COSC 4780 - Principles of Programming Languages - Spring 2022

Lecture 7: Introduction to Type Checking

Finley McIlwaine

University of Wyoming

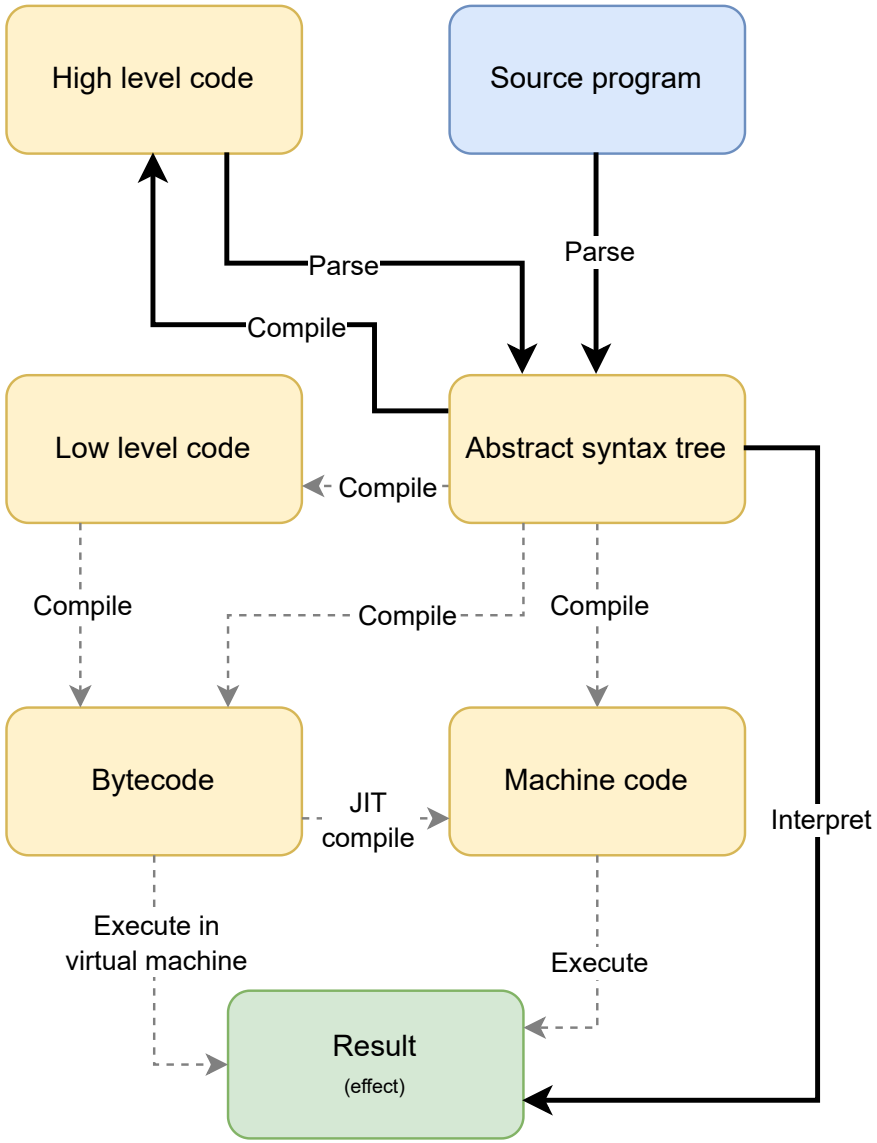
fmcilwai@uwyo.edu

February 10, 2022

Review - Zoom Out

Let's zoom out again. What arrows did the circuit emulator from last time cover?

It really only covered the source code → AST → interpret path. But it was a bit more involved than our other interpreters, because we had to track the signals on all of the wires!



Review

Today, we're going to introduce the concept of type checking. We'll motivate our journey today by looking at the simple imperative language from last week and observing the effects that type systems can have on programs in the language.

We will also take a look at some very important concepts that will be prominent in nearly everything we do for the rest of the course.

Types

What is a type? The specific notion varies across languages, however there is a general consensus: **A type is a set of values**. A large part of what determines how we write programs is the facilities a language provides for defining/using operations on those values.

Type Systems

Three general categories of type systems are prominent in modern programming languages:

- Dynamically typed: All or almost all type checking is done during the *execution* of a program.
  - Python
  - JavaScript
- Statically typed: All or almost all type checking is done during the *compilation* of a program.
  - Java
  - C++
  - Go
  - Haskell
- Untyped: No type checking at all
  - Some assembly languages
  - Sometimes mixed up with dynamic typing

Type Systems

We're going to explore variants of each of these systems by implementing them in our small imperative language from last week.

But what does it mean to "implement a type system"?

Type Systems

We're going to explore variants of each of these systems by implementing them in our small imperative language from last week.

But what does it mean to "implement a type system"? It means we implement a type checker or interpreter that enforces that type system. The type checker or interpreter itself may be informed by a formal specification of the system, but in this case we're going to remain informal.

Type Systems

The previous grammar for the imperative language we developed had types built in to the non-terminals for expressions. This is a little atypical. It is more common to have a single non-terminal for expressions and then let the type system determine what types of expressions can go where. So we'll need to change the grammar that we developed last week. The result looks something like this:

```
...
EEq.  Exp1 ::= Exp1 "==" Exp2 ;
ELeq. Exp1 ::= Exp1 "<=" Exp2 ;
ENot. Exp2 ::=      "!"  Exp4 ;
ESub. Exp2 ::= Exp2 "-"  Exp3 ;
EAdd. Exp2 ::= Exp2 "+"  Exp3 ;
...
```

Untyped

Starting with an untyped type system... how do we implement a type checker for an untyped language? We don't! The way we'll illustrate an untyped system is with an interpreter that simply assumes everything is well typed.

Let's look at what an interpreter like this looks like in Java...

Dynamically Typed

A dynamically typed system looks remarkably similar to an untyped system, which is why they are commonly referred to as the same thing. A dynamically typed system, however, will not be as hands-off as an untyped system. It will actually perform some explicit checks of the types of expressions during execution.

Let's see how implementing this changes the untyped interpreter...

Statically Typed

The previous two systems performed their type checking (if any) at runtime. With static typing, we move type checking to *compile time*, which means we don't go straight from parsing to interpretation! We add an intermediate compiler stage that type checks the abstract syntax tree before any evaluation.

The type checker should aim to find any type errors and prevent the program from executing at all if it finds any. If the program passes type checking, the interpreter can omit the runtime checks that we saw in the dynamically typed system, since the type checker has guaranteed the types are okay!

Statically Typed

But what exactly are the type errors that our type checker wants to find? In general, most type checkers aim to guarantee the following:

- All operators/functions are applied to the correct types
- All expressions *have* a type
- All variables are initialized before use

Some aspects of our language make the type checker easier to implement. For example: No functions! Also, we allow variables to be reinitialized/redefined as values of different types. This makes the difference between initialization/assignment very small, and renders the distinction slightly pointless, but that's okay.

Statically Typed

An important fact about type checking is that it is a mix of two tasks:

- Type checking: Checking that a given expression has some expected type.
- Type inference: Finding the type of a given expression.

Type checkers do both of these. For example, we always check that the conditional expression in an if statement has boolean type. That is type checking. In order to discover the type of that expression so that we can check it, we use type inference.

Statically Typed

Once a program passes type checking, our interpreter can just assume that all expressions are well-typed! Program execution becomes a lot safer since many erroneous programs are guaranteed to be caught by the type checker. In fact, we can just use the untyped interpreter again!

Let's see what the type checker looks like, and check that the interpreter is again just the untyped interpreter...

Formal Type Systems

Our discussion throughout this lecture has been rather informal. Hopefully you've found that none of these aspects of interpreters and type checkers are too surprising. As seasoned programmers, these ideas are probably rather intuitive at this point!

However, it is very common to specify type systems and semantics (interpreters) of programming languages mathematically before converting them into implementations. Next time, we'll start studying the notations and methods used to do this. The purpose of this lecture was to give us some concrete implementations that we can refer back to, which should make the math easier to deal with.

The End