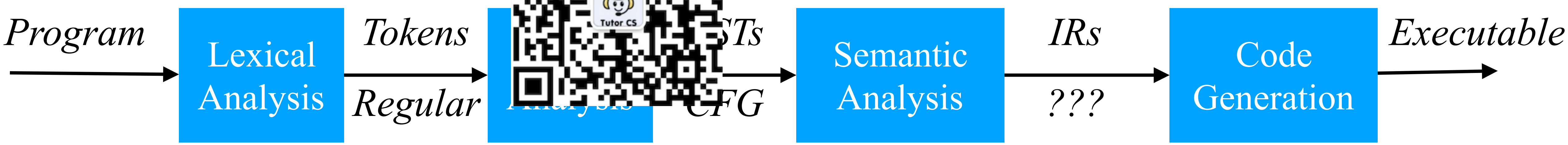
CS 160 Compilers

Lecture 11: Type Checking

Yu Feng

Fall 2021

A typical flow of a compiler



Outline

- We will talk about

- What types compute

- Why types are useful

- Brief survey of types in the real world

Motivation

- When writing programs, everything is great as long as the program works.
- Unfortunately, this is usually not the case
- Programs crash, don't compute what we want them to compute, etc.
- This is arguably the biggest problem software faces today

Software correctness

- We would really w[illegible]e that software has the properties we care about

- And in some sense, we seem to have all the ingredients:

  - A formal understanding of syntax

  - A rigorous mathematic notation to express meaning of programs

  - Some proofs in class showing that a small toy program must evaluate to a certain integer

- So what is the problem?

Software correctness

- Problem: Rice's th[illegible]ny non-trivial property about a Turing machine is [illegible]ble

- This means that we can never give an algorithm, that for all programs can decide if this program has an error on some inputs.

- What can we do?

- Give up?

Big idea

- Big Idea: Just bec[illegible]annot prove something about the original program d[illegible]ean we cannot prove something about an *abstraction* of the program.

- Strategy: In addition to the operational semantics, we will also define *abstract semantics* that will overapproximate the states a program is in.

- Example: In Patina, the operational semantics compute a concrete integer or list, while our abstract semantics only compute the if the result is integer or list.

Abstraction

- Of course, any abs[illegible]will be less precise than the program
- One popular abstraction: types
- Let's assume we have types Int and List
- Example: let x = 10 in x
- Operational semantics yield concrete value 10
- Abstract semantics that only differentiate the kind (or type) of the expression yield: Integer

Abstraction

- But we don't just [illegible] abstraction, we need abstractions that *overapproxim*[illegible]sult of the concrete program

- Recall the example: let x = 10 in x

- Abstract value *Integer* overapproximates 10 since 10 is a kind of integer

- On the other hand, abstract value *List* does not overapproximate 10.

Soundness

- Specifically, we o[illegible]bout abstract semantics that are sound

- Soundness means that for any program: If we evaluate it under *concrete* semantics (operational semantics) and our *abstract* semantics, the abstract value obtained overapproximates the concrete value.

Soundness

- The reason we only care about sound abstract semantics is the following:

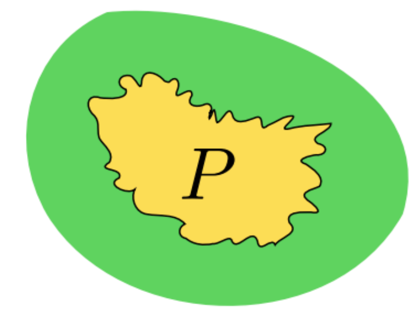
- Theorem: If some abstract semantics are sound and an expression is of abstract value x, then its concrete value y is always part of the abstract value x.

- Why is this useful?

- This means that if a program has no error in the abstract semantics, it is guaranteed not to have an error in the concrete semantics.

- ASTREE tools: http://www.astree.ens.fr/

Cost of abstraction



- But using an abstraction comes at a cost:
- What do we know if a program has an error in the abstract semantics?

- Nothing. We only know that the program may have an error (or not)

- If under some abstract semantics a program has an error, but the program in fact never has this error under concrete semantics, we say this is a false positive

- Finding the right abstractions is key! Abstraction must match properties of interest to be proven.

Types

- In this class, we w[illegible]n one kind of abstraction: types

- This means abstract values are the types in the language

- What is a type? An abstract value representing an (usually) infinite set of concrete values

- Question: For proving what kind of properties are types as abstract values useful?

- Answer: To avoid run-time type errors!

Type checking v.s. Type inference

- We saw earlier that types are just a kind of abstract value
- Two strategies to compute types:

  - (1) Ask the programmer

  - (2) Compute types of expressions from the known types of concrete values

- Most popular languages use strategy (1), known as type checking

Type checking

- Type checking: Th[illegible]mmer provides some types (typically, every variable) an[illegible]piler complains if some types are inconsistent.

- Languages with type checking: C, C++, Java, ...

- We will (formally) study type checking first.

Type inference

- In languages with type inference, you don't have to write any types!

- The compiler automatically computes the "best" type of every expression and reports an error if the computed types are not compatible

- Very cool and intriguing idea. We will learn exactly how it works in a few lectures

- There are languages with this feature: ML, OCaml, Haskell, Go