Dependency Analysis

Swift's intra-module dependency analysis is based on a "provides" / "depends" system, which is ultimately trying to prove which files do not need to be rebuilt. In its simplest form, every file has a list of what it "provides" and what it "depends on", and when a file is touched, every file that "depends on" what the first file "provides" needs to be rebuilt.

The golden rule of dependency analysis is to be conservative. Rebuilding a file when you don't have to is annoying. *Not* rebuilding a file when you *do* have to is tantamount to a debug-time miscompile!

Kinds of Dependency

There are four major kinds of dependency between files:

- `top-level`: use of an unqualified name that is looked up at module scope, and definition of a name at module scope. This includes free functions, top-level type definitions, and global constants and variables.
- `nominal`: use of a particular type, in any way, and declarations that change the "shape" of the type (the original definition, and extensions that add conformances). The type is identified by its mangled name.
- `member`: a two-part entry that constitutes either *providing* a member or *accessing* a specific member of a type. This has some complications; see below.
- `dynamic-lookup`: use of a specific member accessed through `AnyObject`, which has special `id`-like rules for member accesses, and definitions of `@objc` members that can be accessed this way.

The `member` dependency kind has a special entry where the member name is empty. This is in the "provides" set of *every file that adds non-private members* to a type, making it a superset of provided `nominal` entries. When listed as a dependency, it means that something in the file needs to be recompiled whenever *any* members are added to the type in question. This is currently used for cases of inheritance: superclasses and protocol conformances.

The "provides" sets for each file are computed after type-checking has completed. The "depends" sets are tracked by instrumenting lookups in the compiler. The most common kinds of lookups (qualified name lookup, unqualified name lookup, and protocol conformance checks) will already properly record dependencies, but direct lookups into a type or module will not, nor will dependencies not modeled by a query of some kind. These latter dependencies must be handled on a case-by-case basis.

Note:

The compiler currently does not track `nominal` dependencies separate from `member` dependencies. Instead, it considers every `member` dependency to implicitly be a `nominal` dependency, since adding a protocol to a type may change its members drastically.

External Dependencies

External dependencies, including imported Swift module files and Clang headers, are tracked using a special `depends-external` set. These dependencies refer to files that are external to the module. The Swift driver interprets this set specially and decides whether or not the cross-module dependencies have changed.

Because every Swift file in the module at least has its imports resolved, currently every file in the module has the same (complete) list of external dependencies. This means if an external dependency changes, everything in the module is rebuilt.

Complications

- A file's "provides" set may be different before and after it is compiled – declarations can be both added and removed, and other files may depend on the declarations that were added and the declarations that were removed. This means the dependency graph has to be updated after each file during compilation. (This is also why reusing build products is hard.)
- If a build stops in the middle, we need to make sure the next build takes care of anything that was scheduled to be rebuilt but didn't get rebuilt this time.