Lecture 19 – Typed Languages

COSE212: Programming Languages

Jihyeok Park

2024 Fall

Recall

PLRG

- 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

Recall

PLRG

- 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**.

Recall

PLRG

- 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 Checker** and **Typing Rules**
  - Interpreter and Natural Semantics

Contents

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

2. Type Checker and Typing Rules
   - Recall: Type Checking
   - Type Environment
   - Numbers
   - Addition and Multiplication
   - Immutable Variable Definition and Identifier Lookup
   - Function Definition and Application
   - Examples

3. Interpreter with Type Checker
   - Interpreter and Natural Semantics
   - Operational Semantics vs Typing Rules

Contents

PLRG

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

2. Type Checker and Typing Rules
 Recall: Type Checking
 Type Environment
 Numbers
 Addition and Multiplication
 Immutable Variable Definition and Identifier Lookup
 Function Definition and Application
 Examples

3. Interpreter with Type Checker
 Interpreter and Natural Semantics
 Operational Semantics vs Typing Rules

TFAE – FAE with Type System

PLRG

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

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

TFAE – FAE with Type System

PLRG

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

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

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

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

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

TFAE – FAE with Type System

PLRG

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

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

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

```
/* 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**.

TFAE – FAE with Type System

PLRG

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

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

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

```
/* 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 its type is `Number => Number` called **arrow type**.

TFAE – FAE with Type System

PLRG

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

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

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

```
/* 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 its type is `Number => Number` called **arrow type**.

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

TFAE – FAE with Type System

PLRG

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

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

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

```
/* 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 its type is `Number => Number` called **arrow type**.

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

How about this?

TFAE – FAE with Type System

PLRG

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

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

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

```
/* 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 its type is `Number => Number` called **arrow type**.

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

How about this? There is no information on the parameter x.

TFAE – FAE with Type System

PLRG

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

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

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

```
/* 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 its type is `Number => Number` called **arrow type**.

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

How about this? There is no information on the parameter x.

One simple solution is to explicitly add **type annotations**!

TFAE – FAE with Type System

PLRG

Let's extend FAE into TFAE with **type annotations** to specify the types of function parameters:

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

TFAE – FAE with Type System

PLRG

Let's extend FAE into TFAE with **type annotations** to specify the types of function parameters:

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

If we define immutable variable definitions as **syntactic sugar**, it requires the type annotations: $\mathcal{D}[\![\texttt{val}\ x\texttt{:}\tau\ \texttt{=}\ e\texttt{;}\ e']\!] = (\lambda x\texttt{:}\tau.\mathcal{D}[\![e']\!])(\mathcal{D}[\![e]\!])$

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

TFAE – FAE with Type System

PLRG

Let's extend FAE into TFAE with **type annotations** to specify the types of function parameters:

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

If we define immutable variable definitions as **syntactic sugar**, it requires the type annotations: $\mathcal{D}[\![\texttt{val}\ x\texttt{:}\tau\ \texttt{=}\ e\texttt{;}\ e']\!] = (\lambda x\texttt{:}\tau.\mathcal{D}[\![e']\!])(\mathcal{D}[\![e]\!])$

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

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

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

Concrete Syntax

PLRG

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**

Concrete Syntax

PLRG

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> ::= "(" <type> ")"         // only for precedence
         | "Number"               // number type
         | <type> "=>" <type>     // arrow type
```

Concrete Syntax

PLRG

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> ::= "(" <type> ")"         // only for precedence
         | "Number"               // number type
         | <type> "=>" <type>     // arrow type
```

Since functions are first-class values, the parameter and return types could be recursively arrow types. And, => is **right-associative**.

Abstract Syntax

PLRG

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

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

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

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, retTy: Type)
```

Contents

PLRG

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

2. Type Checker and Typing Rules
   Recall: Type Checking
   Type Environment
   Numbers
   Addition and Multiplication
   Immutable Variable Definition and Identifier Lookup
   Function Definition and Application
   Examples

3. Interpreter with Type Checker
   Interpreter and Natural Semantics
   Operational Semantics vs Typing Rules

Recall: Type Checking

PLRG

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**:

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

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.

Recall: Type Checking

PLRG

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**:

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

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.

We need to

1. design **typing rules** to define when an expression is well-typed
2. implement a **type checker** in Scala according to typing rules

Type Environment

PLRG

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

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

Type Environment

PLRG

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

$$\vdash e : \tau$$

and ❷ implement a **type checker** in Scala according to typing rules:

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

The type checker returns the **type** of e if it is well-typed, or rejects it and throws a **type error** otherwise.

Type Environment

PLRG

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

$$\vdash e : \tau$$

and ➋ implement a **type checker** in Scala according to typing rules:

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

The type checker returns the **type** of e if it is well-typed, or rejects it and throws a **type error** otherwise.

In addition, we need to keep track of the **variable types**.

Type Environment

PLRG

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

$$\Gamma \vdash e : \tau$$

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.

In addition, we need to keep track of the **variable types**.

Let's define a **type environment** Γ as a mapping from variable names to their types and pass it to the type checker.

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

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

Numbers

PLRG

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

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

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

Numbers


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

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

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

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

Addition

PLRG

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

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

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

Addition

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

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

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

Type checker should do

1. get the type of e_1 in Γ

Addition

PLRG

```scala
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 mismatch: ${lty.str} != ${rty.str}")
```

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

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

Type checker should do

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

Addition

PLRG

```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 mismatch: ${lty.str} != ${rty.str}")
```

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

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

Type checker should do

1. check the types of e_1 and e_2 are $\mathtt{num}$ in Γ

Addition

PLRG

```scala
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 mismatch: ${lty.str} != ${rty.str}")
```

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

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

Type checker should do

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

Multiplication

PLRG

```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 mismatch: ${lty.str} != ${rty.str}")
```

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

$$\tau\text{–}\mathtt{Mul}\ \frac{\Gamma \vdash e_1 : \mathtt{num} \qquad \Gamma \vdash e_2 : \mathtt{num}}{\Gamma \vdash e_1\ \mathtt{*}\ e_2 : \mathtt{num}}$$

Type checker should do

1. check the types of e_1 and e_2 are `num` in Γ
2. return `num` as the type of e_1 `*` e_2

Immutable Variable Definition

PLRG

```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))
```

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

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

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

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

Identifier Lookup

PLRG

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

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

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

This rule looks up the type of x in Γ.

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

Function Definition

PLRG

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

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

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

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

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

Function Application

PLRG

```
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 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
```

Examples

```
/* TFAE */ val x = 1; x + 2                    // 3: Number
```

Examples

```
/* TFAE */ val x = 1; x + 2                    // 3: Number
```

$$\dfrac{\dfrac{}{\varnothing \vdash 1 : \texttt{num}} \quad \dfrac{\dfrac{x \in \textsf{Domain}([x : \texttt{num}])}{[x : \texttt{num}] \vdash x : \texttt{num}} \quad \dfrac{}{[x : \texttt{num}] \vdash 2 : \texttt{num}}}{[x : \texttt{num}] \vdash x \texttt{ + } 2 : \texttt{num}}}{\varnothing \vdash \texttt{val } x \texttt{ = } 1\texttt{; } x \texttt{ + } 2 : \texttt{num}}$$

Examples

```
/* TFAE */ val x = 1; x + 2                    // 3: Number
```

$$\dfrac{\dfrac{}{\varnothing \vdash 1 : \texttt{num}} \quad \dfrac{\dfrac{x \in \textsf{Domain}([x : \texttt{num}])}{[x : \texttt{num}] \vdash x : \texttt{num}} \quad \dfrac{}{[x : \texttt{num}] \vdash 2 : \texttt{num}}}{[x : \texttt{num}] \vdash x \texttt{ + } 2 : \texttt{num}}}{\varnothing \vdash \texttt{val } x \texttt{ = } 1\texttt{; } x \texttt{ + } 2 : \texttt{num}}$$

```
/* TFAE */ ((x: Number) => x)(2) * 3          // 6: Number
```

Examples

PLRG

```
/* TFAE */ val x = 1; x + 2                    // 3: Number
```

$$\dfrac{\dfrac{}{\varnothing \vdash 1 : \texttt{num}} \quad \dfrac{\dfrac{x \in \mathsf{Domain}([x : \texttt{num}])}{[x : \texttt{num}] \vdash x : \texttt{num}} \quad \dfrac{}{[x : \texttt{num}] \vdash 2 : \texttt{num}}}{[x : \texttt{num}] \vdash x \texttt{ + } 2 : \texttt{num}}}{\varnothing \vdash \texttt{val}\ x \texttt{ = } 1;\ x \texttt{ + } 2 : \texttt{num}}$$

```
/* TFAE */ ((x: Number) => x)(2) * 3           // 6: Number
```

$$\dfrac{\dfrac{\dfrac{\dfrac{x \in \mathsf{Domain}([x : \texttt{num}])}{[x : \texttt{num}] \vdash x : \texttt{num}}}{\varnothing \vdash \lambda x{:}\texttt{num}.x : \texttt{num} \to \texttt{num}} \quad \dfrac{}{\varnothing \vdash 2 : \texttt{num}}}{\varnothing \vdash (\lambda x{:}\texttt{num}.x)(2) : \texttt{num}} \quad \dfrac{}{\varnothing \vdash 3 : \texttt{num}}}{\varnothing \vdash (\lambda x{:}\texttt{num}.x)(2) \texttt{ * } 3 : \texttt{num}}$$

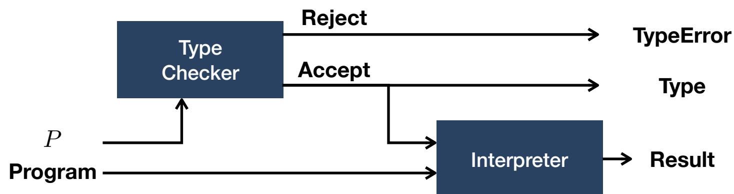
Contents

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

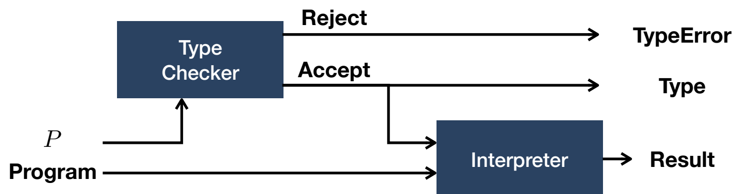
2. Type Checker and Typing Rules
    Recall: Type Checking
    Type Environment
    Numbers
    Addition and Multiplication
    Immutable Variable Definition and Identifier Lookup
    Function Definition and Application
    Examples

3. Interpreter with Type Checker
    Interpreter and Natural Semantics
    Operational Semantics vs Typing Rules

Interpreter with Type Checker

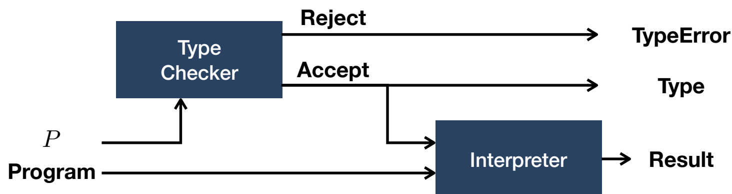


Interpreter with Type Checker




```
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}"
```

Interpreter with Type Checker




```
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}"
```

```
eval("val x = 1; x + 2")          // 3: Number
eval("((x: Number) => x)(2) * 3") // 6: Number
eval("1 + (x: Number) => x")      // a type error thrown by `typeCheck`
```

Interpreter and Natural Semantics

PLRG

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.

Dynamic vs Static and Concrete vs Abstracts

PLRG

What is the difference between **operational semantics** and **typing rules**?

$$\boxed{\sigma \vdash e \Rightarrow v} \qquad \text{vs} \qquad \boxed{\Gamma \vdash e : \tau}$$

Dynamic vs Static and Concrete vs Abstracts

PLRG

What is the difference between **operational semantics** and **typing rules**?

$$\boxed{\sigma \vdash e \Rightarrow v} \quad \text{vs} \quad \boxed{\Gamma \vdash e : \tau}$$

See the table below for the comparison.

	Operational Semantics	**Typing Rules**
Mathematical Notation	$\sigma \vdash e \Rightarrow v$	$\Gamma \vdash e : \tau$
Dynamic/Static	Dynamic	Static
Concrete/Abstract	Concrete	Abstract
Purpose	Evaluation	Type Checking
Implementation	Interpreter	Type Checker
Result	Value	Type

Summary

PLRG

1. TFAE – FAE with Type System
   - Concrete Syntax
   - Abstract Syntax
2. Type Checker and Typing Rules
   - Recall: Type Checking
   - Type Environment
   - Numbers
   - Addition and Multiplication
   - Immutable Variable Definition and Identifier Lookup
   - Function Definition and Application
   - Examples
3. Interpreter with Type Checker
   - Interpreter and Natural Semantics
   - Operational Semantics vs Typing Rules

Exercise #11

PLRG

https://github.com/ku-plrg-classroom/docs/tree/main/cose212/tfae

- 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.

Next Lecture

PLRG

- Typing Recursive Functions

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