Modern languages have module mechanisms to support modular programming. For example, Java has classes and OCaml has modules. Modules promote loose coupling between different parts of the program, by hiding internal implementation details and by restricting access to internal mechanisms. They also support *separate compilation*, in which the program is composed of separately compiled components that are later linked together.

Module mechanisms complicate type checking because they introduce multiple namespaces that must be used at the right times to resolve names of variables, functions and types. Modules can also be parameterized on type variables, can introduce abstract type variables, and can be recursive. These features make type checking even trickier.

1 Symbol tables for modules

Modules define a set of named resources that can be used by other components of the program. The resources exported by the module are the module's *interface*. Resources may include variables, constants, functions, and types. For each module used by the current code, the compiler needs an environment corresponding to its interface, in order to check uses of named resources and to generate code using them. These environments are essentially symbol tables.

When compiling a module, the compiler must also know about names that are defined internally to the module but are not accessible outside (e.g., Java's "private" fields and methods). Therefore, within the module, we have an internal environment that is larger than the one that is exported. In many languages, such as OCaml or C, a module's interface is declared separately from the implementation, and there are really two separate environments. The compiler must check these two environments to ensure they conform—they must agree on the type and kind of all names that appear in the interface.

Sometimes conformance is deferred until link- or run-time, as shown in Figure 1. In the figure, the file `A.java`, containing the classes A and B, is compiled against the interface of the class C. The interface of C is obtained automatically from the class file for C, where the compiler puts it. In other languages, the interface might be be a separate ASCII source file.

During compilation, the compiler builds a top-level environment that maps globally visible names such as `A`, `B`, and `C` to compile-time representations of the corresponding classes. Typically these would be objects roughly corresponding to the class objects that the Java runtime system maintains; they would contain their own environments mapping names of class members to the corresponding types (and other information). For example, there is an `A` environment mapping the names `x` and `f` to representations of their types. Similarly, the `B` environment maps the name `y` to its type, which is `C`, for which there is an environment that can be used to resolve the type of an expression like `y.g`.

At run time, there is no guarantee that the actual C code implements the interface that was read at compile time. For type safety, a conformance check must be done. Therefore, the class file for class A contains type signatures for the parts of C that are used; these are checked when used to make sure they agree with the actual C class file that is loaded at run time.

2 Abstract types

Modules can hide types completely. For example, in OCaml we can write a module interface `M` that announces the existence of a type `t` without revealing what the type is:

```
module type M = sig
    type t
    val f : T -> T
end
```

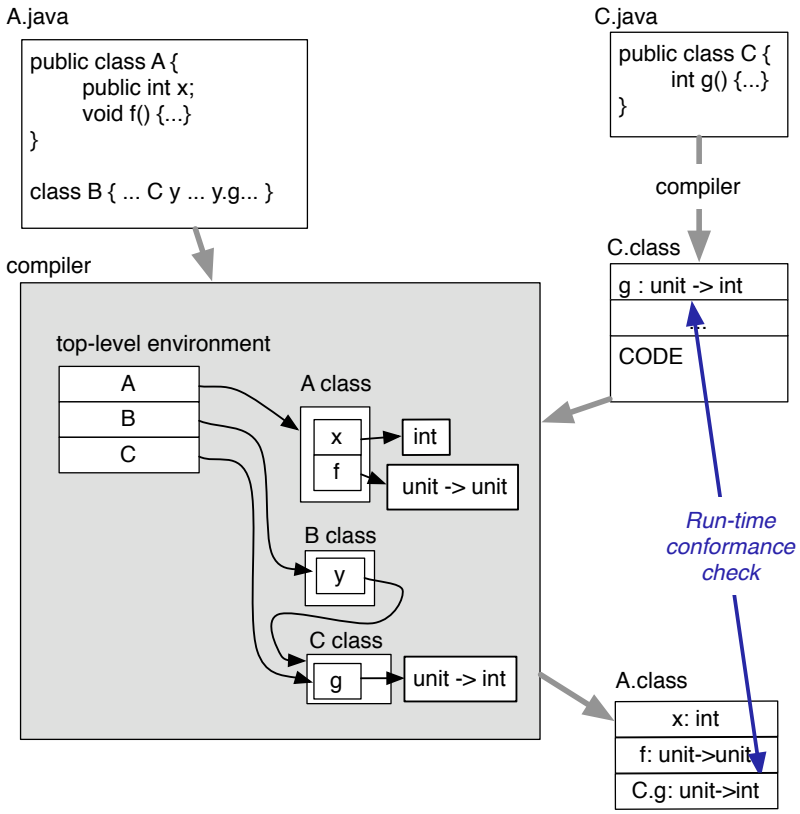

Figure 1: Separately compiling Java classes

Here is it necessary to introduce a binding in the environment for `M` that says `t` is a type without identifying which type it is.

Since the compiler doesn't know which type `t` is, it doesn't know how much storage is required to implement `t`. Unless the linker can later patch the code using a `t`, the compiler must generate code that works regardless of the size of `t`. This is usually achieved by through an indirection, relying on the fact that pointers have the same size no matter what they point to. As far as client code is concerned, the type `t` is a word-sized pointer, and the client code doesn't have to know how big the chunk of memory is that the pointer points to.

3 Representing types

In Figure 1, we might wonder why the B environment maps `y` to the C class object rather than to the abstract syntax tree node that was generated during parsing.

The strategy of using AST nodes to represent types is feasible, but has some gotchas. Consider the following Java-like example, in which we have nested classes:

```
class A extends B {
    int x;
    D y;
    class C { int z; ... y.w ... }
}
class D { C w }
class C { Object z; }
```

The type of the term `y.w` is `C`, but it is the top-level `C`. However, the name `C` within the context where the term `y.w` occurs refers to the nested class `C`. If we are not careful about how we represent and resolve types, the compiler will think that `y.w.z` is an `int` rather than an `Object`, allowing run-time type errors.

In general, the AST strategy for representing types makes it necessary to represent a type not as just an AST, but as a *pair* of an AST *plus* the symbol table that should be used to resolve any identifiers that are found within the type. For example, we can represent the type of `y.w` as the type expression `C`, plus the symbol table that was being used at the point where the term `y.w` occurred.

4 Handling recursion with multiple passes

Alternatively, we can try to resolve all type names as shown in Figure **??**, to point directly to the type objects with environments that let the compiler look up names directly and efficiently. This creates challenges for dealing with recursive modules, because it suggests we may need to resolve class names to class objects before the class objects are created. For example, class A might refer to class B, so while resolving type declarations in A, there needs to already be some kind of class object for B. And vice-versa.

To break the circularity, we can do type checking in multiple passes over the AST:

1. First pass: Create type objects. Find all classes that are referenced in the source code and create empty class objects for all. classes. This handles recursion among types.
2. Second pass: Construct method signatures. Walk over all methods or functions and construct their types. This handles recursion among methods.
3. Third pass: Type-check the code, using the method signatures and type objects created in previous passes.

5 Visitors

(See slides)