We have seen that types can be complex, and therefore so can type checking. So we would like to have a concise way of specifying how to do type checking. This is the role of a *static semantics*, which defines how to ascribe types to terms.

In code we said that that we could implement type checking recursively as a method `typeCheck` on AST nodes, something like the following:

```
class Expr {
    Type typeCheck(Context c);
}
```

Formally, we will express the idea that `t = e.typeCheck(c)` with a *typing judgment* written $\Gamma \vdash e : t$. In this judgment, Γ is the typing context (symbol table), e is the term to be type-checked, and t is its type in the given typing context. We read the judgment as "Γ proves e has type t."

A typing context Γ is a finite (and possibly empty) map from variable names to types, which we write as $x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n$. As a shorthand, the judgment $\vdash e : t$ means that e has type t in the empty typing context. For example, we have $\vdash 5 : \texttt{int}$, because 5 is an integer in every typing context.

1 Inference rules

A *type system* is a set of types, plus a set of inference rules for deriving typing judgments; in other words, a type system includes a proof system for typing judgments.

An example of a typing rule is the following inference rule:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \text{ (PLUS)}$$

The way to interpret this rule is this: if we can show that e_1 is an `int` in some context Γ, and we can show e_2 is an `int` in that context, then in the same context Γ we can show $e_1 + e_2$ is an `int`.

The judgment below the line is the *conclusion*. The judgments above the line are *premises*. In general we may write additional conditions above the line that must be true to derive the conclusion; these non-judgment conditions are called *side conditions*. If a rule has no premises, we call it an *axiom*. On the side of the rule we sometimes write the name of the rule (PLUS) so we can talk about it elsewhere.

Examples of axioms are the following. First, an axiom for the type of an integer literal n:

$$\frac{}{\Gamma \vdash n : \texttt{int}} \text{ (INTLIT)}$$

Another axiom lets us derive the type of a variable by finding it in the current typing context. This axiom has a side condition but has no true premises, hence is an axiom.

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \text{ (VAR)}$$

An inference rule must express reasoning that is correct under all consistent substitutions of syntactic expressions (drawn from the correct set) for metavariables appearing in the inference rule. That is, an inference rule is implicitly universally quantified over its metavariables (such as $e_1, e_2, and \Gamma$ in the rule PLUS). Since axioms have no premises, they must state things that are true no matter what.

The job of a type checker is to determine whether the typing rules can be used to construct a *derivation* of a typing judgment for the given term. A derivation a tree of instances of inference rules, showing how to start from axioms and derive the final judgment. For example, we can prove $x : \texttt{int} \vdash x + 2 : \texttt{int}$ as follows, using the inference rules we have already seen:

$$\dfrac{\dfrac{}{x : \texttt{int} \vdash x : \texttt{int}}\ (\textsc{Var}) \quad \dfrac{}{x : \texttt{int} \vdash 2 : \texttt{int}}\ (\textsc{IntLit})}{x : \texttt{int} \vdash x + 2 : \texttt{int}}\ (\textsc{Plus})$$

To see how we get this derivation, consider the use of the rule PLUS. We get the corresponding step in this derivation by applying the substitution $e_1 \mapsto x, e_2 \mapsto 2, \Gamma \mapsto x : \texttt{int}$ to the inference rule.

2 Inference rules for an Xi-like language

We can also type-check statements in a language like Xi. Statements don't return any interesting value, but we can think of them as computing a value of *unit* type. A unit type is a type with only one value. If a computation produces this value, it merely means that the computation terminated. The declaration `void` in Java, used as a return type of methods, is essentially a declaration of unit type. We write `unit` for the unit type, as in OCaml. The typing judgment $\Gamma \vdash s$: `unit` means for us that s is a well-typed statement, though the notation is not essential—we could equally well invent a judgment written $\Gamma \vdash s$, or alternatively, $\Gamma \vdash s$ `stmt`.

Now we can write rules for type-checking `if` and `while`:

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s_1 : \texttt{unit} \quad \Gamma \vdash s_2 : \texttt{unit}}{\Gamma \vdash \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \texttt{unit}}\ (\textsc{If}) \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s : \texttt{unit}}{\Gamma \vdash \texttt{while}\ (e)\ s : \texttt{unit}}\ (\textsc{While})$$

$$\frac{\Gamma \vdash \{s\} : \texttt{unit}}{\Gamma \vdash s : \texttt{unit}}\ (\textsc{Block}) \qquad \frac{\Gamma \vdash s_1 : \texttt{unit} \quad \Gamma \vdash s_2 : \texttt{unit}}{\Gamma \vdash s_1; s_2 : \texttt{unit}}\ (\textsc{Seq})$$

$$\frac{\Gamma \vdash x : t \quad \Gamma \vdash e : t}{\Gamma \vdash x = e : \texttt{unit}}\ (\textsc{Assign}) \qquad \frac{\Gamma \vdash e_1 : t[] \quad \Gamma \vdash e_2 : \texttt{int} \quad \Gamma \vdash e_3 : t}{\Gamma \vdash e_1(e_2) = e_3 : \mathit{unit}}\ (\textsc{ArrAssign})$$

3 Implementing a type checker

A key property of these rules is that they are *syntax-directed*: given a statement, we know which rule must be used to derive a the typing judgment for the statement. This means that we can implement a type checker as a simple recursive traversal over the AST. If the rules were not syntax-directed, we might have to search for a derivation, which could take time exponential in the height of the derivation.

For example, consider the rule IF. We can implement type checking of this statement as a method `typeCheck` that recursively invokes the same method on subexpressions, to satisfy premises. Side conditions are checked by non-recursive tests.

```
class If extends Stmt {
    Expr guard;
    Stmt consequent, alternative;
    void typeCheck(Context c) {
        Type tg = guard.typeCheck(c); // premise 1
        if (!tg.equals(boolType))
            throw new TypeError("guard must be boolean", guard.position());
        consequent.typeCheck(); // premise 2
        alternative.typeCheck(); // premise 3
    }
}
```

4 Top-level context

We need a top-level context that can includes bindings for all of the functions in the program. In an object-oriented language, it would also map each class name to some representation of the class. If we assume that the program is a sequence of declarations

$$f_1(x_1 : t_1) : t'_1 = s_1 \ldots f_n(x_n : t_n) : t'_n = s_n$$

then the top-level context we want is:

$$\Gamma_0 = f_1 : t_1 \to t'_1, \ldots f_n : t_n \to t'_n$$

We can use this context to type-check function calls:

$$\frac{(f : t \to t') \in \Gamma \quad \Gamma \vdash e : t}{\Gamma \vdash f(e) : t'} \text{ (APPLY)}$$

Of course, we also need to type-check function bodies to make sure that they satisfy the contract implied by their signatures. One trick we use is to record the return type of the function in a special name ρ, which allows us to type-check `return` statements. For

$$\Gamma_0, \rho : t'_i \vdash s_i : \texttt{unit}$$

The `return` statement is type-checked as follows:

$$\frac{}{\Gamma, \rho : t \vdash \texttt{return}\ t : \texttt{unit}} \text{ (RETVAL)} \qquad \frac{}{\Gamma, \rho : \texttt{unit} \vdash \texttt{return} : \texttt{unit}} \text{ (RET)}$$

One nice thing about type systems is that they let us clearly and concisely specify the job of semantic analysis. Another important use is that a formal type system allows us to *prove* that a statically typed language is strongly typed. However, showing you how to do this is a job for a different course.