Request-Evaluator

The request-evaluator is a new architectural approach for the Swift compiler that attempts to provide infrastructure for fine-grained tracking and visualization of dependencies, separately caching the results of computation within the compiler, and eliminating mutable state, particular from the Abstract Syntax Tree (AST). It is intended to address a number of problems that arise in the Swift compiler:

- Type checking (as well as later stages) tend to depend on mutable AST state, which is hard to reason about: which mutable state contributed to the result of a type check? How was it computed?
- Type checking, particularly in targets with more source files, is intended to be "lazy": the compiler will try to type check only what is needed to compile the given source file, and nothing more. Combined with mutable state, we often end up with compiler bugs that only manifest in multi-file targets, which are (1) the norm for real-world development, but (2) far less friendly for compiler writers writing test cases.
- Lazy type checking uses an ad hoc set of callbacks captured in the lazy resolver. This ad hoc recursion can lead to cycles in the type-checking problem (e.g., to compute A we need to compute B, which depends on A): with mutable state, it's easy to cause either wrong results or infinite recursion.
- Because it's not clear which state is set by a given callback in the `LazyResolver`, the callbacks there tend to be very coarse-grained, e.g., "resolve everything about a declaration that might be used by a client." This performs too much work in multi-file targets (making the compiler slower than it should be) and contributes to more cycles (because coarse-grained queries are more likely to be cyclic where finer-grained ones might not be).

The request-evaluator attempts to address these issues by providing a central way in which one can request information about the AST (e.g., "what is the superclass of the given class?"). The request-evaluator manages any state in its own caches, and tracks the dependencies among these requests automatically, identifying (and reporting) cycle dependencies along the way. It is designed for incremental adoption within the compiler to provide a smooth path from ASTs with mutable state, and provides useful debugging and performance-monitoring utilities to aid in its rollout.

Requests

At the core of the request-evaluator is a "request", which is a query provided to the "evaluator", which will return the result of servicing the request. For example, a request could be "get the superclass of class `Foo`" or "get the interface type for function `bar(using:)`". Each of these request kinds would produce a type as its result, but other requests could have other result types: "get the formal access kind for declaration `x`" would return a description of the access level, or "get the optimized SIL for function `bar(using:)`" would produce a `SILFunction`.

Each "request" is described by, effectively, a description of a computation to perform. A request is a value type that is capable of comparing/hashing its state, as well as printing its value for debugging purposes. Each request has an evaluation function to compute the result of the request. Only the evaluator itself will invoke the evaluation function, memoizing the result (when appropriate) for subsequent requests.

All existing requests inherit the `SimpleRequest` class template, which automates most of the boilerplate involved in defining a new request, and mostly abstracts away the handling of the storage/comparison/hashing/printing of the inputs to a request. Some example requests for access control are provided by `AccessRequests.h`.

Dependencies and Cycles

Each request can issue other requests, also through the evaluator. The evaluator automatically tracks such dependencies (by keeping a stack of the currently-active request evaluations). This information is valuable for a few reasons. First, it can help with debugging both correctness and performance, allowing one to visualize the dependencies evaluated when compiling a program. The current protocol has both full-graph visualization (via GraphViz output) and dependencies-for-a-single-request visualization (via a tree-like dump). Second, it ensures that we can detect cyclic dependencies (e.g., a cyclic inheritance hierarchy) correctly, because they show up as cycles in the dependency graph, allowing for proper diagnostics (for the user) and recovery (in the compiler). Third, it can eventually be leveraged to enable better incremental compilation--providing the ability to discover what information affected a particular type-checking decision, and propagate the effects of a specific change through the dependency graph.

The frontend option `-debug-cycles` will provide debugging output for any cycle detected while processing the given source files. For example, running the `circular_inheritance.swift` test from the Swift repository using this flag, e.g.,

```
$ swiftc -frontend -typecheck -debug-cycles test/decl/class/circular_inheritance.swift
```

provides a debugging dump that illustrates the one of the dependency cycles via a textual tree dump:

```
===CYCLE DETECTED===
 `--SuperclassTypeRequest(circular_inheritance.
(file).Outer2@test/decl/class/circular_inheritance.swift:38:7)
     `--InheritedTypeRequest(circular_inheritance.
(file).Outer2@test/decl/class/circular_inheritance.swift:38:7, 0)
         `--SuperclassTypeRequest(circular_inheritance.
(file).Outer2@test/decl/class/circular_inheritance.swift:38:7) (cyclic dependency)
```

Within the compiler, the core `Evaluator` class provides dumping routines such as `dumpDependencies()`, so one can see the dependencies for any request as part of a debugging session. It uses the same textual dump format as `-debug-cycles`.

The request-evaluator has the ability to emit "real" Swift diagnostics when it encounters a cycle, but they are currently disabled because they fire too often in real-world code. Once enough of the cyclic dependencies in well-formed code have been removed, such that the only errors come from ill-formed code, these diagnostics will be enabled. For example, they will be used to detect ill-formed programs such as:

```
typealias A = B
typealias B = A
```

Caching

The request-evaluator contains a cache of any requests that have already been evaluated (and have opted into caching). The cache in the evaluator is meant to (eventually) replace the mutable state on the AST itself. Once it does so, it becomes possible to throw away cached information and compute it again later, which could form a basis for incrementally updating a program in response to changes to the source code. Getting here depends on *everything* going through the evaluator, however.

Until then, the request-evaluator lives in a compiler that has mutable ASTs, and will for the foreseeable future. To cope with this, requests can opt for "separate" caching of their results, implementing a simple protocol to query their own cache (`getCachedResult`) and set the value for the cache (`cacheResult`). For now, these functions can directly modify state in the AST, allowing the requests to be mixed with direct mutation of the state. For each request, the intent is to make all state access go through the evaluator, but getting there can be an incremental process.

Incremental Dependency Tracking

Request evaluation naturally captures the dependency structure of any given invocation of the compiler frontend. In fact, it captures it so well that the request graph trace generated by a certain kind of lookup request can be used to completely recover the information relevant to the Swift compiler's incremental compilation subsystem. For these select dependency-relevant requests, we can further subdivide them into so-called *dependency sources* and *dependency sinks*. A dependency source is any (usually high-level) request that introduces a new context under which dependencies can be registered. Currently, these are the requests that operate at the level of individual source files. A dependency sink is any (usually lower-level) request that executes as a sub-computation of a dependency source. Any names that are dependency-relevant are then registered against trackers in the active dependency source (file). Using this, the evaluator pushes and pops sources and sinks automatically as request evaluation proceeds, and sink requests pair automatically to source requests to write out dependency information.

To see an example of this in action, suppose the following chain of requests is currently being evaluated:

```
TypeCheckSourceFileRequest(File.swift) -> ... -> DirectLookupRequest(Foo, "bar")
```

Because `TypeCheckSourceFileRequest` is a dependency source, `File.swift` is automatically set as the active dependency scope. When the subsequent dependency sink `DirectLookupRequest` completes, it will automatically write the fact that `Foo.bar` has been looked up in the appropriate referenced name tracker in the active source - `File.swift`.

To define a request as a dependency source, it must implement an accessor for the new active scope (`readDependencySource`). To define a request as a dependency sink, it must implement a function that writes the result of evaluating the request into the current active source (`writeDependencySink`).

Open Projects

The request-evaluator is relatively new to the Swift compiler, having been introduced in mid-2018. There are a number of improvements that can be made to the evaluator itself and how it is used in the compiler:

- The evaluator uses a `DenseMap<AnyRequest, AnyValue>` as its cache: we can almost certainly do better with per-request-kind caches that don't depend on so much type erasure.
- Explore how best to cache data structures in the evaluator. For example, caching `std::vector<T>` or `std::string` implies that we'll make copies of the underlying data structure each time we access the data. Could we automatically intern the data into an allocation arena owned by the evaluator, and vend `ArrayRef<T>` and `StringRef` to clients instead?
- Cycle diagnostics are far too complicated and produce very poor results. Consider replacing the current `diagnoseCycle` / `noteCycleStep` scheme with a single method that produces summary information (e.g., a short summary string + source location information) and provides richer diagnostics from that string.
- The `isCached()` check to determine whether a specific instance of a request is worth caching may be at the wrong level, because one generally has to duplicate effort (or worse, code!) to make the decision in `isCached()`. Consider whether the `evaluator()` function could return something special to say "produce this value without caching" vs. the normal "produce this value with caching".
- Try to eliminate more boilerplate from subclasses of `SimpleRequest`. We are going to have a *lot* of requests.
- Each request supports a simple printing operation (via `simple_display`): implement a parsing scheme so we can take the output of `simple_display` and parse it into a request. Build a command-line testing interface so we can parse a source file and make a specific request (e.g., `SuperclassTypeRequest("Foo")`) to see the results.

- Port more mutable-state AST queries over to requests. This often requires a lot of refactoring!
- Port higher-level queries (e.g., those that come from SourceKit) over to the request-evaluator, so we can see the dependencies of a given SourceKit request for testing and performance tuning.

Prior art

Rust's compiler went through a similar transformation to support demand-driven compilation. We should learn from their experience!