Lecture 17 – Universal Types

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

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

Contents

Contents

Polymorphism

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.

Polymorphism

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**).

Contents

Motivation

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

$$\texttt{twiceNum} \triangleq \lambda \texttt{f} : \texttt{num} \to \texttt{num}.\ \lambda \texttt{x} : \texttt{num}.\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))$$

Motivation

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

$$\texttt{twiceNum} \triangleq \lambda \texttt{f} : \texttt{num} \to \texttt{num}.\ \lambda \texttt{x} : \texttt{num}.\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))$$

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

$$\texttt{twiceBool} \triangleq \lambda \texttt{f} : \texttt{bool} \to \texttt{bool}.\ \lambda \texttt{x} : \texttt{bool}.\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))$$

$$\begin{aligned}\texttt{twiceNumFun} \triangleq\ & \lambda \texttt{f} : (\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num}). \\ & \lambda \texttt{x} : (\texttt{num} \to \texttt{num}).\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))\end{aligned}$$

$$\vdots$$

Motivation

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

$$\texttt{twiceNum} \triangleq \lambda \texttt{f} : \texttt{num} \to \texttt{num}.\ \lambda \texttt{x} : \texttt{num}.\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))$$

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

$$\texttt{twiceBool} \triangleq \lambda \texttt{f} : \texttt{bool} \to \texttt{bool}.\ \lambda \texttt{x} : \texttt{bool}.\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))$$

$$\begin{aligned}\texttt{twiceNumFun} \triangleq\ & \lambda \texttt{f} : (\texttt{num} \to \texttt{num}) \to (\texttt{num} \to \texttt{num}). \\ & \lambda \texttt{x} : (\texttt{num} \to \texttt{num}).\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))\end{aligned}$$

$$\vdots$$

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

Universal Types – Syntax and Dynamic Semantics

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

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

$$\text{Values} \quad v ::= \lambda x{:}\tau.\ e \mid n \mid \Lambda\alpha.\ e$$

Universal Types – Syntax and Dynamic Semantics

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

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

$$\text{Values} \quad v ::= \lambda x{:}\tau.\ e \mid n \mid \Lambda\alpha.\ e$$

$$\boxed{e \to e}$$

$$\frac{n = n_1 + n_2}{n_1 + n_2 \to n} \qquad \frac{}{(\lambda x{:}\tau.\ e)(v) \to e[x \mapsto v]} \qquad \frac{e_1 \to e_2}{E[e_1] \to E[e_2]}$$

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

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

Universal Types – Well-Formedness

PLRG

$$\text{Types} \quad \tau ::= \texttt{num} \mid \tau \to \tau \mid \alpha \mid \forall \alpha.\tau$$

Universal Types – Well-Formedness

PLRG

$$\text{Types} \quad \tau ::= \texttt{num} \mid \tau \to \tau \mid \alpha \mid \forall \alpha.\tau$$

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

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

$$\frac{}{\Gamma \vdash \texttt{num}} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash \forall \alpha.\tau}$$

Universal Types – Type Equivalence

PLRG

$$\text{Types} \quad \tau ::= \texttt{num} \mid \tau \to \tau \mid \alpha \mid \forall \alpha.\tau$$

Universal Types – Type Equivalence

PLRG

$$\text{Types} \quad \tau ::= \texttt{num} \mid \tau \to \tau \mid \alpha \mid \forall \alpha.\tau$$

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

$$\forall \alpha.\alpha \to \alpha \qquad \equiv \qquad \forall \beta.\beta \to \beta$$

Universal Types – Type Equivalence

PLRG

$$\text{Types} \quad \tau ::= \texttt{num} \mid \tau \to \tau \mid \alpha \mid \forall \alpha.\tau$$

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

$$\forall \alpha.\alpha \to \alpha \qquad \equiv \qquad \forall \beta.\beta \to \beta$$

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

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

$$\frac{}{\texttt{num} \equiv \texttt{num}} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{}{\alpha \equiv \alpha} \qquad \frac{\tau \equiv \tau'[\alpha \mapsto \beta]}{\forall \alpha.\tau \equiv \forall \beta.\tau'}$$

Universal Types – Typing Rules

Now, let's define the typing rules:

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

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

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

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

Type Erasure

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

Type Erasure

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 $\texttt{erase}(e)$ to remove all type annotations from an expression e:

$$
\begin{aligned}
\texttt{erase}(n) &= n \\
\texttt{erase}(x) &= x \\
\texttt{erase}(e_1 \texttt{ + } e_2) &= \texttt{erase}(e_1) \texttt{ + } \texttt{erase}(e_2) \\
\texttt{erase}(\lambda x\texttt{:}\tau.\ e) &= \lambda x.\ \texttt{erase}(e) \\
\texttt{erase}(e_1\ e_2) &= \texttt{erase}(e_1)\ \texttt{erase}(e_2) \\
\texttt{erase}(\Lambda\alpha.\ e) &= \texttt{erase}(e) \\
\texttt{erase}(e[\tau]) &= \texttt{erase}(e)
\end{aligned}
$$

Example: Twice Function

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

$$\texttt{twice} \triangleq \Lambda\alpha.\ \lambda\texttt{f} : \alpha \to \alpha.\ \lambda\texttt{x} : \alpha.\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))$$

Example: Twice Function

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

$$\texttt{twice} \triangleq \Lambda\alpha.\ \lambda\texttt{f} : \alpha \to \alpha.\ \lambda\texttt{x} : \alpha.\ \texttt{f}\ (\texttt{f}\ (\texttt{x}))$$

We can use this function for different types:

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

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

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

Example: Product Types

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

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

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

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

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

Example: Product Types

PLRG

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

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

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

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

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

For example, we can use this encoding as follows:

$$\begin{array}{l}\texttt{let p = pair}[\texttt{num}][\texttt{num}\to\texttt{num}]\ 2\ (\lambda(x:\texttt{num}).\ x\ \texttt{+}\ 1)\ \texttt{in}\\ \texttt{let f = fst}[\texttt{num}][\texttt{num}\to\texttt{num}]\ \texttt{p in}\\ \texttt{let n = snd}[\texttt{num}][\texttt{num}\to\texttt{num}]\ \texttt{p in}\\ \texttt{f n}\end{array}$$

will evaluate to $2 + 1 = 3$.

Example: Sum Types

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

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

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

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

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

Example: Sum Types

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

$$\begin{aligned}
\tau_1 + \tau_2 &\triangleq \forall\alpha.(\tau_1 \to \alpha) \to (\tau_2 \to \alpha) \to \alpha \\
\texttt{inl} &\triangleq \Lambda\alpha.\ \Lambda\beta.\ \lambda(x : \alpha).\ \Lambda\gamma.\ \lambda(f : \alpha \to \gamma).\ \lambda(g : \beta \to \gamma).\ f\ x \\
\texttt{inr} &\triangleq \Lambda\alpha.\ \Lambda\beta.\ \lambda(y : \beta).\ \Lambda\gamma.\ \lambda(f : \alpha \to \gamma).\ \lambda(g : \beta \to \gamma).\ g\ y \\
\texttt{case} &\triangleq \Lambda\alpha.\ \Lambda\beta.\ \Lambda\gamma.\ \lambda(s : \tau_1 + \tau_2).\ \lambda(f : \alpha \to \gamma).\ \lambda(g : \beta \to \gamma).\ s[\gamma]\ f\ g
\end{aligned}$$

For example, we can use this encoding as follows:

$$\begin{aligned}
&\texttt{let s = inl}[\texttt{num}][\texttt{num} \to \texttt{num}]\ 2\ \texttt{in} \\
&\texttt{let l} = (\lambda(x : \texttt{num}).\ x\ \texttt{+}\ 1)\ \texttt{in} \\
&\texttt{let r} = (\lambda(f : \texttt{num} \to \texttt{num}).\ f\ 3)\ \texttt{in} \\
&\texttt{case}[\texttt{num}][\texttt{num} \to \texttt{num}][\texttt{num}]\ \texttt{s l r}
\end{aligned}$$

will evaluate to $2 + 1 = 3$.

Example: Self Application

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

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

Example: Self Application

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

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

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

$$\texttt{self} \triangleq \lambda x : \forall\alpha.\alpha \to \alpha.\ x[\forall\alpha.\alpha \to \alpha]\ x$$

Example: Self Application

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

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

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

$$\texttt{self} \triangleq \lambda x : \forall\alpha.\alpha \to \alpha.\ x[\forall\alpha.\alpha \to \alpha]\ x$$

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

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

Normalization

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

For example, the following terms are in normal form:

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

Normalization

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.

Normalization

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`).

Next Lecture

- Existential types

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