CS 162 Programming languages

Lecture 9: Type Checking

Yu Feng

Winter 2023

Outline

- We will talk about types
- 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 want to prove 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 theorem. Any non-trivial property about a Turing machine is undecidable

- 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 because we cannot prove something about the original program does not mean 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 λ^+, the operational semantics compute a concrete integer or list, while our abstract semantics only compute the if the result is of kind integer or list.

Abstraction

- Of course, any abstraction 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

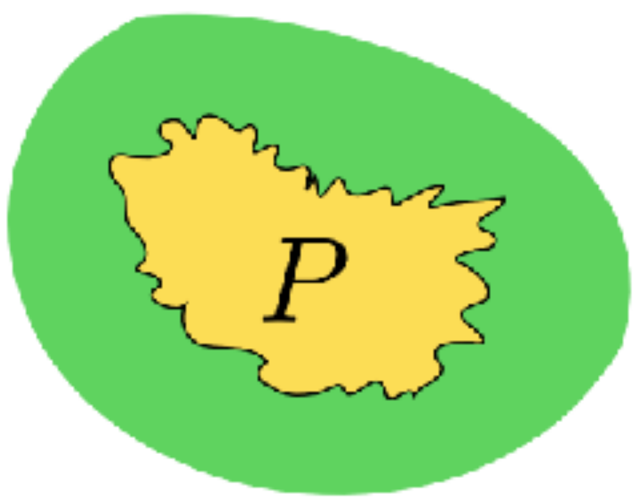
Abstraction

- But we don't just want any abstraction, we need abstractions that *overapproximate* the result 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 only care about abstract semantics that are sound

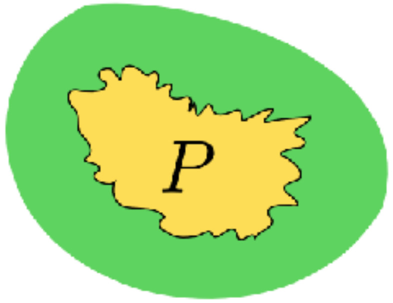
- Soundness means that for any program: If we evaluate it under *concrete* semantics (operational semantics) and our *abstract* semantics, the abstract value obtained overapproximates the concrete value.



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.

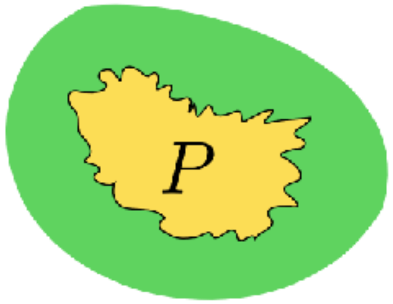


- Why is this useful?

- 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 will focus on 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:
    - Ask the programmer
    - Compute types of expressions from the known types of concrete values.
- Most popular languages use the strategy, known as type checking

Type checking

- Type checking: The programmer provides some types (typically, every variable) and the compiler 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

Operational semantics in λ+

$$\frac{}{\mathbb{E} \vdash i \Downarrow i}\ \textsc{Int} \qquad \frac{\mathbb{E} \vdash e_1 \Downarrow i_1 \qquad \mathbb{E} \vdash e_2 \Downarrow i_2}{\mathbb{E} \vdash e_1 \oplus e_2 \Downarrow i_1 \oplus i_2}\ \textsc{Arith}$$

$$\frac{}{\mathbb{E} \vdash \mathsf{lambda}\, x.\ e \Downarrow \mathsf{lambda}\, x.\ e}\ \textsc{Lambda}$$

$$\frac{\mathbb{E} \vdash e_1 \Downarrow \mathsf{lambda}\, x.\ e_1' \qquad \mathbb{E} \vdash e_2 \Downarrow v \qquad \mathbb{E} \vdash [x \mapsto v]e_1' \Downarrow v'}{\mathbb{E} \vdash (e_1\ e_2) \Downarrow v'}\ \textsc{App}$$

$$\frac{\mathbb{E}(x) = v}{\mathbb{E} \vdash x \Downarrow v}\ \textsc{Var} \qquad \frac{\mathbb{E} \vdash e_1 \Downarrow v_1 \qquad \mathbb{E}[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\mathbb{E} \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Downarrow v_2}\ \textsc{Let}$$

Types in λ^+

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}\ \textsc{T-Int} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Int}}\ \textsc{T-Arith}$$

Type environment

$$\frac{\Gamma, x : \mathsf{T}_1 \vdash e : \mathsf{T}_2}{\Gamma \vdash \mathsf{lambda}\, x : \mathsf{T}_1.\ e : \mathsf{T}_1 \to \mathsf{T}_2}\ \textsc{T-Lambda}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \vdash e_2 : \mathsf{T}_1}{\Gamma \vdash (e_1\ e_2) : \mathsf{T}_2}\ \textsc{T-App}$$

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}}\ \textsc{T-Var} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma, x : \mathsf{T}_1 \vdash e_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \mathsf{T}_2}\ \textsc{T-Let}$$