Lecture 19 – Typed Languages

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

- Safe Language Systems
  - Dynamic vs Static Analysis for Detecting Run-Time Errors
  - Soundness vs Completeness of Analysis
- Type Systems
  - Types
  - Type Errors
  - Type Checking
  - Type Soundness
- In this lecture, we will define our first **typed language**.
- **TFAE** – FAE with **type system**.
  - **Type Checking** and **Typing Rules**
  - Interpreter and Natural Semantics

Contents

1. TFAE – FAE with Type System
    - Concrete Syntax
    - Abstract Syntax
2. Type Checking and Typing Rules
    - Type Checking
    - Typing Rules
3. Interpreter and Natural Semantics

Contents

1. TFAE – FAE with Type System
   - Concrete Syntax
   - Abstract Syntax
2. Type Checking and Typing Rules
   - Type Checking
   - Typing Rules
3. Interpreter and Natural Semantics

TFAE – FAE with Type System

Before defining TFAE, consider the types of the following FAE expressions:

```
/* FAE */ 42
```

Since it produces a **number**, let's say it is a `Number` type expression.

```
/* FAE */ x => x + 1
```

It produces a function value, but can we say more about its type? **Yes!**

It should take a **number** type argument and return a **number**.

Let's say it is a `Number => Number` type expression.

```
/* FAE */ x => x
```

There is no information on the parameter x.

To handle such cases, let's explicitly add **type annotations** to the parameters of function definitions.

TFAE – FAE with Type System

Let's extend FAE into TFAE to support **type system** by adding **type annotations** to function definitions:

```
/* TFAE */
(x: Number) => x                    // x is `Number` type
(f: Number => Number) => f(42)      // f is `Number => Number` type
```

If we define immutable variable definitions as **syntactic sugar**, it requires the type annotations:

```
/* TFAE */
val x: Number = 42; x + 1           // == `((x: Number) => x + 1)(42)`
```

However, we we can infer variable types from their initial values if we **explicitly define** them rather than syntactic sugar:

```
/* TFAE */
val x = 42; x + 1                   // x is `Number` type because of `42`
```

Concrete Syntax

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

1. **function definitions** with **type annotations**
2. **immutable variable definitions** without **type annotations**
3. **types**

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

```
// expressions
<expr> ::= ...
         | "(" <id> ":" <type> ")" "=>" <expr>
         | "val" <id> "=" <expr> ";" <expr>

// types
<type> ::= "Number"                  // number type
         | <type> "=>" <type>        // arrow type
```

Since functions are first-class values, the parameter and return types could be recursively arrow types.

Abstract Syntax

We can extend the **abstract syntax** of FAE for TFAE as follows:

$$
\begin{array}{llll}
\text{Expressions} & \mathbb{E} \ni e ::= & \ldots & \\
 & & \mid \lambda x{:}\tau.e & (\texttt{Fun}) \\
 & & \mid \texttt{val}\ x\texttt{=}e;\ e & (\texttt{Val})
\end{array}
$$

$$
\begin{array}{llll}
\text{Types} & \mathbb{T} \ni \tau ::= & \texttt{Number} & (\texttt{NumT}) \\
 & & \mid \tau \rightarrow \tau & (\texttt{ArrowT})
\end{array}
$$

We can define the abstract syntax of TFAE in Scala as follows:

```
enum Expr:
  ...
  case Fun(param: String, ty: Type, body: Expr)
  case Val(name: String, init: Expr, body: Expr)

enum Type:
  case NumT
  case ArrowT(paramTy: Type, bodyTy: Type)
```

Contents

1. TFAE – FAE with Type System
   - Concrete Syntax
   - Abstract Syntax

2. Type Checking and Typing Rules
   - Type Checking
   - Typing Rules

3. Interpreter and Natural Semantics

Type Checking

If the following conditions hold, we say **"the expression e has type τ"**:

- e does not cause any type error, and
- e evaluates to a value of type τ or does not terminate.

If so, we use the following notation and say that e is **well-typed**:

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

It is defined by **typing rules** and implemented as the follows in Scala:

```scala
def typeCheck(expr: Expr): Type = ???
```

Type Checking

If the following conditions hold, we say **“the expression *e* has type τ”**:

- *e* does not cause any type error, and
- *e* evaluates to a value of type τ or does not terminate.

If so, we use the following notation and say that *e* is **well-typed**:

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

It is defined by **typing rules** and implemented as the follows in Scala:

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = ???
```

We need a **type environment** Γ to keep track of the variable types in *e*:

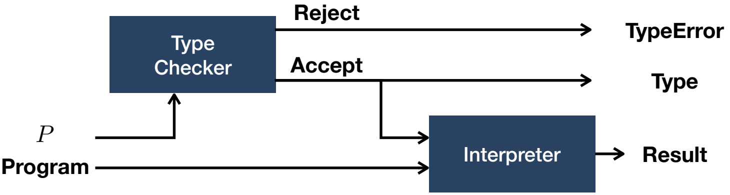
$$\text{Type Environments} \qquad \Gamma \;\in\; \mathbb{X} \xrightarrow{\text{fin}} \mathbb{T} \quad (\texttt{TypeEnv})$$

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

Type Checking

Definition (Type Checking)

Type checking is a kind of static analysis checking whether a given expression e is **well-typed**. A **type checker** returns the **type** of e if it is well-typed, or rejects it and reports the detected **type error** otherwise.




```scala
def eval(str: String): String =
  val expr = Expr(str)
  val ty = typeCheck(expr, Map.empty)
  val v = interp(expr, Map.empty)
  s"${v.str}: ${ty.str}"
```

Typing Rules

Now, lets' define the typing rules for TFAE in the following form:

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

and fill out the body of the `typeCheck` function:

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = ???
```

with type environments:

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

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

Typing Rules – Numbers

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  case Num(_) => NumT
  ...
```

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

$$\tau\text{–}\texttt{Num}\ \frac{}{\Gamma \vdash n : \texttt{Number}}$$

The number literal n has `Number` type in any type environment Γ.

Typing Rules – Addition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Add(left, right) =>
    ???
```

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

$$\tau\texttt{-Add}\ \frac{???}{\Gamma \vdash e_1 + e_2 : ???}$$

Typing Rules – Addition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Add(left, right) =>
    typeCheck(left, tenv)
    ???
```

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

$$\tau\text{--}\mathtt{Add}\ \frac{\Gamma \vdash e_1 : \tau \qquad ???}{\Gamma \vdash e_1 + e_2 : ???}$$

Type checker should do

1. get the type of e_1 in Γ

Typing Rules – Addition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Add(left, right) =>
    mustSame(typeCheck(left, tenv), NumT)
    ???


def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type error: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{--}\mathtt{Add}\ \frac{\Gamma \vdash e_1 : \mathtt{Number} \qquad \mathtt{???}}{\Gamma \vdash e_1 + e_2 : \mathtt{???}}$$

Type checker should do

1. check the type of e_1 is $\mathtt{Number}$ in Γ

Typing Rules – Addition

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Add(left, right) =>
    mustSame(typeCheck(left, tenv), NumT)
    mustSame(typeCheck(right, tenv), NumT)
    ???

def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type error: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{–}\mathtt{Add}\ \frac{\Gamma \vdash e_1 : \mathtt{Number} \qquad \Gamma \vdash e_2 : \mathtt{Number}}{\Gamma \vdash e_1 + e_2 : \color{red}{???}}$$

Type checker should do

1. check the types of e_1 and e_2 are `Number` in Γ

Typing Rules – Addition

```
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Add(left, right) =>
    mustSame(typeCheck(left, tenv), NumT)
    mustSame(typeCheck(right, tenv), NumT)
    NumT

def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type error: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{–}\mathtt{Add}\ \frac{\Gamma \vdash e_1 : \mathtt{Number} \qquad \Gamma \vdash e_2 : \mathtt{Number}}{\Gamma \vdash e_1 + e_2 : \mathtt{Number}}$$

Type checker should do

1. check the types of e_1 and e_2 are `Number` in Γ
2. return `Number` as the type of $e_1 + e_2$

Typing Rules – Multiplication

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Mul(left, right) =>
    mustSame(typeCheck(left, tenv), NumT)
    mustSame(typeCheck(right, tenv), NumT)
    NumT

def mustSame(lty: Type, rty: Type): Unit =
  if (lty != rty) error(s"type error: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{–}\texttt{Mul}\ \frac{\Gamma \vdash e_1 : \texttt{Number} \qquad \Gamma \vdash e_2 : \texttt{Number}}{\Gamma \vdash e_1 \times e_2 : \texttt{Number}}$$

Typing Rules – Immutable Variable Definition

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Val(x, init, body) =>
    val initTy = typeCheck(init, tenv)
    typeCheck(body, tenv + (x -> initTy))
```

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

$$\tau\text{–}\texttt{Val}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{val}\ x\texttt{=}e_1\texttt{;}\ e_2 : \tau_2}$$

This rule stores the type of x in Γ inferred from the initial value.

```
/* TFAE */ val x = 1; x + 2      // `x: Number` in `tenv` <- `1: Number`
```

Typing Rules – Identifier Lookup

```scala
def typeCheck(expr: Expr, tenv: TypeEnv): Type = expr match
  ...
  case Id(x) =>
    tenv.getOrElse(x, error(s"free identifier: $x"))
```

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

$$\tau\text{–}\mathtt{Id}\ \frac{x \in \mathsf{Domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

This rule looks up the type of x in Γ.

```
/* TFAE */ val x = 1; x + 2      // `x: Number` in `tenv` <- `1: Number`
```

Typing Rules – Function Definition

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

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

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

We can check the body of a function with the its parameter type.

```
/* TFAE */ (x: Number) => x      // Number => Number
```

Typing Rules – Function Application

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

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

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

We don't have to check the type of the function body because it is already checked when the function is defined.

```
/* TFAE */ ((x: Number) => x)(1) // Number
```

Contents

1. TFAE – FAE with Type System
    - Concrete Syntax
    - Abstract Syntax
2. Type Checking and Typing Rules
    - Type Checking
    - Typing Rules
3. Interpreter and Natural Semantics

Interpreter and Natural Semantics

For interpreter and natural semantics for TFAE, it is just enough to extend the those for function definitions in FAE.

```
def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Fun(p, t, b) => CloV(p, b, env)
```

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

$$\texttt{Fun}\ \frac{}{\sigma \vdash \lambda x{:}\tau.e \Rightarrow \langle \lambda x.e, \sigma \rangle}$$

The type annotation is ignored in the interpreter and natural semantics.

Summary

1. TFAE – FAE with Type System
   - Concrete Syntax
   - Abstract Syntax

2. Type Checking and Typing Rules
   - Type Checking
   - Typing Rules

3. Interpreter and Natural Semantics

Exercise #10

- 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/tfae.

Next Lecture

- Typing Recursive Functions

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