# Lecture 18 – Type Systems

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

## Recall

- We learned about **continuations** with the following topics:
  - **Continuations** (Lecture 14 & 15)
  - **First-Class Continuations** (Lecture 16)
  - **Compiling with continuations** (Lecture 17)

- In this lecture, we will learn about **type systems**.

- We will focus on its **motivation** and the **basic concepts**.

## Contents

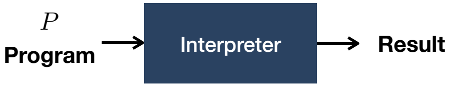
## Contents

## Run-Time Errors

So far, we have designed diverse programming languages with:

- **Syntax**: a grammar that defines the structure of programs
- **Semantics**: a set of rules that defines the meaning of programs

and implemented their **interpreters** in Scala:



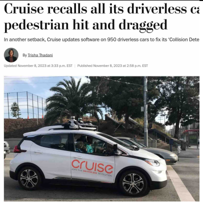
However, we don't have any automatic system to **check** whether a program is evaluated without any **run-time errors**.

For example, following FAE expressions are syntactically correct, but they throw **run-time errors**:

```
/* FAE */
x * 42                  // error: free identifier
0 + (x => x)            // error: cannot add a function
1(2)                    // error: cannot apply a number
```

## Errors in Saftety-Critical Software

Unexpected errors in **safety-critical software** cause serious problems:

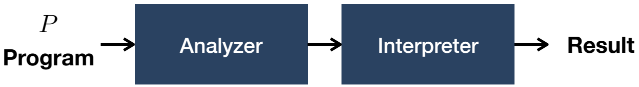
| | | | |
|---|---|---|---|
| **June 4, 1996: Ariane-5 explodes after lift off** Today in History: June 4, 1996: Ariane-5 explodes after lift off Published: June 03, 2016 16:41 Abdul Kareem, Head of Archive Follow us + | **Knight Capital Says Trading Glitch Cost It** BY NATHANIEL POPPER AUGUST 2, 2012 9:07 AM 356 **Runaway Trades Spread Turmoil Across Wall St.** | **Heathrow Airport apologises for IT failure disruption** 16 February 2020 PA MEDIA Heathrow is the busiest airport in Europe |  **Cruise recalls all its driverless ca pedestrian hit and dragged** In another setback, Cruise updates software on 950 driverless cars to fix its 'Collision Dete By Trisha Thadani Updated November 8, 2023 at 3:33 p.m. EST \| Published November 8, 2023 at 2:58 p.m. EST |
| **Rocket** | **Financial** | **Airport** | **Auto. Vehicle** |
| (1996) | (2012) | (2020) | (2023) |

Then, how can we **prevent** such errors?

Can we **automatically** check whether a program does not have any **run-time errors**?

## Detecting Run-Time Errors

We can use various **analysis** techniques to detect run-time errors:



An **analyzer** is a program that takes a program as an input and determines whether the program has a certain property. In this case, the property is **run-time errors**.

We can categorize them into two groups:

- **Dynamic Analysis**: analyze programs by **executing** them
- **Static Analysis**: analyze programs **without executing** them

## Dynamic Analysis

**Dynamic analysis** is a program analysis technique by **executing** them.

Let's perform **dynamic analysis** for the following Scala program:

```
def abs(x: Int): Int = { /* L1 */
  if (x < 0)             /* L2 */
    -x                   /* L3 */
  else                   /* L4 */
    x                    /* L5 */
}                        /* L6 */
```

## Dynamic Analysis

**Dynamic analysis** analyzes programs by **executing** them.

Let's perform **dynamic analysis** for the following Scala program:

```
def abs(x: Int): Int = { /* L1 */
  if (x < 0)             /* L2 */
    -x                   /* L3 */
  else                   /* L4 */
    x                    /* L5 */
}                        /* L6 */
```

| | |
|---|---|
| **L1** | -5 |
| **L2** | -5 |
| **L3** | 5 |
| **L4** | |
| **L5** | |
| **L6** | 5 |

## Dynamic Analysis

**Dynamic analysis** analyzes programs by **executing** them.

Let's perform **dynamic analysis** for the following Scala program:

```
def abs(x: Int): Int = { /* L1 */
  if (x < 0)             /* L2 */
    -x                   /* L3 */
  else                   /* L4 */
    x                    /* L5 */
}                        /* L6 */
```

| | | |
|---|---|---|
| **L1** | -5 | 42 |
| **L2** | -5 | |
| **L3** | 5 | |
| **L4** | | 42 |
| **L5** | | 42 |
| **L6** | 5 | 42 |

## Dynamic Analysis

**Dynamic analysis** analyzes programs by **executing** them.

Let's perform **dynamic analysis** for the following Scala program:

```
def abs(x: Int): Int = { /* L1 */
  if (x < 0)             /* L2 */
    -x                   /* L3 */
  else                   /* L4 */
    x                    /* L5 */
}                        /* L6 */
```

| | | | | | | |
|---|---|---|---|---|---|---|
| **L1** | -5 | 42 | -7 | 99 | 0 | ... |
| **L2** | -5 | | -7 | | | ... |
| **L3** | 5 | | 7 | | | ... |
| **L4** | | 42 | | 99 | 0 | ... |
| **L5** | | 42 | | 99 | 0 | ... |
| **L6** | 5 | 42 | 7 | 99 | 0 | ... |

We can **easily get** the result of the program for each **single input**.

However, it is **difficult** to get information about **all the inputs**.

## Static Analysis

**Static analysis** is a program analysis technique **without executing** them.

Let's perform **static analysis** for the following Scala program:

```
def abs(x: Int): Int = { /* L1 */
  if (x < 0)             /* L2 */
    -x                   /* L3 */
  else                   /* L4 */
    x                    /* L5 */
}                        /* L6 */
```

$$\mathbb{D} = \left\{ \begin{tikzcd} & \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ \texttt{-0} \arrow[d, dash] \arrow[dr, dash] & \texttt{-+} \arrow[dl, dash] \arrow[dr, dash] & \texttt{0+} \arrow[dl, dash] \arrow[d, dash] \\ \texttt{-} \arrow[dr, dash] & \texttt{0} \arrow[d, dash] & \texttt{+} \arrow[dl, dash] \\ & \bot & \end{tikzcd} \right\}$$

Let's define an **abstract domain** $\mathbb{D}$ for integers to analyze the program.

$$\begin{array}{lll} \bot = \varnothing & \top = \mathbb{Z} & \\ \texttt{0} = \{0\} & \texttt{-} = \{x \in \mathbb{Z} \mid x < 0\} & \texttt{+} = \{x \in \mathbb{Z} \mid x \geq 0\} \\ \texttt{-0} = \texttt{-} \cup \texttt{0} & \texttt{-+} = \texttt{-} \cup \texttt{+} & \texttt{0+} = \texttt{0} \cup \texttt{+} \end{array}$$

## Static Analysis

**Static analysis** analyzes programs **without executing** them.

Let's perform **static analysis** for the following Scala program:

```scala
def abs(x: Int): Int = { /* L1 */
  if (x < 0)             /* L2 */
    -x                   /* L3 */
  else                   /* L4 */
    x                    /* L5 */
}                        /* L6 */
```

$$\mathbb{D} = \left\{ \begin{tikzcd} & \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ \texttt{-0} \arrow[d, dash] \arrow[dr, dash] & \texttt{-+} \arrow[dl, dash] \arrow[dr, dash] & \texttt{0+} \arrow[dl, dash] \arrow[d, dash] \\ \texttt{-} \arrow[dr, dash] & \texttt{0} \arrow[d, dash] & \texttt{+} \arrow[dl, dash] \\ & \bot & \end{tikzcd} \right\}$$

| | |
|---|---|
| **L1** | |
| **L2** | |
| **L3** | |
| **L4** | |
| **L5** | |
| **L6** | |

Let's define an **abstract domain** $\mathbb{D}$ for integers to analyze the program:

$$\begin{array}{llllll} \bot & = \varnothing & \top & = \mathbb{Z} & & \\ 0 & = \{0\} & \texttt{-} & = \{x \in \mathbb{Z} \mid x < 0\} & \texttt{+} & = \{x \in \mathbb{Z} \mid x \geq 0\} \\ \texttt{-0} & = \texttt{-} \cup 0 & \texttt{-+} & = \texttt{-} \cup \texttt{+} & \texttt{0+} & = 0 \cup \texttt{+} \end{array}$$

## Static Analysis

**Static analysis** analyzes programs **without executing** them.

Let's perform **static analysis** for the following Scala program:

```
def abs(x: Int): Int = { /* L1 */
  if (x < 0)             /* L2 */
    -x                   /* L3 */
  else                   /* L4 */
    x                    /* L5 */
}                        /* L6 */
```

$$\mathbb{D} = \left\{ \begin{tikzcd} & \top \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ \texttt{-0} \arrow[d, dash] \arrow[dr, dash] & \texttt{-+} \arrow[dl, dash] \arrow[dr, dash] & \texttt{0+} \arrow[dl, dash] \arrow[d, dash] \\ \texttt{-} \arrow[dr, dash] & \texttt{0} \arrow[d, dash] & \texttt{+} \arrow[dl, dash] \\ & \bot & \end{tikzcd} \right\}$$

| | |
|---|---|
| **L1** | ⊤ |
| **L2** | - |
| **L3** | + |
| **L4** | 0+ |
| **L5** | 0+ |
| **L6** | 0+ |

Let's define an **abstract domain** $\mathbb{D}$ for integers to analyze the program:

$$\begin{array}{lll} \bot = \varnothing & \top = \mathbb{Z} & \\ 0 = \{0\} & \texttt{-} = \{x \in \mathbb{Z} \mid x < 0\} & \texttt{+} = \{x \in \mathbb{Z} \mid x \geq 0\} \\ \texttt{-0} = \texttt{-} \cup 0 & \texttt{-+} = \texttt{-} \cup \texttt{+} & \texttt{0+} = 0 \cup \texttt{+} \end{array}$$

We can prove that `abs` always returns a **non-negative** integer (i.e., 0+).

## Soundness vs Completeness

- $\vDash \psi$ denotes that a statement $\psi$ is **provable**.
- $\vdash \psi$ denotes that a statement $\psi$ is **true**.

In a **sound** proof system, all **provable** statements are **true**.

$$\vDash \psi \qquad \Longrightarrow \qquad \vdash \psi$$

In a **complete** proof system, all **true** statements are **provable**.

$$\vdash \psi \qquad \Longrightarrow \qquad \vDash \psi$$

Analysis techniques can be used to prove that a program is **error-free**.

- $\vDash P$ denotes that a program $P$ is **analyzed** as error-free.
- $\vdash P$ denotes that a program $P$ is truly **error-free**.

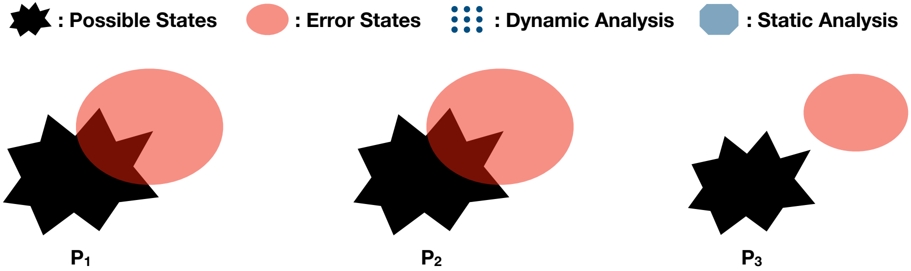
Then, is dynamic/static analysis **sound** or **complete**?

## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can prove a program is error-free.

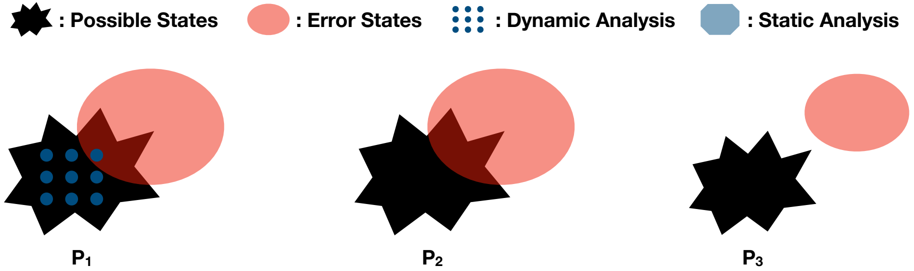
## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



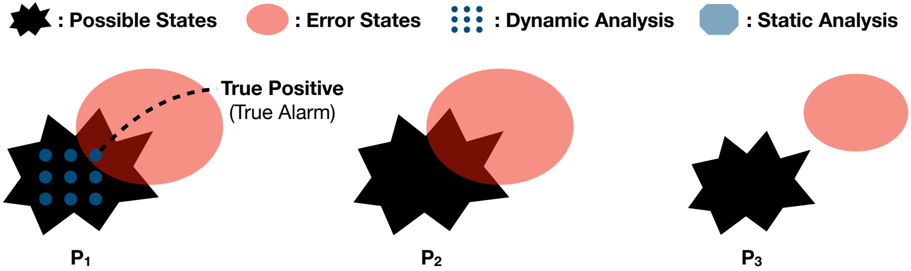
## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



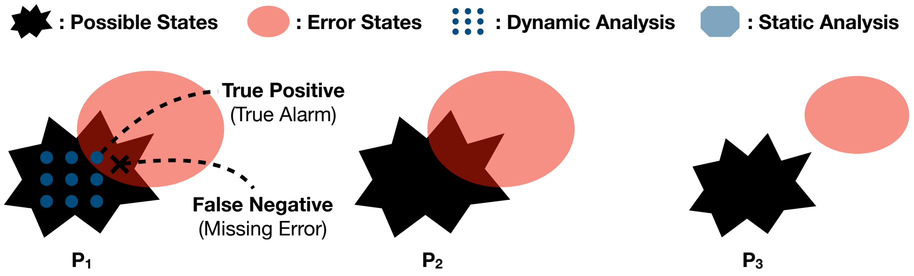
## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



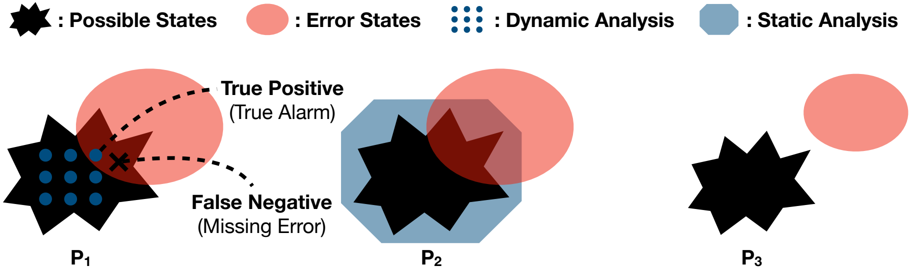
## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



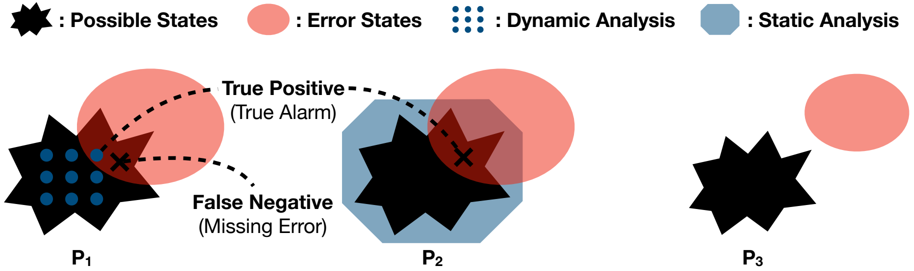
# Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



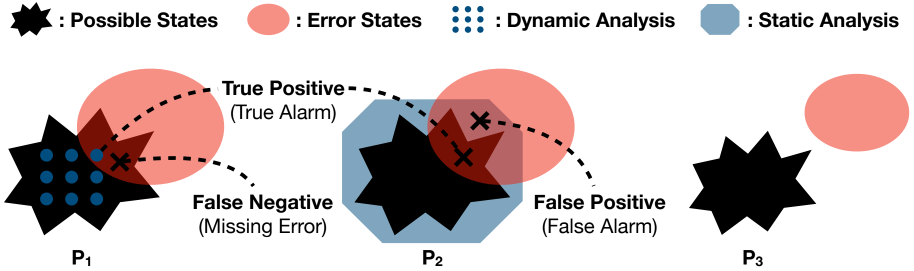
## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



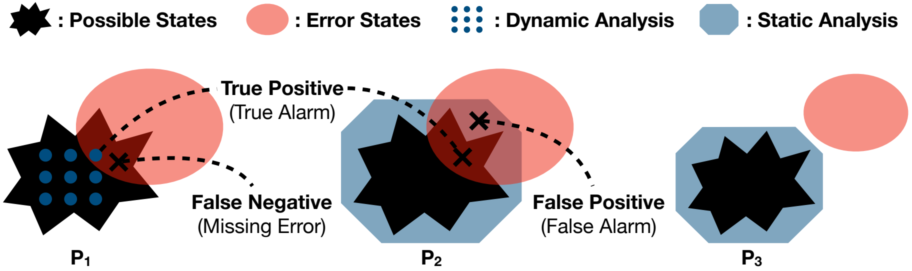
## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



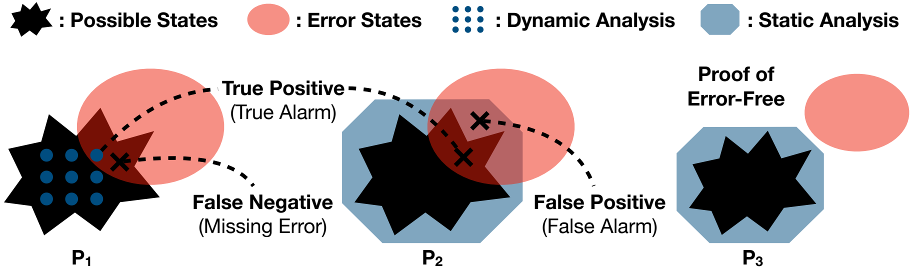
## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



## Soundness vs Completeness

- **Dynamic analysis** is **complete** but **unsound**
  - All the detected errors are **true alarms** (**true positive (TP)**).
  - It will not detect any errors in error-free programs.
  - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete**.
  - **Not** all detected errors are **true alarms**.
  - It suffers from **false alarms** (**false positive (FP)**).
  - There is **no missing errors**. We can **prove** a program is error-free.



## Contents

## Types

**Definition (Types)**

A **type** is a set of values.

For example, the `Int` and `Boolean` types are defined in Scala:

$$\begin{aligned}
\texttt{Int} &= \{n \in \mathbb{Z} \mid -2^{31} \leq n < 2^{31}\} \\
\texttt{Boolean} &= \{\texttt{true}, \texttt{false}\} \\
\texttt{Int => Int} &= \{f \mid f \text{ is a function from } \texttt{Int} \text{ to } \texttt{Int}\}
\end{aligned}$$

```
val n: Int = 42              // 42   : Int
n + 1                        // 43   : Int
val b: Boolean = n > 10      // true : Boolean
def f(x: Int): Int = x + 1   // f    : Int => Int
f(42)                        // 43   : Int
```

## Types and Type Errors

**Definition (Type Errors)**

A **type error** occurs when a program tries to use a value having a type that is **incompatible** with the expected type.

For example, the following program has following type errors:

```
42 + true            // `Int` expected for `+`, but `Boolean` found
if (1) 2 else 3      // `Boolean` expected for `if`, but `Int` found
def f(x: Int): Int = x + 1
f(false)             // `Int` expected for `f`, but `Boolean` found
```

However, not all **run-time errors** are **type errors**:

```
42 / 0               // `ArithmeticException` at run-time
case class A(k: Int)
val x: A = null
x.k                  // `NullPointerException` at run-time
```

## Type Checking

If the following conditions hold, we say **"the expression *e* has type $\tau$"**:

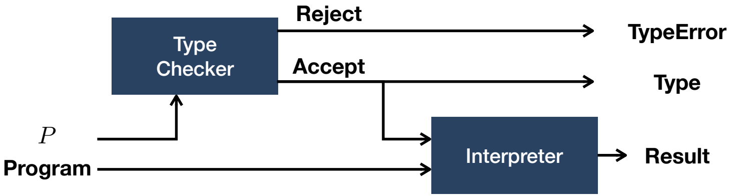
- *e* does not cause any type error, and
- *e* evaluates to a value of type $\tau$ or does not terminate.

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.



## 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.

There are two categories of languages in the context of type system:

- **Statically-typed languages** (or simply typed-language) only allow **well-typed** programs to be executed.
  (e.g. Java, Scala, Haskell, OCaml, Rust, etc.)
- **Dynamically-typed languages** (or simply untyped-language) allow any program to be executed, and types exist only at run-time.
  (e.g. Python, Ruby, JavaScript, etc.)

Most of statically-typed languages has **sound** type systems.

## Summary

1. Safe Language Systems
   - Detecting Run-Time Errors
   - Dynamic vs Static Analysis
   - Soundness vs Completeness

2. Type Systems
   - Types
   - Types
   - Type Checking
   - Type Soundness

## Next Lecture

- Typed Languages

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