ECE251: Programming Languages & Translators Fall 2010

Lecture 22 — November 10, 2010

Patrick Lam *version 1*

Type Checking

A key part of most compilers is the *type checker.* It ensures that values and variables are used correctly. The textbook contains an exhaustive discussion of types in Chapter 7. I'll mention some of the most important points; notably, enough so that you can implement a type checker.

Useful terms

Let's start by defining type-related terms.

- A *type* is a summary of the possible values of an expression. Every evaluable expression must have a unique type.
- A *type system* consists of a set of rules for assigning types to expressions, as well as a type checker.
- *Type checking* verifies that operators and functions are only used with parameters of the appropriate types. (The book mentions type equivalence, compatibility, and inference. We'll implicitly see them in our type checker.)
- A language implementation is *strongly-typed* if its compiler guarantees that programs will never violate the rules of the type system.
- A *static type system* does most of its checks at compile-time, while a *dynamic type system* does most of its checks at run-time.
- Type *synthesis* computes types based on declared types for variables, while type *inference* computes types based on the operations that the program contains.

Let's look at some examples of all of these terms.

Types. Examples of primitive types in Java:

Other languages have slightly different sets of primitive types. For instance, `string` is a primitive type in PHP.

Many languages support user-defined types. What are examples of user-defined types in C?

In Java, types can be related to each other in a type hierarchy, as we've seen before.

Type checking examples: primitive types. Let's manually work through Java type checking on a series of examples. Draw the AST and assign types to each of the parts of the statement or expression.

- `5 * 9 + 22 / 354`
- `2 + 'b'`
- `2 + "foo"` (the coercoin is actually quite complicated)
- `if (i > 5) System.out.println(i + 2); else x = i * 9;`

Object types and polymorphism. We've seen this before, but now we type check.

```
class A { int f; }
class B extends A {
  public void m(int x) { }
  public void m(Object o) { }
}
```

```
void foo() {
  A a = new A(), b = new B();
  ((B)b).m(45);
  ((B)b).m(a);
  a.m(2);
  System.out.println(b.f);
}
```

Type coercion. We've seen in the above examples that types don't always match up exactly. Compilers need to be able to deal with this to make a language usable.

We therefore automatically convert between types. In general, there are two kinds of possible conversions, *widening* and *narrowing*. You can always do widening conversions, since they don't lose data, and many compilers do these conversions automatically. Narrowing conversions are allowed, but you should beware. (Contrast that to, say, converting an `String` to a `List`. You can't.) The programmer usually has to write the narrowing conversion explicitly, e.g. `(int)4.2`.

(from *Compilers*, Aho, Lam, Sethi and Ullman, Figure 6.25:)

$$\begin{tikzcd}
& \text{double} \arrow[d, dash] & \\
& \text{float} \arrow[d, dash] & \\
& \text{long} \arrow[d, dash] & \\
& \text{int} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{short} \arrow[d, dash] & & \text{char} \\
\text{byte} & &
\end{tikzcd}$$

(a) Widening conversions

$$\begin{tikzcd}
& \text{double} \arrow[d] & \\
& \text{float} \arrow[d] & \\
& \text{long} \arrow[d] & \\
& \text{int} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{char} \arrow[r, leftrightarrow] \arrow[rr, leftrightarrow, bend right] & \text{short} \arrow[r, leftrightarrow] & \text{byte}
\end{tikzcd}$$

(b) Narrowing conversions