# Lecture 23 – Parametric Polymorphism

## COSE212: Programming Languages

Jihyeok Park


2023 Fall

# Recall

- An **algebraic data type** is a **recursive sum type of product types**.
- **ATFAE** – TRFAE with **ADTs** and **pattern matching**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**

# Recall

- An **algebraic data type** is a **recursive sum type of product types**.
- **ATFAE** – TRFAE with **ADTs** and **pattern matching**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**
- In this lecture, we will learn **parametric polymorphism**.

# Recall

- An **algebraic data type** is a **recursive sum type of product types**.
- **ATFAE** – TRFAE with **ADTs** and **pattern matching**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**
- In this lecture, we will learn **parametric polymorphism**.
- **PTFAE** – TFAE with **parametric polymorphism**.
  - **Interpreter** and **Natural Semantics**
  - **Type Checker** and **Typing Rules**

# Contents

# Contents

# Polymorphism

In the following Scala program, `f` is an identity function and we want to pass ❶ `1`, ❷ `true`, and ❸ `(y: Int) => y` to `f`, respectively.

```
def f(x: ???): ??? = x;
f(1); f(true); f((y: Int) => y)
```

# Polymorphism

In the following Scala program, `f` is an identity function and we want to pass ❶ `1`, ❷ `true`, and ❸ `(y: Int) => y` to `f`, respectively.

```
def f(x: ???): ??? = x;
f(1); f(true); f((y: Int) => y)
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be ❶ `Int`, ❶ `Boolean`, and ❸ `Int => Int`, simultaneously.

# Polymorphism

In the following Scala program, `f` is an identity function and we want to pass ❶ `1`, ❷ `true`, and ❸ `(y: Int) => y` to `f`, respectively.

```
def f(x: ???): ??? = x;
f(1); f(true); f((y: Int) => y)
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be ❶ `Int`, ❶ `Boolean`, and ❸ `Int => Int`, simultaneously.

How can we resolve this problem?

# Polymorphism

In the following Scala program, `f` is an identity function and we want to pass ❶ `1`, ❷ `true`, and ❸ `(y: Int) => y` to `f`, respectively.

```
def f(x: ???): ??? = x;
f(1); f(true); f((y: Int) => y)
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be ❶ `Int`, ❶ `Boolean`, and ❸ `Int => Int`, simultaneously.

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

# Polymorphism

PLRG

In the following Scala program, `f` is an identity function and we want to pass ❶ `1`, ❷ `true`, and ❸ `(y: Int) => y` to `f`, respectively.

```
def f(x: ???): ??? = x;
f(1); f(true); f((y: Int) => y)
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be ❶ `Int`, ❶ `Boolean`, and ❸ `Int => Int`, simultaneously.

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

**Polymorphism** is to use a single entity as **multiple types**, and there are various kinds of polymorphism:

- **Parametric polymorphism**
- **Subtype polymorphism**
- **Ad-hoc polymorphism**
- ...

# Polymorphism

In the following Scala program, `f` is an identity function and we want to pass ❶ `1`, ❷ `true`, and ❸ `(y: Int) => y` to `f`, respectively.

```
def f(x: ???): ??? = x;
f(1); f(true); f((y: Int) => y)
```

Unfortunately, we cannot assign any type to `x` because the type of `x` should be ❶ `Int`, ❶ `Boolean`, and ❸ `Int => Int`, simultaneously.

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

**Polymorphism** is to use a single entity as **multiple types**, and there are various kinds of polymorphism:

- **Parametric polymorphism**
- **Subtype polymorphism**
- **Ad-hoc polymorphism**
- ...

Among them, let's learn **parametric polymorphism** in this lecture.

# Parametric Polymorphism

PLRG

## Definition (Parametric Polymorphism)

**Parametric polymorphism** is a form of polymorphism by introducing **type variables** and instantiating them with **type arguments**.

# Parametric Polymorphism

PLRG

## Definition (Parametric Polymorphism)

**Parametric polymorphism** is a form of polymorphism by introducing **type variables** and instantiating them with **type arguments**.

```
def f[T](x: T): T = x;
f[Int](1); f[Boolean](true); f[Int => Int]((y: Int) => y)
```

The type `T` is a **type variable** (or **type parameter**), and it can be **instantiated** to any types (e.g., `Int`, `Boolean`, and `Int => Int`) by passing them as **type arguments**.

# Parametric Polymorphism

PLRG

## Definition (Parametric Polymorphism)

**Parametric polymorphism** is a form of polymorphism by introducing **type variables** and instantiating them with **type arguments**.

```
def f[T](x: T): T = x;
f[Int](1); f[Boolean](true); f[Int => Int]((y: Int) => y)
```

The type `T` is a **type variable** (or **type parameter**), and it can be **instantiated** to any types (e.g., `Int`, `Boolean`, and `Int => Int`) by passing them as **type arguments**.

In general, parametric polymorphism is applied to **functions** and **data types**, and they are sometimes called **generic functions** and **generic data types**, respectively.

# Parametric Polymorphism

Many modern typed languages support **parametric polymorphism**:

- Scala

```
def f[T](x: T): T = x
```

- C++

```
template <typename T> T f(T x) { return x; }
```

- Rust

```
fn f<T>(x: T) -> T { return x; }
```

- Haskell

```
f :: a -> a
f x = x
```

- ...

# Contents

# PTFAE – TFAE with Parametric Polymorphism

Now, let's extend TFAE into PTFAE to support **parametric polymorphism**.

```
/* PTFAE */
val f = forall[T] { (x: T) => x }          // [T](T => T)
val x = f[Number](42)                      // Number
val y = f[Number => Number](f[Number])     // Number => Number
val z = f[[T](T => T)](f)                  // [T](T => T)
...
```

`forall[t] e` parameterizes an expression `e` with a type variable `t`, and `e[t]` instantiates the type variable with a type `t` of an expression `e`.

# PTFAE – TFAE with Parametric Polymorphism

PLRG

Now, let's extend TFAE into PTFAE to support **parametric polymorphism**.

```
/* PTFAE */
val f = forall[T] { (x: T) => x }          // [T](T => T)
val x = f[Number](42)                      // Number
val y = f[Number => Number](f[Number])     // Number => Number
val z = f[[T](T => T)](f)                  // [T](T => T)
...
```

`forall[t] e` parameterizes an expression `e` with a type variable `t`, and `e[t]` instantiates the type variable with a type `t` of an expression `e`.

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

1. **type abstraction** (`forall`)
2. **type application**
3. **polymorphic type**

# Concrete Syntax

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

1. **type abstraction** (`forall`)
2. **type application**
3. **polymorphic type**

# Concrete Syntax

PLRG

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

1. **type abstraction** (`forall`)
2. **type application**
3. **polymorphic type**

We can extend the **concrete syntax** of TFAE as follows:

```
// expressions
<expr> ::= ...
         | "forall" "[" <id> "]" <expr>
         | <expr> "[" <type> "]"

// types
<type> ::= ...
         | <id>
         | "[" <id> "]" <type>
```

# Abstract Syntax

PLRG

| | | | |
|---|---|---|---|
| Expressions | $\mathbb{E} \ni e$ | $::= \ldots$ | |
| | | $\mid \forall\alpha.e$ | (TypeAbs) |
| | | $\mid e[\tau]$ | (TypeApp) |
| Types | $\mathbb{T} \ni \tau$ | $::= \ldots$ | |
| | | $\mid \alpha$ | (VarT) |
| | | $\mid \forall\alpha.\tau$ | (PolyT) |
| Type Variables | $\alpha$ | $\in \mathbb{X}_\alpha$ | (String) |

```
enum Expr:
  ...
  case TypeAbs(name: String, body: Expr)
  case TypeApp(expr: Expr, ty: Type)

enum Type:
  ...
  case VarT(name: String)
  case PolyT(name: String, ty: Type)
```

# Contents

# Interpreter and Natural Semantics

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

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

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

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

# Interpreter and Natural Semantics

PLRG

For PTFAE, 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 **type abstraction values**:

$$\begin{array}{rlll} \text{Values} & \mathbb{V} \ni v ::= & n & (\texttt{NumV}) \\ & & \mid \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) \\ & & \mid \langle \forall \alpha.e, \sigma \rangle & (\texttt{TypeAbsV}) \end{array}$$

```
enum Value:
  case NumV(number: BigInt)
  case CloV(param: String, body: Expr, env: Env)
  case TypeAbsV(name: String, body: Expr, env: Env)
```

# Type Abstraction

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

  case TypeAbs(name, body) => TypeAbsV(name, body, env)

  case TypeApp(expr, ty) => interp(expr, env) match
    case TypeAbsV(name, body, fenv) => interp(body, fenv)
    case v => error(s"not a type abstraction: ${v.str}")
```

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

$$\textsc{TypeAbs}\ \frac{}{\sigma \vdash \forall\alpha.e \Rightarrow \langle \forall\alpha.e, \sigma \rangle}$$

$$\textsc{TypeApp}\ \frac{\sigma \vdash e \Rightarrow \langle \forall\alpha.e', \sigma' \rangle \qquad \sigma' \vdash e' \Rightarrow v}{\sigma \vdash e[\tau] \Rightarrow v}$$

# Contents

# Type Checker and Typing Rules

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

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

and ❷ 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** $\Gamma$ as a mapping from variable names to their types.

$$\text{Type Environments} \quad \Gamma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T} \quad (\texttt{TypeEnv})$$

```
type TypeEnv = Map[String, Type]
```

# Type Environment for Type Variables

PLRG

However, we need additional information in type environments to keep track of which **type variables** are defined by **type abstractions**.

$$\text{Type Environments} \quad \Gamma \in (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}) \times \mathcal{P}(\mathbb{X}_\alpha) \quad (\texttt{TypeEnv})$$

$\Gamma[\alpha]$ is an extension of $\Gamma$ with the type variable $\alpha$ defined.

# Type Environment for Type Variables

PLRG

However, we need additional information in type environments to keep track of which **type variables** are defined by **type abstractions**.

$$\text{Type Environments} \quad \Gamma \in (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{T}) \times \mathcal{P}(\mathbb{X}_\alpha) \quad (\texttt{TypeEnv})$$

$\Gamma[\alpha]$ is an extension of $\Gamma$ with the type variable $\alpha$ defined.

```
case class TypeEnv(
  vars: Map[String, Type] = Map(),
  tys: Set[String] = Set(),
) {
  def addVar(pair: (String, Type)): TypeEnv =
    TypeEnv(vars + pair, tys)
  def addVars(pairs: Iterable[(String, Type)]): TypeEnv =
    TypeEnv(vars ++ pairs, tys)
  def addType(name: String): TypeEnv = TypeEnv(vars, tys + name)
}
```

# Well-Formedness of Types

PLRG

Similar to ATFAE, we need to check the **well-formedness** of types with **type environment** to prevent the use of not-defined type variables.

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

$$\frac{}{\Gamma \vdash \texttt{num}} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau'}{\Gamma \vdash \tau \to \tau'} \qquad \frac{\alpha \in \mathsf{Domain}(\Gamma)}{\Gamma \vdash \alpha} \qquad \frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash \forall \alpha.\tau}$$

```
def mustValid(ty: Type, tenv: TypeEnv): Type = ty match
  case NumT =>
    NumT
  case ArrowT(pty, rty) =>
    ArrowT(mustValid(pty, tenv), mustValid(rty, tenv))
  case VarT(name) =>
    if (!tenv.tys.contains(name)) error(s"unknown type: $name")
    VarT(name)
  case PolyT(name, ty) =>
    PolyT(name, mustValid(ty, tenv.addType(name)))
```

# Function Definition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Fun(param, paramTy, body) =>
    mustValid(paramTy, tenv)
    ArrowT(paramTy, typeCheck(body, tenv.addVar(param -> paramTy)))
```

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

$$\tau\text{–Fun}\ \frac{\Gamma \vdash \tau \qquad \Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'}$$

Similar to ATFAE, we check the **well-formedness** of parameter types.

# Type Abstraction

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeAbs(name, body) =>
    PolyT(name, typeCheck(body, tenv.addType(name)))
```

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

$$\tau\text{-TypeAbs}\ \frac{\Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \forall\alpha.e : \forall\alpha.\tau}$$

# Type Abstraction

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeAbs(name, body) =>
    PolyT(name, typeCheck(body, tenv.addType(name)))
```

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

$$\tau\text{-TypeAbs}\ \frac{\Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \forall\alpha.e : \forall\alpha.\tau}$$

It is indeed **type unsound**, and we will fix it later in this lecture.

# Type Application

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeApp(expr, ty) => typeCheck(expr, tenv) match
    case PolyT(name, bodyTy) => subst(bodyTy, name, mustValid(ty, tenv))
    case t => error(s"not a polymorphic type: ${t.str}")
```

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

$$\tau\text{-}\mathtt{TypeApp}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \forall\alpha.\tau'}{\Gamma \vdash e[\tau] : \tau'[\alpha \leftarrow \tau]}$$

We also need to check the **well-formedness** of type arguments.

$\tau'[\alpha \leftarrow \tau]$ means replacing all occurrences of **free type variable** $\alpha$ in $\tau'$ with $\tau$. For example,

$$(\alpha \to \beta \to (\forall\alpha.\alpha) \to \alpha)[\alpha \leftarrow \mathtt{num}] \quad = \quad \mathtt{num} \to \beta \to (\forall\alpha.\alpha) \to \mathtt{num}$$

# Type Application – Substitution

We can implement the substitution as follows:

```
def subst(bodyTy: Type, name: String, ty: Type): Type = bodyTy match
  case NumT =>
    NumT
  case ArrowT(pty, rty) =>
    ArrowT(subst(pty, name, ty), subst(rty, name, ty))
  case VarT(x) =>
    if (name == x) ty
    else VarT(x)
  case PolyT(x, bodyTy) =>
    if (name == x) PolyT(x, bodyTy)
    else PolyT(x, subst(bodyTy, name, ty))
```

Now, we can instantiate type variables with given types in specific types:

```
val ty = Type("T => U => ([T](T)) => T")
subst(ty, "T", NumT).str        // "Number => U => ([T](T)) => Number"
```

# Function Application

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case App(fun, arg) => typeCheck(fun, tenv) match
    case ArrowT(paramTy, retTy) =>
      mustSame(typeCheck(arg, tenv), paramTy)
      retTy
    case t => error(s"not a function type: ${t.str}")
```

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

$$\tau\text{-App}\ \frac{\Gamma \vdash e_0 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0(e_1) : \tau_2}$$

# Function Application

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case App(fun, arg) => typeCheck(fun, tenv) match
    case ArrowT(paramTy, retTy) =>
      mustSame(typeCheck(arg, tenv), paramTy)
      retTy
    case t => error(s"not a function type: ${t.str}")
```

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

$$\tau\text{-App}\ \frac{\Gamma \vdash e_0 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0(e_1) : \tau_2}$$

While we can use the same rule in TFAE, but we can improve it.

```
/* PTFAE */
val f = ((x: [T](T => T)) => x[Number](7)) // ([T](T => T)) => Number
val x = forall[U] (x: U) => x              // [U] (U => U)
f(x)                  // type checking failed: [T](T => T) != [U](U => U)
```

# Function Application – Equivalence of Types

PLRG

Let's define the equivalence ($\equiv$) of types as follows:

```
def isSame(lty: Type, rty: Type): Boolean = (lty, rty) match
  case (NumT, NumT) => true
  case (ArrowT(lpty, lrty), ArrowT(rpty, rrty)) =>
    isSame(lpty, rpty) && isSame(lrty, rrty)
  case (VarT(lname), VarT(rname)) => lname == rname
  case (PolyT(lname, lty), PolyT(rname, rty)) =>
    isSame(lty, subst(rty, rname, VarT(lname)))
  case _ => false

def mustSame(l: Type, r: Type): Unit =
  if (!isSame(l, r)) error(s"type mismatch: ${l.str} != ${r.str}")
```

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

$$\frac{}{\mathtt{num} \equiv \mathtt{num}} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{(\tau_1 \to \tau_2) \equiv (\tau_1' \to \tau_2')} \qquad \frac{}{\alpha \equiv \alpha} \qquad \frac{\tau \equiv \tau'[\alpha' \leftarrow \alpha]}{\forall\alpha.\tau \equiv \forall\alpha'.\tau'}$$

# Function Application – Revised

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case App(fun, arg) => typeCheck(fun, tenv) match
    case ArrowT(paramTy, retTy) =>
      mustSame(typeCheck(arg, tenv), paramTy)
      retTy
    case t => error(s"not a function type: ${t.str}")
```

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

$$\tau\text{-App}\ \frac{\Gamma \vdash e_0 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 : \tau_3 \qquad \tau_1 \equiv \tau_3}{\Gamma \vdash e_0(e_1) : \tau_2}$$

While we can use the same rule in TFAE, but we can improve it.

```
/* PTFAE */
val f = ((x: [T](T => T)) => x[Number](7)) // ([T](T => T)) => Number
val x = forall[U] (x: U) => x              // [U] (U => U)
f(x)                                       // well-typed: [T](T => T) == [U](U => U)
```

# Contents

# Recall: Type Soundness

PLRG

## Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

# Recall: Type Soundness

PLRG

## Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

```
/* PTFAE */
val f = forall[U] (x: U) => {
  val y = forall[U] x                    // [U](U)
  y[Number => Number]                    // Number => Number
}                                        // [U](U => (Number => Number))
val g = f[Number](1)                     // Number => Number
g(2)                                     // Number
```

# Recall: Type Soundness

PLRG

## Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

```
/* PTFAE */
val f = forall[U] (x: U) => {
  val y = forall[U] x                    // [U](U)
  y[Number => Number]                    // Number => Number
}                                        // [U](U => (Number => Number))
val g = f[Number](1)                     // Number => Number
g(2)                                     // Number
```

It throws a **type error** when evaluating `1(2)` at run-time while this expression is **well-typed** (i.e., **unsound type system**).

# Recall: Type Soundness

PLRG

## Definition (Type Soundness)

A **type system** is **sound** if it guarantees that a **well-typed** program will **never** cause a **type error** at run-time.

```
/* PTFAE */
val f = forall[U] (x: U) => {
  val y = forall[U] x              // [U](U)
  y[Number => Number]              // Number => Number
}                                  // [U](U => (Number => Number))
val g = f[Number](1)               // Number => Number
g(2)                               // Number
```

It throws a **type error** when evaluating 1(2) at run-time while this expression is **well-typed** (i.e., **unsound type system**).

We can resolve this problem by **forbidding** the redefinition of **same type variable** in the scope of **type abstractions**!

# Type Abstraction - Revised

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeAbs(name, body) =>
    if (tenv.tys.contains(name)) error(s"already defined type: $name")
    PolyT(name, typeCheck(body, tenv.addType(name)))
```

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

$$\tau\text{–TypeAbs}\ \frac{\alpha \notin \text{Domain}(\Gamma) \qquad \Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \forall\alpha.e : \forall\alpha.\tau}$$

# Summary

PLRG

1. Parametric Polymorphism
2. PTFAE – TFAE with Parametric Polymorphism
    - Concrete Syntax
    - Abstract Syntax
3. Interpreter and Natural Semantics for PTFAE
4. Type Checker and Typing Rules
    - Type Environment for Type Variables
    - Well-Formedness of Types
    - Function Definition
    - Type Abstraction
    - Type Application
    - Function Application
5. Type Soundness of PTFAE
    - Type Abstraction - Revised

# Exercise #13

- Please see this document[1] 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.

[1] https://github.com/ku-plrg-classroom/docs/tree/main/cose212/ptfae.

# Next Lecture

- Subtype Polymorphism

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