CSCI 740 - Programming Language Theory

Lecture 11

Introduction to Simple Types

Instructor: Hossein Hojjat

September 25, 2017

Recap

- A λ-calculus expression is defined as

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & e\ e & \text{application}
\end{array}
$$

α-conversion	$\lambda x.e \ \rightarrow_\alpha \ \lambda y.e[x \mapsto y]$	if $y \notin \mathsf{FV}(e)$
β-reduction	$(\lambda x.e)\ e' \ \rightarrow_\beta \ e[x \mapsto e']$	
η-conversion	$\lambda x.(e\ x) \ \rightarrow_\eta \ e$	if $x \notin \mathsf{FV}(e)$

Recap

- A λ-calculus expression is defined as

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & e\ e & \text{application}
\end{array}
$$

α-conversion	$\lambda x.e \ \rightarrow_\alpha \ \lambda y.e[x \mapsto y]$	if $y \notin \mathsf{FV}(e)$
β-reduction	$(\lambda x.e)\ e' \ \rightarrow_\beta \ e[x \mapsto e']$	
η-conversion	$\lambda x.(e\ x) \ \rightarrow_\eta \ e$	if $x \notin \mathsf{FV}(e)$

This lecture: Types

Paradoxes & Types

- One of the original motivations for type type theory was to eliminate certain logical paradoxes in untyped λ-calculus

Russell's Paradox

- Consider the function R which negates its argument applied to itself

$$R = \lambda x.\text{NOT } (x\ x)$$

- Now apply R to itself

$$\begin{aligned} (R\ R) &= (\lambda x.\text{NOT } (x\ x))\ R \\ &=_\beta \text{NOT } (R\ R) \end{aligned}$$

- Simply typed-calculus forbids $x\ x$ with a type system

Types for Programming Languages

> "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute"
> Benjamin Pierce, "Types and Programming Languages" (MIT, 2002)

Type System

Narrow View

- A mechanism for ensuring that variables only take values from predefined sets
  - e.g. Integers, Strings, Characters
- A mechanism for avoiding unchecked errors
  - by ruling out programs with undefined behaviors
  - by specifying how a program should fail (e.g. `NullPointerException`)

Expansive View

- A light-weight proof system and annotation mechanism for efficiently checking for a specific property of interest
- Address bugs that go beyond corner-cases in the semantics
  - Information flow violations
  - Deadlocks
  - ...

Types

- **Type:** A method of classifying objects (values) in a language
- $x : \tau$ says object x has type τ or object x belongs to a type τ
- τ denotes a set of values
- This notion of types is different from types in languages like C, where a type is a storage class specifier

Type Correctness

- If $x : \tau$ then only those operations that are appropriate to set τ may be performed on x
- A program is **type correct** if it never performs a wrong operation on an object
  - Add an `Int` and a `Boolean`
  - Head of an `Int`
  - Square root of a `list`

Classification of Type Systems

- **Monomorphic:** Each identifier has exactly one type
- **Polymorphic:** An identifier can have multiple types
- **Static:** Type correctness is checked at compile time
- **Dynamic:** Type correctness is checked at run time

	static	dynamic
monomorphic	Pascal	
polymorphic	ML, Haskell, Scala , Java	Lisp, Smalltalk

Type systems: a Discipline for Programming

- Proper type systems provide strong guarantees
  - Scala, ML, Haskell: memory corruption impossible
- Sometimes frustrating: seen as a hurdle
- For some applications (rapid application development) are overly restrictive
  - Dynamic scripting languages (Perl,Tcl, Python, ...)

Static Analysis

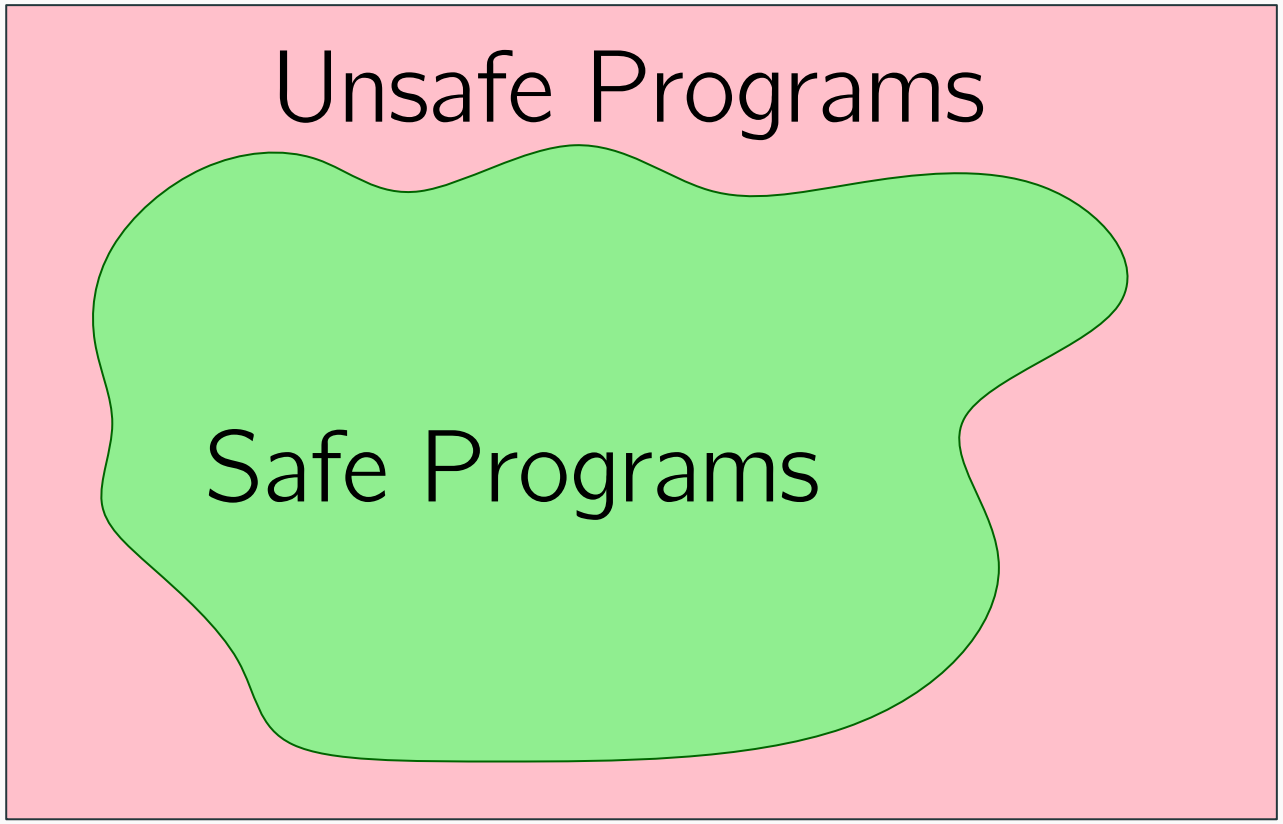
- Static type checking is the most widely used form of static analysis
- **Static Analysis:** analyze the program without running it
- Some other type of typical errors that static analysis can catch:
  - Security: Buffer overruns, improperly validated input
  - Memory safety: Null dereference, uninitialized data
  - Resource leaks: Memory, OS resources
  - Exceptions: Arithmetic/library/user-defined
  - Encapsulation: Accessing internal data, calling private functions
  - ...

Rice's Theorem

"Any nontrivial property about the language recognized by a Turing machine is undecidable."

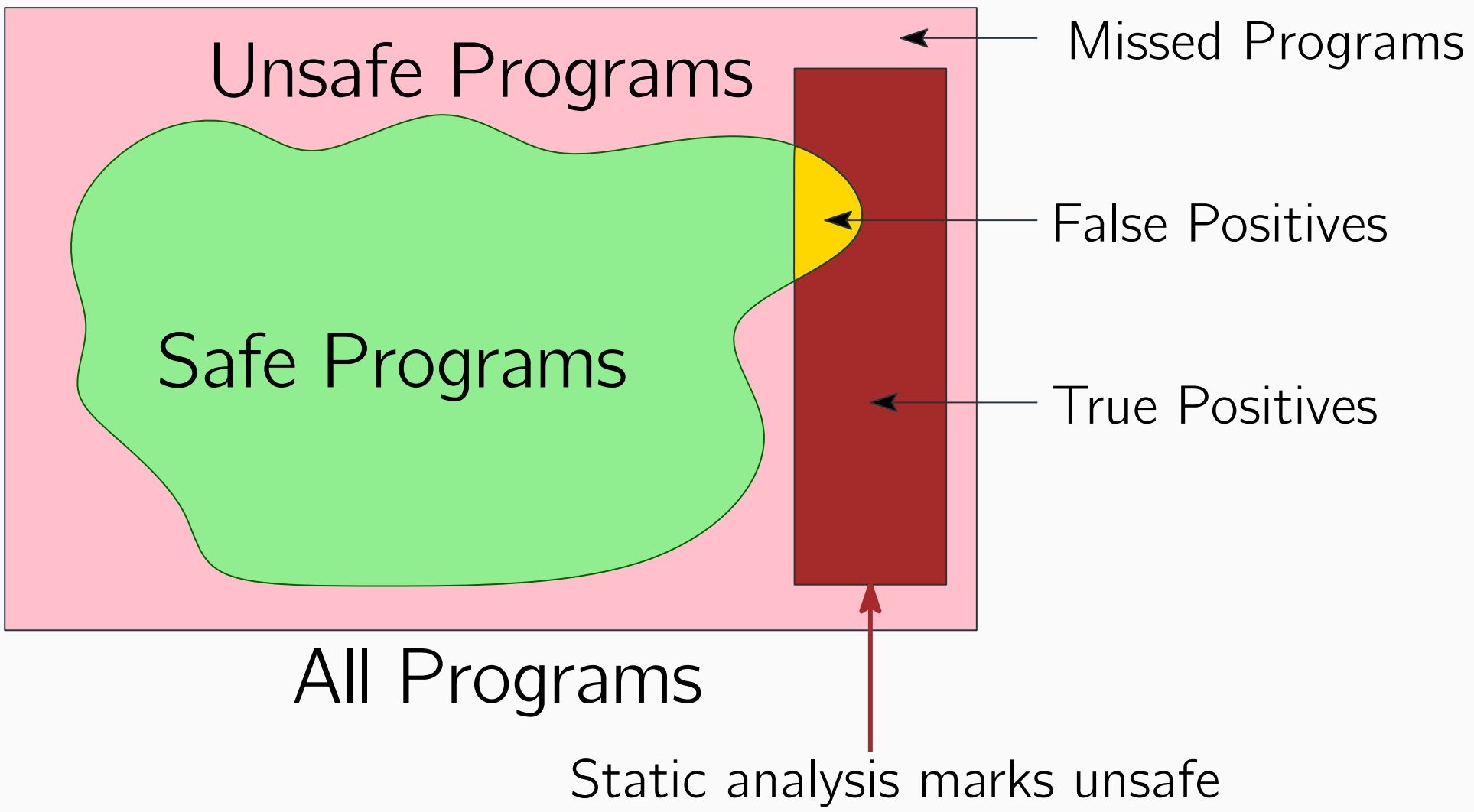
- Many static analysis techniques are sound but incomplete

Space of Programs



All Programs

Space of Programs



False Positives (False Alarms)

- Static analysis makes a number of approximations:
  it may raise false alarms
- False alarms often hated by users
  - Sometimes too many potential problems raised by tool
- Often programmers need to check errors to rule out false alarms
- True defects may be lost, buried in details
- Tools compete on false positive rate for usability

False Positives in Type Checking

```
short s = 0;
int i = s;
short r = i;
```

False Positives in Type Checking

```
short s = 0;
int i = s;
short r = i;
```

```
$ javac test.java

test.java:7: error: incompatible types: possible lossy
                    conversion from int to short
short r = i;
          ^
1 error
```

False Positives in Type Checking

```
int i = 0;
if (false) {
  i = i + "hello";
}
```

False Positives in Type Checking

```
int i = 0;
if (false) {
  i = i + "hello";
}
```

```
$ javac test.java

test.java:7: error: incompatible types:
                    String cannot be converted to int
i = i + "hello";
      ^
1 error
```

False Positives in Type Checking

- No false positives in Python

```
i = 0;
if (False):
  i = i + "hello";
```

- Dynamic type checking has run-time overhead
- Not suitable for time-critical applications

Gradual Typing

- Static and dynamic type systems have complimentary strengths
- **Static typing:** full-coverage error checking, efficient execution, machine-checked documentation
- **Dynamic typing:** rapid development and fast adaption to changing requirements
- **Gradual typing:** an approach for integrating static and dynamic type checking
- Gives the programmer fine-grained control over
  - which regions of a program are statically checked
  - which regions of a program are dynamically checked

Polymorphism

- In a monomorphic language (e.g. Pascal):
  you define a different `length` function for each type of `list`
- In a polymorphic language (e.g. ML):
  you define a polymorphic type (`list t`), where `t` is a type variable, and a single function for computing the length
- Haskell and most modern functional languages have polymorphic types and follow the Hindley-Milner type system

Simple types = Non polymorphic types

Formalizing a Type System

- Type system is almost never orthogonal to the semantics of the language
  - The types in a program can affect its behavior
    (e.g. operator overloading)
- We don't define the type system in isolation, we define a typed language including definitions of
  - The syntax
  - dynamic semantics (e.g. operational semantics)
  - static semantics
    - also known as typing rules
    - describe how types are assigned to elements in a program
  - type soundness argument
    - describe the relationship between static and dynamic semantics

Basic Notation

- Type system assigns types to elements in the language
  - Basic notation: $e : T$ (e is of type T)
- The types of some elements depends on the **environment**
  - Basic notation: $\Gamma \vdash e : T$
    (Given environment Γ, we can derive that e is of type T)
  - An environment associates types with free variables
  - This is called a **Judgment**
  - Example:

$$x : \texttt{Int}, y : \texttt{Int} \vdash x + y : \texttt{Int}$$

Static Semantics

Typing Rules

- Typing rules tell us how to derive typing judgments
- Very similar to derivation rules in Big Step Operational Semantics

$$\frac{\text{Premises}}{\text{Judgment}}$$

- Example: Language of Expressions

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

Example: Language of Expressions

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

- Show that the following Judgment is valid

$$x : \mathtt{Int}, y : \mathtt{Int} \ \vdash \ x + (y + 5) : \mathtt{Int}$$

$$\dfrac{\dfrac{x : \mathtt{Int} \in \{x : \mathtt{Int}, y : \mathtt{Int}\}}{x : \mathtt{Int}, y : \mathtt{Int} \ \vdash \ x : \mathtt{Int}} \qquad \dfrac{\dfrac{y : \mathtt{Int} \in \{x : \mathtt{Int}, y : \mathtt{Int}\}}{x : \mathtt{Int}, y : \mathtt{Int} \vdash y : \mathtt{Int}} \qquad \dfrac{}{x : \mathtt{Int}, y : \mathtt{Int} \vdash 5 : \mathtt{Int}}}{x : \mathtt{Int}, y : \mathtt{Int} \vdash (x + 5) \ : \ \mathtt{Int}}}{x : \mathtt{Int}, y : \mathtt{Int} \ \vdash \ x + (y + 5) : \mathtt{Int}}$$

Simply Typed λ-calculus (F_1)

- Modify the syntax of the λ-calculus to add type annotations for parameters of functions

Syntax

Terms	$e ::= x$	$\mid \lambda x{:}\,\boldsymbol{\tau}.\ e$	$\mid e_1\ \ e_2$
	$\mid n$	$\mid e_1 + e_2$	$\mid e_1 = e_2$
	$\mid$ if e_1 then e_t else e_f		
Types	$\boldsymbol{\tau} ::=$ Int	$\mid$ Bool	$\mid \tau_1 \to \tau_2$

- $\tau_1 \to \tau_2$ is the function type
- $\to$ associates to the right
- This language is also called F_1

Static Semantics of F_1

- Basic Typing Rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \ \ e_2 : \tau}$$

- Extensions

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 \ = \ e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}$$