

Lecture 23 – Parametric Polymorphism

COSE212: Programming Languages

Jihyeok Park



2025 Fall

- 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**

- 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**.

- 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

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

Contents

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

Polymorphism

In the following Scala program, f is an identity function and we want to pass ① 1, ② `true`, and ③ `(y: Int) => y + 1` to f, respectively.

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

Polymorphism

In the following Scala program, f is an identity function and we want to pass ① 1, ② `true`, and ③ `(y: Int) => y` to f, respectively.

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

Unfortunately, we cannot assign any type to x because the type of x should be ① `Int`, ② `Boolean`, and ③ `Int => Int`, simultaneously.

Polymorphism

In the following Scala program, f is an identity function and we want to pass ① 1, ② `true`, and ③ `(y: Int) => y` to f, respectively.

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

Unfortunately, we cannot assign any type to x because the type of x should be ① Int, ② Boolean, and ③ `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 ① 1, ② `true`, and ③ `(y: Int) => y` to f, respectively.

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

Unfortunately, we cannot assign any type to x because the type of x should be ① Int, ② Boolean, and ③ `Int => Int`, simultaneously.

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

Polymorphism

In the following Scala program, f is an identity function and we want to pass ① 1, ② `true`, and ③ `(y: Int) => y` to f, respectively.

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

Unfortunately, we cannot assign any type to x because the type of x should be ① Int, ② Boolean, and ③ `Int => Int`, simultaneously.

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

Polymorphism is to use a single entity as **multiple types**.

Polymorphism

In the following Scala program, f is an identity function and we want to pass ① 1, ② `true`, and ③ `(y: Int) => y` to f, respectively.

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

Unfortunately, we cannot assign any type to x because the type of x should be ① Int, ② Boolean, and ③ `Int => Int`, simultaneously.

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

Polymorphism is to use a single entity as **multiple types**.

There are various kinds of polymorphism:

- **Parametric polymorphism**
- **Subtype polymorphism**
- **Ad-hoc polymorphism** (overloading (Java), type classes (Haskell))
- ...

Polymorphism

In the following Scala program, f is an identity function and we want to pass ① 1, ② `true`, and ③ `(y: Int) => y` to f, respectively.

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

Unfortunately, we cannot assign any type to x because the type of x should be ① Int, ② Boolean, and ③ `Int => Int`, simultaneously.

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

Polymorphism is to use a single entity as **multiple types**.

There are various kinds of polymorphism:

- **Parametric polymorphism**
- **Subtype polymorphism**
- **Ad-hoc polymorphism** (overloading (Java), type classes (Haskell))
- ...

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

Definition (Parametric Polymorphism)

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

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**.

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

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

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`.

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**

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>
```

Expressions

$\mathbb{E} \ni e ::= \dots$

| $\forall \alpha.e$ (TypeAbs)
| $e[\tau]$ (TypeApp)

Types

$\mathbb{T} \ni \tau ::= \dots$

| α (VarT)
| $\forall \alpha.\tau$ (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

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

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:

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

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:

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

with a new kind of values called **type abstraction values** to **delay** the evaluation of type abstractions until they are applied to type arguments.

$$\begin{array}{ll} \text{Values} & \forall \exists v ::= n \quad (\text{NumV}) \\ & | \langle \lambda x.e, \sigma \rangle \quad (\text{CloV}) \\ & | \langle \forall \alpha.e, \sigma \rangle \quad (\text{TypeAbsV}) \end{array}$$

```
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}")
```

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

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

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

Contents

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

Type Checker and Typing Rules

Let's ① design **typing rules** of PTFAE to define when an expression is well-typed in the form of:

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

and ② 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** Γ as a mapping from variable names to their types.

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

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

Type Environment for Type Variables

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

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

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

Type Environment for Type Variables

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

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

$\Gamma[\alpha]$ is an extension of Γ with the type variable α 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

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

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

$$\frac{}{\Gamma \vdash \text{num}} \quad \frac{\Gamma \vdash \tau \quad \Gamma \vdash \tau'}{\Gamma \vdash \tau \rightarrow \tau'} \quad \frac{\alpha \in \text{Domain}(\Gamma)}{\Gamma \vdash \alpha} \quad \frac{\Gamma[\alpha] \vdash \tau}{\Gamma \vdash \forall \alpha. \tau}$$

```
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)))
```

```
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)))
```

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

$$\tau\text{-Fun} \quad \frac{\Gamma \vdash \tau \quad \Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'}$$

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) =>
    ???
```

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

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

Type Abstraction

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

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

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

First, we need to check the **body** of a type abstraction.

Type Abstraction

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

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

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

We need to extend the **type environment** with the type variable α .

Type Abstraction

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

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

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

The type of a type abstraction is a **polymorphic type**.

Type Abstraction

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

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

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

The type of a type abstraction is a **polymorphic type**.

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) => ???
```

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

$$\frac{\textcolor{red}{???}}{\tau\text{-TypeApp} \quad \frac{}{\Gamma \vdash e[\tau] : \textcolor{red}{???}}}$$

Type Application

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case TypeApp(expr, ty) => typeCheck(expr, tenv); ???
```

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

$$\tau\text{-TypeApp} \frac{\Gamma \vdash e : ???}{\Gamma \vdash e[\tau] : ???}$$

First, we need to check the type of e with the given type environment Γ .

Type Application

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

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

$$\tau\text{-TypeApp} \quad \frac{\Gamma \vdash e : \forall \alpha. \tau' \quad ???}{\Gamma \vdash e[\tau] : ???}$$

But, we need to allow type application only if the type of e is a **polymorphic type**.

Type Application

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

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

$$\tau\text{-TypeApp} \quad \frac{\Gamma \vdash e : \forall \alpha. \tau' \quad \Gamma \vdash \tau}{\Gamma \vdash e[\tau] : ???}$$

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

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}")
```

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

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

Finally, we need to **substitute** the type variable α with the type argument τ in the body type τ' .

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}")
```

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

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

Finally, we need to **substitute** the type variable α with the type argument τ in the body type τ' .

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

$$(\alpha \rightarrow \beta \rightarrow (\forall \alpha. \alpha) \rightarrow \alpha)[\alpha \leftarrow \text{num}] = \text{num} \rightarrow \beta \rightarrow (\forall \alpha. \alpha) \rightarrow \text{num}$$

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}")
```

$$\frac{\Gamma \vdash e_0 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0(e_1) : \tau_2}$$

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}")
```

$$\frac{\Gamma \vdash e_0 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0(e_1) : \tau_2}$$

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

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}")
```

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

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

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}")
```

$$\frac{\Gamma \vdash e_0 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_3 \quad \tau_1 \equiv \tau_3}{\Gamma \vdash e_0(e_1) : \tau_2}$$

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

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

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.

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
}
val g = f[Number](1)            // Number => Number
g(2)                          // Number
```

Recall: Type Soundness

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
}
val g = f[Number](1)           // [U] (U => (Number => Number))
g(2)                          // Number
```

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

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
}
val g = f[Number](1)           // Number => Number
g(2)                          // Number
```

It throws a **type error** when evaluating `g(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)))
```

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

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

Summary

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 #14

<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/ptfae>

- Please see above document 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.

<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/mini-scala>

- Please see above document on GitHub:
 - Implement `typeCheck` function.
 - Implement `interp` functions.
- The due date is 23:59 on Dec. 10 (Wed.).
- Please only submit `Implementation.scala` file to [LMS](#).
- The given test cases are **not sufficient**. Please write your own test cases to test your implementation. We will use **randomly generated** test cases for grading.

Next Lecture

- Subtype Polymorphism

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