \text{erase}(e_1) \text{erase}(e_2)$$

$$\text{erase}(\Lambda\alpha. e) = \text{erase}(e)$$

$$\text{erase}(e[\tau]) = \text{erase}(e)$$

Example: Twice Function

We can define the **twice** function as follows:

$$\text{twice} \triangleq \Lambda\alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. f (f (x))$$

We can use this function for different types:

$$\text{twice}[\text{num}](\lambda x : \text{num}.x + 1)(2) \rightarrow^* 4$$

$$\text{twice}[\text{bool}](\lambda x : \text{bool}.\neg x)(\text{true}) \rightarrow^* \text{true}$$

$$\text{twice}[\text{num} \rightarrow \text{num}](\text{twice}[\text{num}])(\lambda x : \text{num}.x + 1)(2) \rightarrow^* 6$$

We can encode **product types** using universal types as follows:

$$\tau_1 \times \tau_2 \triangleq \forall \alpha. (\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha$$

$$\text{pair} \triangleq \Lambda \alpha. \Lambda \beta. \lambda(x : \alpha). \lambda(y : \beta). \Lambda \gamma. \lambda(f : \alpha \rightarrow \beta \rightarrow \gamma). f \ x \ y$$

$$\text{fst} \triangleq \Lambda \alpha. \Lambda \beta. \lambda(p : \tau_1 \times \tau_2). p[\alpha](\lambda(x : \alpha). \lambda(y : \beta). x)$$

$$\text{snd} \triangleq \Lambda \alpha. \Lambda \beta. \lambda(p : \tau_1 \times \tau_2). p[\beta](\lambda(x : \alpha). \lambda(y : \beta). y)$$

For example, we can use this encoding as follows:

```
let p = pair[num][num → num] 2 (λ(x : num). x + 1) in
let f = fst[num][num → num] p in
let n = snd[num][num → num] p in
f n
```

will evaluate to $2 + 1 = 3$.

We can also encode **sum types** using universal types as follows:

$$\tau_1 + \tau_2 \triangleq \forall \alpha. (\tau_1 \rightarrow \alpha) \rightarrow (\tau_2 \rightarrow \alpha) \rightarrow \alpha$$

$$\text{inl} \triangleq \Lambda \alpha. \Lambda \beta. \lambda(x : \alpha). \Lambda \gamma. \lambda(f : \alpha \rightarrow \gamma). \lambda(g : \beta \rightarrow \gamma). f \ x$$

$$\text{inr} \triangleq \Lambda \alpha. \Lambda \beta. \lambda(y : \beta). \Lambda \gamma. \lambda(f : \alpha \rightarrow \gamma). \lambda(g : \beta \rightarrow \gamma). g \ y$$

$$\text{case} \triangleq \Lambda \alpha. \Lambda \beta. \Lambda \gamma. \lambda(s : \tau_1 + \tau_2). \lambda(f : \alpha \rightarrow \gamma). \lambda(g : \beta \rightarrow \gamma). s[\gamma] \ f \ g$$

For example, we can use this encoding as follows:

```
let s = inl[num][num → num] 2 in
let l = (λ(x : num). x + 1) in
let r = (λ(f : num → num). f 3) in
case[num][num → num][num] s l r
```

will evaluate to $2 + 1 = 3$.

Without polymorphism, we cannot provide a type for the following self-application function:

$$\mathbf{self} \triangleq \lambda x. x x$$

On the other hand, we can define its typed version using universal types:

$$\mathbf{self} \triangleq \lambda x : \forall \alpha. \alpha \rightarrow \alpha. x [\forall \alpha. \alpha \rightarrow \alpha] x$$

However, it is still impossible to define a typed version of the following function fixed-point combinator:

$$\mathbf{fix} \triangleq \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$$

A term is in **normal form** if it cannot be reduced any further.

For example, the following terms are in normal form:

$$n, \quad \lambda x : \tau. e, \quad \Lambda \alpha. e$$

There are two types of normalization properties:

- **Weak normalization:** Every term can be reduced to a normal form.
- **Strong normalization:** Every reduction sequence of a term is finite.

System F is **strongly normalizing**, which means that every well-typed term can be reduced to a normal form in a finite number of steps. Thus,

System F cannot represent non-terminating computations (e.g., `fix`).

- Existential types

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