CVO103: Programming Languages

Lecture 8 — Type System
(1) Motivation

Hakjoo Oh
2018 Spring

Review: Our Programming Language System So Far

- Designed and implemented a programming language system:
    - Rigorously defined syntax and semantics of the language.
    - Faithfully implemented the interpreter based on the formal design.

$$\text{Program} \rightarrow \boxed{\text{Interpreter}} \rightarrow \text{Result}$$

- A well-designed language indeed, with clean syntax and semantics :-)
- However, the current system has a significant shortcoming.

The Language System is Unsafe

- It attempts to execute unsafe programs too, only to fail at runtime.

  Unsafe Program $\rightarrow$ $\boxed{\text{Interpreter}}$ $\rightarrow$ Runtime Failure

  For example,
  - `if 3 then 88 else 99`
  - `(proc (x) (x 3)) 4`
  - `let x = iszero 0 in (3-x)`
- We want to avoid evaluating unsafe programs but the language system puts all the burden of writing safe programs on the programmers.
  - Also in C, C++, Python, JavaScript, etc.
- This manual approach of avoiding software errors has proven extremely unsuccessful.

Software Failures in History

- (1996) The Arian-5 rocket, whose development required 10 years and $8 billion, exploded just 37s after launch due to software error.

- (1998) NASA's Mars climate orbiter lost in space. Cost: $125 million
- (2000) Accidents in radiation therapy system. Cost: 8 patients died
- (2007) Air control system shutdown in LA airport. Cost: 6,000 passengers stranded
- (2012) Glitch in trading software of Knight Captal. Cost: $440 million
- (2014) Airbag malfunction of Nissan vehicles. Cost: $1 million vehicles recalled
- ... Countless software projects failed in history.

Dream: Safe Language System

- Automated technology for analyzing the safety and detecting all bugs of programs statically.

$$\text{Program} \rightarrow \boxed{\text{Analyzer}} \rightarrow \boxed{\text{Interpreter}} \rightarrow \text{Result}$$

Dream: Safe Language System

- Automated technology for analyzing the safety and detecting all bugs of programs statically.

$$\text{Program} \rightarrow \boxed{\text{Analyzer}} \rightarrow \boxed{\text{Interpreter}} \rightarrow \text{Result}$$

- Unfortunately, "static analysis" is undecidable.


Halting problem is undecidable!

- More precisely, sound and complete static analysis is impossible.
- Approximate (yet useful) ones are possible.

Soundness and Completeness

- **Soundness**: Analyzer can prove the absence of errors. If analyzer accepts a program, then the program is safe. If a program has errors, analyzer rejects the program. All unsafe programs are rejected. No false negatives.

- **Completeness**: Analyzer can prove the presence of errors. If analyzer rejects a program, then the program has errors. If the program is safe, analyzer accepts the program. All safe programs are accepted. No false positives.

Plan: Building a Static Type System for Our Language

Static analyzer that detects type errors (runtime failures caused by type mismatches).
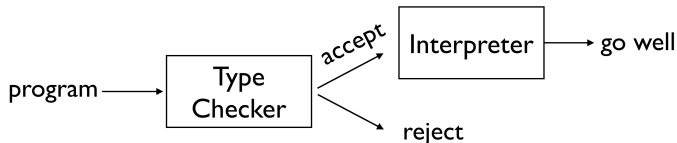
- if true then 88 else 99
- if 3 then 88 else 99
- (proc (x) (x 3)) (proc (x) x)
- (proc (x) (3 x)) e
- let x = iszero 0 in (3-x)

cf) Detecting other types of errors is beyond the scope of our type system, e.g., ((proc (x) (4 / x)) 0.

Sound but Incomplete Type System

- We settle for a sound but incomplete type system.
    - Sound: detecting all type errors.
    - Incomplete: some safe programs will not pass our type system.

- Type systems in modern programming languages such as ML, Haskell, and Scala are also sound but incomplete.

- cf) Type systems in languages like C and C++ are neither sound nor complete.

Next: Sound Type System for PROC

$$
\begin{aligned}
E \;\rightarrow\; & n \\
\mid\; & x \\
\mid\; & E + E \\
\mid\; & E - E \\
\mid\; & \texttt{iszero } E \\
\mid\; & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
\mid\; & \texttt{let } x = E \texttt{ in } E \\
\mid\; & \texttt{proc } x \; E \\
\mid\; & E \; E
\end{aligned}
$$