orphan:

Declaration Type Checker

Contents

Purpose

This document describes some of the problems with our current "declaration" type checker, which validates and assigns types to the various declarations in a Swift programs. It analyzes the dependencies within the Swift type system that need to be reflected within the implementation and proposes an alternative architecture for the type checker that eliminates these problems.

Problems with the Current Approach

The current declaration type checker is the source of a large number of Swift bugs, including crashes on both well-formed and ill-formed code, different behavior depending on the order of declarations within a file or across multiple files, infinite recursion, and broken ASTs. The main issues are:

Conceptual phases are tangled together: We have a vague notion that there are phases within the compiler, e.g., extension binding occurs before name binding, which occurs before type checking. However, the implementations in the compiler don't respect phases: extension binding goes through type validation, which does both name binding and type checking. Name lookup attempts to do type checking so that it can establish whether one declaration shadows another.

Unprincipled recursion: Whenever type checking some particular declaration requires information about another declaration, it recurses to type-check that declaration. There are dozens of these recursion points scattered throughout the compiler, which makes it impossible to reason about the recursion or deal with, e.g., recursion that is too deep for the program stack.

Ad hoc recursion breaking: When we do encounter circular dependencies, we have scattered checks for recursion based on a number of separate bits stashed in the AST: `isComputingRequirementSignature()`, `isComputingPatternBindingEntry()`, `isComputingGenericSignature()`/`hasComputedGenericSignature()`, etc. Adding these checks is unprincipled: adding a new check in the wrong place tends to break working code (because the dependency is something the compiler should be able to handle), while missing a check permits infinite recursion to continue.

Type checker does too much work: validating a declaration is all-or-nothing. It includes computing its type, but also checking redeclarations and overrides, as well as numerous other aspects that a user of that declaration might not care about. Aside from the performance impact of checking too much, this can introduce false circularities in type-checking, because the user might only need some very basic information to continue.

Phases of Type Checking

Type checking for Swift can be divided into a number of phases, where each phase depends on information from the previous phases. A phase may depend on information from another declaration also computed in that phase; such cases need to manage dependencies carefully to detect and diagnose circular dependencies. The granularity of phases can differ: earlier phases tend to more global in nature, while later phases tend to be at a much finer granularity (per-declaration, per-`TypeRepr`, etc.). Generally speaking, we want to minimize the amount of work the type checker performs by moving a given declaration only up to the phase that's absolutely required to make a decision.

Parsing: Parsing produces untyped ASTs describing the structure of the code. Name lookup can find module-scope names in the current module, but any lookup for nested names or names from other modules is unavailable at this point. This is a global phase, because we need to have parsed at least the top-level declarations of each file to enable name lookup.

Import resolution: Resolve the import declarations within a source file to refer to modules. Name lookup can now find module-scope names in the current module or any imported module. This is a file-scoped phase, because one need only resolve the imports within a given file when we're trying to resolve names within that file.

Extension binding: Associate each extension with the nominal type that it extends. This requires name lookup on the name of the type being extended (e.g., "B" in "extension B { ... }"). Extending nested types introduces dependencies within this phase. For example:

```
struct A { }
extension A.Inner { }
```

```
extension A { struct Inner { } }
```

Here, the second extension must be bound before the first can be resolved. This phase is global in nature, because we need to bind all of the extensions to a particular nominal type before we can proceed with later phases for that nominal type.

Type hierarchies: Establish superclass, protocol-refinement, and protocol-conformance relationships. Detect cycles in inheritance hierarchies and break them so that later phases don't need to reason about them. At this point, general name lookup into a nominal type is possible.

Primary name binding: Resolve names to refer to declarations. The vast majority of names within declarations can be bound at this point, with the exception of names that are dependent on a type parameter.

Generic signature resolution: Collect the requirements placed on a set of generic type parameters for a given declaration, establishing equivalence classes among type parameters and associated types, and so on. Once complete, names dependent on a type parameter can be resolved to, e.g., a particular associated type. At this point, all names that occur within declarations are bound.

Declaration type validation: Produces a type for each declaration. This will involve resolving typealiases and type witnesses, among other things.

Override checking: Determine whether a given class member overrides a member of a superclass.

Attribute checking: Determine whether a given member has a particular attribute, e.g., `@objc`, `@available`, `dynamic`. This depends on override checking because many attributes are inherited. Note that most attributes are independent, so this can be thought of as a per-attribute phase.

Semantics checking: Check that a particular declaration meets the semantic requirements of the language. This particular phase is only important for the file that is currently being type-checked. In particular, for any file that is not the primary file, this checking is useless work.

Challenges in the Language

There are a few aspects of the language that make it challenging to implement the declaration type checker. In some cases, we simply need to be more careful in the implementation, while others may require us to restrict the language. Some specific challenges:

Extension binding requiring later phases: When an extension refers to a typealias, we end up with a dependency through the typealias. For example, consider:

```
struct B { }
typealias C = B.Inner
extension C { }
extension B { struct Inner { } }
```

Here, the name lookup used for the first extension needs to resolve the typealias, which depends on the second extension having already been bound. There is a similar dependency on resolving superclasses before binding extensions:

```
class X { struct Inner { } }
class Y : X { }
extension Y.Inner { }
```

We can address this problem by restricting the language to disallow extensions via typealiases and limit the name lookup used for extensions to not consider anything in the superclass or within protocols. It's also possible that a sufficiently lazy type checker could resolve such dependencies.

Type witness inference: Type witnesses can be inferred from other requirements. For example:

```
protocol SequenceType {
  typealias Element
  mutating func makeIterator() -> Element?
}

struct IntRangeGenerator : SequenceType {
  var current: Int
  let limit: Int

  // infers SequenceType's Element == Int
  mutating func makeIterator() -> Int? {
    if current == limit { return nil }
    return current++
  }
}
```

Type witness inference is a global problem, which involves (among other things) matching the requirements of a protocol to potential witnesses within the model type as well as protocol extensions, performing overload resolution to find the best potential witness, and validating that the potential type witnesses meets the requirements of the protocol. Supporting this feature correctly likely means recording type variables in the AST for type witnesses that are being inferred.

Inferring a property's type from its initial value: The type of a property can be inferred from its initial value, which makes the declaration type checker dependent on the expression type checker. This requires recursion checking that goes through the

expression type checker to diagnose, e.g.:

```
var x = y + z
var y = 1
var z = x + y
```

Fortunately, this should be fairly simple: when inferring the type of a property, it can be temporarily recorded as having unresolved type. Any attempt to refer to a property of unresolved type within the expression type checker will be considered ill-formed due to recursion.

Proposed Architecture

To address the problems with the current declaration type checker, we propose a new architecture. The key components of the new architecture are:

Represent phases in the AST: Each AST node should know to which phase it has been type-checked. Any accessor on the AST has a corresponding minimum phase, which it can assert. For example, the accessor that retrieves the superclass of a class declaration will assert that the class is at least at the "type hierarchies" phase; it's programmer error to not have established that the class is at that phase before asking the question.

Model dependencies for phase transitions: For a given AST node and target phase, we need to be able to enumerate the phase transitions that are required of other AST nodes before the transition can be performed. For example:

```
protocol P {
  typealias Assoc
}

struct X<T> : P {

}

func foo(_ x: X<Int>.Assoc) { }
```

To bring the `TypeRepr` for `X<Int>.Assoc` to the "primary name binding" phase, we need to bring `X` up to the "primary name binding" phase. Once all dependencies for a phase transition have been resolved, we can perform the phase transition. As noted earlier, it's important to make the dependencies minimal: for example, note that we do not introduce any dependencies on the type argument (`Int`) because it does not affect name lookup. It could, however, affect declaration type validation.

Iteratively solve type checking problems: Immediately recursing to satisfy a dependency (as the current type checker does) leads to unbridged recursion in the type checker. Instead, unsatisfied dependencies should be pushed into a dependency graph that tracks all of the active AST node dependencies as well as a priority queue that guides the type checker to the next AST node whose dependencies have been satisfied.

Detect and diagnose recursive dependencies: When the priority queue contains only AST nodes that have dependencies that have not yet been satisfied, we have a circular dependency in the program. We can find the cycle within the active dependency graph and report it to the user.

Separate semantic information from the AST: Rather than stash all of the semantic and type information for declarations directly on the AST, we can keep it in a separate side table. That makes it easier to handle both global inference problems (where we need to tentatively make assumptions about type variables) and also, in the longer term, to perform more incremental compilation where we can throw away semantic and type information that has been evaluated.

Global symbol table: Name lookup within a type or extension context typically requires us to bring that type up to the "type hierarchies" phase. If we were to use a global symbol table that also had information about members, name lookup could find declarations using that symbol table, possibly without having to bring the type context past the "parsing" phase.

Case study: conformance lookup table

The protocol conformance lookup table (in `lib/AST/ProtocolConformance.cpp`), which answers questions about the set of protocols that a given class conforms to, has a similar architecture to what is proposed here. Each nominal type has a conformance lookup table, which is lazily constructed from the nominal type, its extensions, and the conformance lookup table of its superclass (if any).

Conformance checking is divided into four phases, modeled by `ConformanceLookupTable::ConformanceStage`: recording of explicitly-written conformances, handling of inherited conformances, expanding out implied conformances (due to protocol inheritance), and resolving ambiguities among different sources of conformances. The phase of nominal type declaration and each of its extensions are separately tracked, which allows for new extensions to be lazily introduced. Phase transitions are handled by a single method (`ConformanceLookupTable::updateLookupTable`) that recurses to satisfy dependencies. For example, bringing a class `C` up to the "inherited" phase requires that its superclass be brought to the "resolved" phase.

Whenever the conformance lookup table encounters a problem, such as a conflict between a superclass's protocol conformance and a subclass's protocol conformance, it records the problem in a diagnostics side-table and resolves the conflict in a manner that allows other type checking to continue. The actual diagnosis of the problem occurs only when performing complete semantics checking of the declaration that owns the erroneous protocol conformance.

Note that the conformance lookup table does *not* implement a dependency graph or priority queue as proposed above. Rather, it performs direct recursion internally (which is generally not a problem) and through the current type-validation logic (which requires it to be re-entrant).

How do we get there?

The proposed architecture is significantly different from the current type checker architecture, so how do we get there from here? There are a few concrete steps we can take:

Make all AST nodes phase-aware: Introduce a trait that can ask an arbitrary AST node (`Decl`, `TypeRepr`, `Pattern`, etc.) its current phase. AST nodes may compute this information on-the-fly or store it, as appropriate. For example, a `TypeRepr` can generally determine its phase based on the existing state of the `IdentTypeRepr` nodes it includes.

Make name lookup phase-aware: Name lookup is currently one of the worst offenders when violating phase ordering. Parameterize name lookup based on the phase at which it's operating. For example, asking for name lookup at the "extension binding" phase might not resolve type aliases, look into superclasses, or look into protocols.

Make type resolution phase-aware: Type resolution effectively brings a given `TypeRepr` up to the "declaration type validation`` phase in one shot. Parameterize type resolution based on the target phase, and start minimizing the amount of work that the type checking does. Use extension binding as a testbed for these more-minimal dependencies.

Dependency graph and priority queue: Extend the current-phase trait with an operation that enumerates the dependencies that need to be satisfied to bring a given AST node up to a particular phase. Start with `TypeRepr` nodes, and use the dependency graph and priority queue to satisfy all dependencies ahead of time, eliminating direct recursion from the type-resolution code path. Build circular-dependency detection within this test-bed.

Incremental adoption of dependency graph: Make other AST nodes (`Pattern`, `VarDecl`, etc.) implement the phase-awareness trait, enumerating dependencies and updating their logic to perform minimal updates. Certain entry points that are used for ad hoc recursion can push/pop dependency graph and priority-queue instances, which leaves the existing ad hoc recursion checking in place but allows isolated subproblems to use the newer mechanisms.

Strengthen accessor assertions: As ad hoc recursion gets eliminated from the type checker, strengthen assertions on the various AST nodes to make sure the AST node has been brought to the appropriate phase.

How do we test it?

Existing code continues to work: As we move various parts of the type checker over to the dependency graph, existing Swift code should continue to work, since we'll have fallbacks to the existing logic and the new type checker should be strictly lazier than the existing type checker.

Order-independence testing: One of the intended improvements from this type checker architecture is that we should get more predictable order-independent behavior. To check this, we can randomly scramble the order in which we type-check declarations in the primary source file of a well-formed module and verify that we get the same results.

Compiler crashers: The compiler crashers testsuite tends to contain a large number of crashes that are effectively due to infinite recursion in the type checker. We expect that many of these will be resolved.

How do we measure progress?

The proposed change is a major architectural shift, and it's only complete when we have eliminated all ad hoc recursion from the front end. There are a few ways in which we can measure progress along the way:

AST nodes that implement the phase-aware trait: Eventually, all of our AST nodes will implement the phase-aware trait. The number of AST nodes that do properly implement that trait (reporting current phase, enumerating dependencies for a phase transition) and become part of the dependency graph and priority queue gives an indication of how far we've gotten.

Accessors that check the current phase: When we're finished, each of the AST's accessors should assert that the AST node is in the appropriate phase. The number of such assertions that have been enabled is an indication of how well the type checker is respecting the dependencies.

Phases of AST nodes in non-primary files: With the current type checker, every AST node in a non-primary file that gets touched when type-checking the primary file will end up being fully validated (currently, the "attribute checking" phase). As the type checker gets lazier, the AST nodes in non-primary files will trend toward earlier phases. Tracking the number of nodes in non-primary files at each phase over time will help us establish how lazy the type checker is becoming.