**ECE251: Programming Languages & Translators** Fall 2010

# Lecture 23 — November 11, 2010

*Patrick Lam* *version 1*

**Doing a type coercion.** Let's write some code to actually carry out a type coercion. It's also a good example of generating three-address code from an AST.

```
int x(int y) { return y; }
double f = x(4) + 2.2d;
```

I also discussed autoboxing and unboxing as further examples of coercions.

**Type equivalence.** In Java, we have a pretty clear idea of when two types are equal—they have the same name[1]. So, if two variables both have `String` type, then they have equivalent types. We can call this *name equivalence.*

Other languages use different rules for type equivalence. For instance, ML uses structural equivalence; two types are equivalent if they contain the same components.

**Type inference.** By the way, even though the languages you've seen so far require you to write out all the types, you don't, strictly speaking, need to do that. Type inference can infer most of the types in your program. ML allows you to enter types when you want to and it'll infer everything else.

## Formalism for Type Checking: $\vdash$

Usually we use type rules to express type systems. Your goal, in implementing a type checker, is to come up with code that implements these rules. Here are some examples of type rules.

$$\frac{}{n : \text{int}} \qquad \frac{}{\textsf{true} : \text{bool}} \qquad \frac{}{\textsf{false} : \text{bool}} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}} \qquad \frac{e : \text{bool} \quad e_1 : \text{int} \quad e_2 : \text{int}}{\text{if } e \text{ then } e_1 \text{ else } e_2 : \text{int}}$$

and a derivation:

$$\frac{\dfrac{\dfrac{}{2\text{:int}} \quad \dfrac{}{3\text{:int}}}{2+3\text{:int}} \qquad \dfrac{}{4\text{:int}}}{2+3+4 : \text{int}}$$

[1]It's actually wrong. A type is determined by its name as well as its class loader, so you can actually have two distinct types, both called `X`. It's confusing and beyond the scope of this class.

Let's look at what our type system is doing for us.

- prevents adding boolean values;
- accepts 5/0
- may unjustly reject some programs: `if true then 1 else false`

**Environments.** This isn't very interesting so far, since it doesn't say anything about variables. This is where the $\vdash$ comes in. We introduce the environment, $\Gamma$, which you implement using the symbol table.

An environment $\Gamma$ contains a list of type assumptions, e.g. $x_1 : t_1, x_2 : t_2, \ldots$. Then we say write $\Gamma \vdash e : t$ and say $\Gamma$ entails that $e$ has type t, if $e$ has type $t$ under the assumptions in $\Gamma$.

For example:

$$x : \text{int}, y : \text{int} \vdash x : \text{int}$$
$$x : \text{int}, y : \text{int} \vdash x + y : \text{int}$$

When we have a variable declaration `T x`, then we add `x:T` to the environment. You can also add functions to the environment.

$$x : \text{int}, f : \text{int} \to \text{int} \vdash f(x) : \text{int}$$

Functions may be overloaded, as we saw last time.

$$x : \text{int}, id : \text{int} \to \text{int}, id : \texttt{Object} \to \texttt{Object} \vdash id(x) : \text{int}$$
$$o : \texttt{Object}, id : \text{int} \to \text{int}, id : \texttt{Object} \to \texttt{Object} \vdash id(o) : \texttt{Object}$$

Introducing rules for object-oriented languages complicates matters a bit. We'll say that writing down the rules for such languages is beyond the scope of this class.

## Implementation issues

Implement type rules as an AST visitor. Keep a type for each AST node. When you get to a node with children (say, `PlusExpr`), then you check that its children have appropriate types.

```
class Expr {
  Type getType();
}

class PlusExpr extends Expr {
  Expr e1, e2;
  boolean typecheck() {
    if (e1.getType() == Type.INT &&
        !e1.getType().equals(e2.getType()))
      reportError("adding an int to a non-int");
  }
}
```