

Lecture 18 – Type Systems

COSE212: Programming Languages

Jihyeok Park



2024 Fall

- We learned about **continuations** with the following topics:
 - **Continuations** (Lecture 14 & 15)
 - **First-Class Continuations** (Lecture 16)
 - **Compiling with continuations** (Lecture 17)
- From now on, we will learn about **type systems** with the following topics until the end of the semester:
 - Typed Languages
 - Typing Recursive Functions
 - Algebraic Data Types
 - Parametric Polymorphism
 - Subtype Polymorphism
 - Type Inference
- In this lecture, we will focus on the **motivation** and **basic concepts** of type systems.

1. Motivation: Safe Language Systems

- Detecting Run-Time Errors

- Dynamic vs Static Analysis

- Soundness vs Completeness

2. Type Systems

- Types

- Type Errors

- Type Checking

- Type Soundness

1. Motivation: Safe Language Systems

Detecting Run-Time Errors

Dynamic vs Static Analysis

Soundness vs Completeness

2. Type Systems

Types

Type Errors

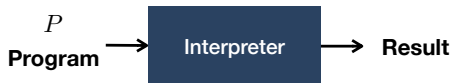
Type Checking

Type Soundness

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
```

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

 <p>June 4, 1996: Ariane-5 explodes after lift off Today In History: June 4, 1996: Ariane-5 explodes after lift off Original Source: 2008-04 Abdul Samad, Head of Archives</p>	 <p>Knight Capital Says Trading Glitch Cost It BY NATHANIEL POPPER AUGUST 2, 2013 6:07 AM 1990 Runaway Trades Spread Turmoil Across Wall St.</p>	 <p>Heathrow Airport apologises for IT failure disruption 14 February 2020</p>	 <p>Cruise recalls all its driverless cars after pedestrian hit and dragged In another setback, Cruise updates software on 250 driverless cars to fix its 'Collision Data' By Steve Berman Updated November 6, 2024 at 2:13 a.m. EST Published November 6, 2024 at 2:09 a.m. EST</p>
Rocket	Financial	Airport	Auto. Vehicle
(1996)	(2012)	(2020)	(2024)

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

Can we **automatically** check whether a program does not have any **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 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 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 */
```

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

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 */
```

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

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 */
```

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	...

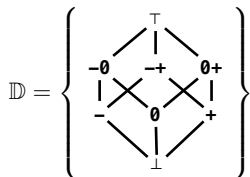
We can easily get the **behavior** of the program for each **single input**.

However, it is **difficult** to get all the **possible behaviors** of the program for **all the inputs**.

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 */
```



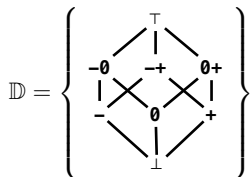
Let's define an **abstract domain** \mathbb{D} for integers to analyze the program.

$$\begin{array}{lll} \perp = \emptyset & \top = \mathbb{Z} & \\ 0 = \{0\} & - = \{x \in \mathbb{Z} \mid x < 0\} & + = \{x \in \mathbb{Z} \mid x > 0\} \\ -0 = - \cup 0 & -+ = - \cup + & 0+ = 0 \cup + \end{array}$$

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 */
```



L1	
L2	
L3	
L4	
L5	
L6	

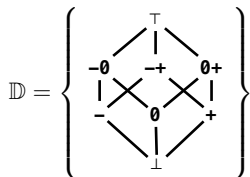
Let's define an **abstract domain** \mathbb{D} for integers to analyze the program:

$$\begin{array}{lll} \perp & = & \emptyset \\ 0 & = & \{0\} \\ -0 & = & - \cup 0 \\ \top & = & \mathbb{Z} \\ - & = & \{x \in \mathbb{Z} \mid x < 0\} \\ -+ & = & - \cup + \\ + & = & \{x \in \mathbb{Z} \mid x > 0\} \\ 0+ & = & 0 \cup + \end{array}$$

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 */
```



L1	\top
L2	$-$
L3	$+$
L4	$0+$
L5	$0+$
L6	$0+$

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

$$\begin{array}{lll} \perp = \emptyset & \top = \mathbb{Z} & \\ 0 = \{0\} & - = \{x \in \mathbb{Z} \mid x < 0\} & + = \{x \in \mathbb{Z} \mid x > 0\} \\ -0 = - \cup 0 & -+ = - \cup + & 0+ = 0 \cup + \end{array}$$

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

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

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

$$\vdash \psi \quad \Longrightarrow \quad \models \psi$$

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

$$\models \psi \quad \Longrightarrow \quad \vdash \psi$$

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.
- $\models P$ denotes that a program P is truly **error-free**.

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

- **Dynamic analysis** is **complete** but **unsound** in general.
 - 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** in general.
 - **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.

- **Dynamic analysis** is **complete** but **unsound** in general.
 - 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** in general.
 - **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.

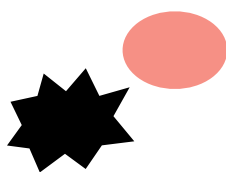
★ : Possible States ● : Error States ●●● : Dynamic Analysis ● : Static Analysis



P₁



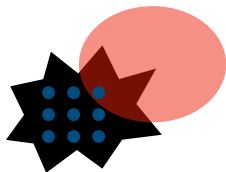
P₂



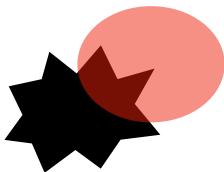
P₃

- **Dynamic analysis** is **complete** but **unsound** in general.
 - 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** in general.
 - **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.

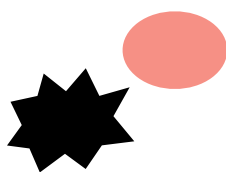
★ : Possible States ● : Error States ●●● : Dynamic Analysis ● : Static Analysis



P₁



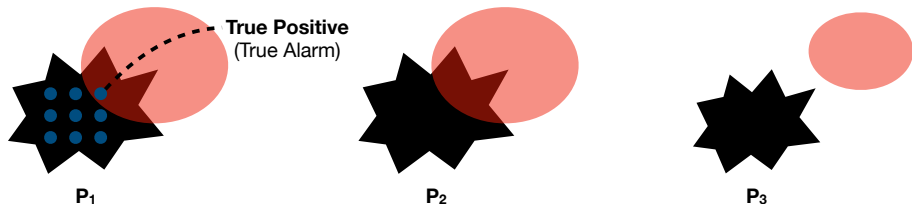
P₂



P₃

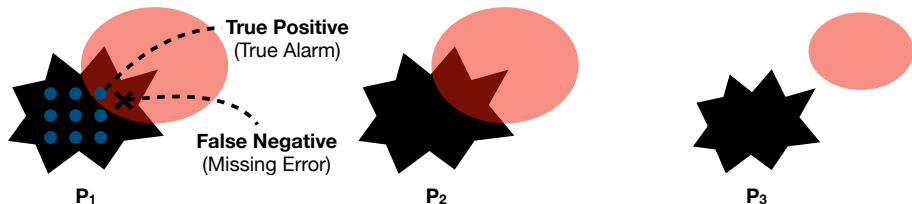
- **Dynamic analysis** is **complete** but **unsound** in general.
 - All the detected errors are **true alarms** (**true positive (TP)**).
 - It will not detect any errors in error-free programs.
 - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete** in general.
 - **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.

★ : Possible States ● : Error States ●●● : Dynamic Analysis ● : Static Analysis



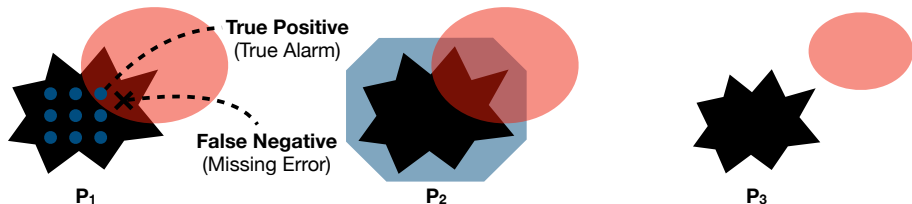
- **Dynamic analysis** is **complete** but **unsound** in general.
 - All the detected errors are **true alarms** (**true positive (TP)**).
 - It will not detect any errors in error-free programs.
 - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete** in general.
 - **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.

★ : Possible States ● : Error States ●●● : Dynamic Analysis ● : Static Analysis

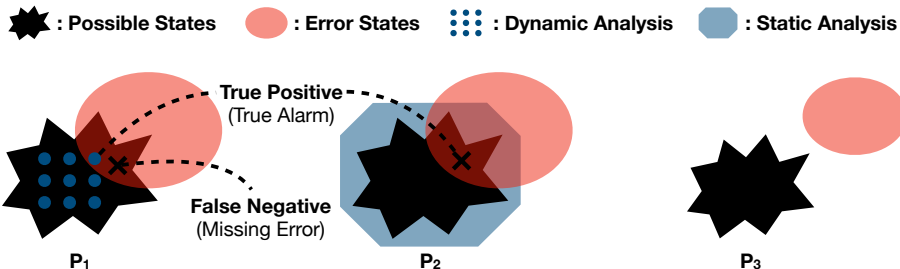


- **Dynamic analysis** is **complete** but **unsound** in general.
 - All the detected errors are **true alarms** (**true positive (TP)**).
 - It will not detect any errors in error-free programs.
 - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete** in general.
 - **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.

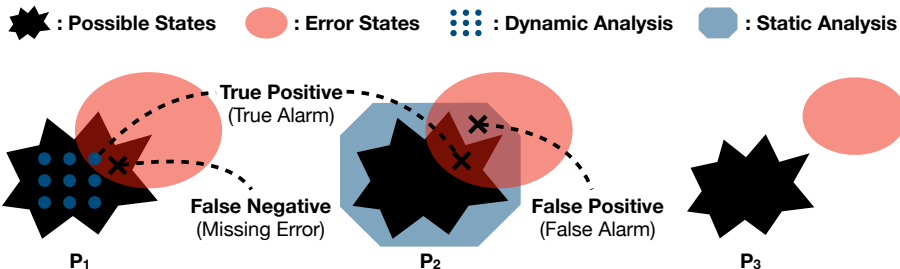
★ : Possible States ● : Error States ●●● : Dynamic Analysis ⬡ : Static Analysis



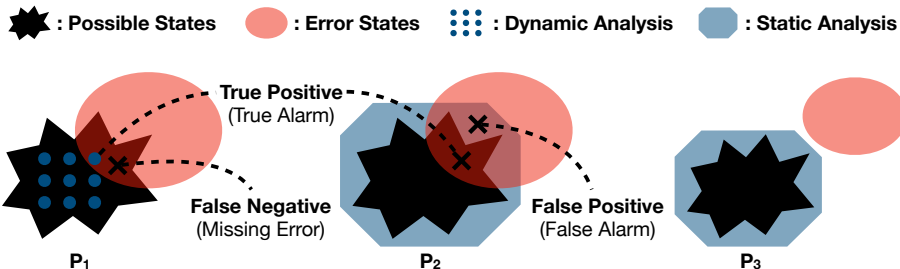
- **Dynamic analysis** is **complete** but **unsound** in general.
 - All the detected errors are **true alarms** (**true positive (TP)**).
 - It will not detect any errors in error-free programs.
 - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete** in general.
 - **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.



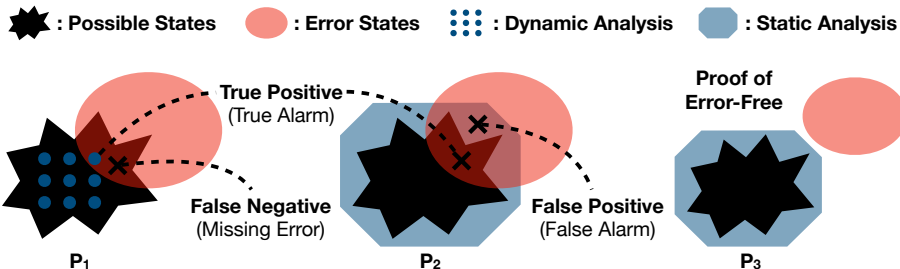
- **Dynamic analysis** is **complete** but **unsound** in general.
 - All the detected errors are **true alarms** (**true positive (TP)**).
 - It will not detect any errors in error-free programs.
 - It suffers from **missing errors** (**false negative (FN)**).
- **Static analysis** is **sound** but **incomplete** in general.
 - **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.



- **Dynamic analysis** is **complete** but **unsound** in general.
 - 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** in general.
 - **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.



- **Dynamic analysis** is **complete** but **unsound** in general.
 - 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** in general.
 - **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.



1. Motivation: Safe Language Systems

Detecting Run-Time Errors

Dynamic vs Static Analysis

Soundness vs Completeness

2. Type Systems

Types

Type Errors

Type Checking

Type Soundness

Definition (Types)

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

For example, the `Int`, `Boolean`, and `Int => Int` types are defined as the following sets of values in Scala.

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

```
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
```

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 Scala program has 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
```

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

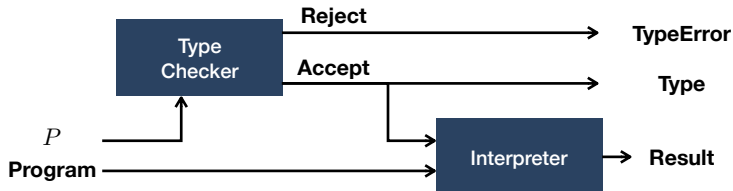
- e does not cause any type error, and
- e evaluates to a value of type τ or does not terminate.

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.



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.)

Type systems in most statically-typed languages are designed to be **sound**.

1. Motivation: Safe Language Systems

- Detecting Run-Time Errors

- Dynamic vs Static Analysis

- Soundness vs Completeness

2. Type Systems

- Types

- Type Errors

- Type Checking

- Type Soundness

- Typed Languages

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