CS 162 Programming languages

Lecture 10: Type Checking II

Yu Feng
Winter 2022

Outline

- We will talk about types in λ^+

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

- 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?

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

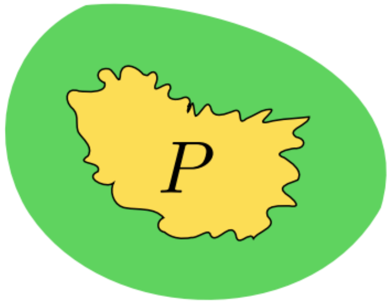
Abstraction

- 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

- 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/

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!

Adding types to λ⁺

- The original syntax of λ^+

$$
\begin{array}{rcl}
e & ::= & e_1 + e_2 \\
 & | & \mathsf{Int} \quad | \quad \texttt{let } x = e_1 \texttt{ in } e_2 \\
 & | & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
 & | & \lambda x.e \quad | \quad e_1\ e_2 \\
 & | & !e \quad | \quad \#e \quad | \quad e_1 @ e_2
\end{array}
$$

- Adding type to λ^+

$$\mathsf{T} \ ::= \ \mathsf{T} \to \mathsf{T} \mid \mathsf{Int} \mid \mathsf{List}[\mathsf{T}]$$

Typing rules for arithmetic

$$\frac{}{\Gamma \vdash i : \mathsf{Int}} \ \textsc{T-Int}$$

Any integer constant i is of type **integer**

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \Box \in \{+, -, *\}}{\Gamma \vdash e_1 \Box e_2 : \mathsf{Int}} \ \textsc{T-Arith}$$

if e_1 and e_2 are both integers, then
$e_1 \Box e_2$ will also be integer

Typing rules for if-else

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{T} \qquad \Gamma \vdash e_3 : \mathsf{T}}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathsf{T}}\ \textsc{If}$$

if e_1 is of type integer, both e_2 and e_3 have the same type T, then the whole if-else expression is of type T

Typing rules for lambda

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

If x is of type T_1 and e is of type T_2,
then the lambda expression has type $T_1 \rightarrow T_2$

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

if e_1 has type $T_1 \rightarrow T_2$, and e_2 has type T_1,
then lambda app return a value of type T_2

Typing rules for let-binding

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}} \text{ T-VAR}$$

If the type of x is T in the current type environment, then x is of type T

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

if x and e_1 are of type T_1 and e_2 is of type T_2, then the whole expression has type T_2

Typing rules for list

$$\frac{}{\Gamma \vdash \mathsf{Nil} : \mathsf{List[T]}} \text{ T-NIL}$$

An empty list is a list

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{List[T]}}{\Gamma \vdash e_1 \mathbin{@} e_2 : \mathsf{List[T]}} \text{ T-CONS}$$

if e_1 is of type T and e_2 is of type List[T], then $e_1@e_2$ is of type List[T]

$$\frac{\Gamma \vdash e : \mathsf{List[T]}}{\Gamma \vdash !e : \mathsf{T}} \text{ T-HEAD}$$

if e contains a list of elements of type T, then !e is of type T

$$\frac{\Gamma \vdash e : \mathsf{List[T]}}{\Gamma \vdash \#e : \mathsf{List[T]}} \text{ T-TAIL}$$

if e contains a list of elements of type T, then #e is of type List[T]

Typing checking by example

wrong!

$x : int \quad x+2 : int$

$\lambda x : int. x+2 : int \to int \qquad y : List[int]$

$[1,2] : List[int] \qquad \lambda x : int.\ x+2 \ \ y$

$let\ y = \underbrace{[1,2]}_{e_1}\ in\ \underbrace{\lambda x : int.\ x+2}_{e_2}\ \ y)$

TODOs by next lecture

- HW4 will be due soon. Please start ASAP!