Introduction

This document is intended to discuss current issues in and possible future expansions of Swift's model for ABI-stable *library evolution*, described in LibraryEvolution.rst. Like other "manifesto" documents in the Swift community, it sets out related tasks in service of a general goal, though this is still an exploration of the space more than a to-do list.

Open Issues

Recompiling changes a protocol's implementation

```
// Library, version 1
protocol MagicType {}
protocol Wearable {}
func use<T: MagicType>(_ item: T) {}
```

```
// Client, version 1
struct Amulet : MagicType, Wearable {}
use(Amulet())
```

```
// Library, version 2
protocol MagicType {
  @available(dishwasherOS 2.0, *)
  func equip()
}
extension MagicType {
  @available(dishwasherOS 2.0, *)
  func equip() { print("Equipped.") }
}

protocol Wearable {}
extension Wearable where Self: MagicType {
  @available(dishwasherOS 2.0, *)
  func equip() { print("You put it on.") }
}

func use<T: MagicType>(_ item: T) { item.equip() }
```

Let's say we're running dishwasherOS 2.0. Before the client is recompiled, the implementation of `equip()` used for `Amulet` instances can only be the default implementation, i.e. the one that prints "Equipped". However, recompiling the client will result in the constrained implementation being considered a "better" match for the protocol requirement, thus changing the behavior of the program.

This should never change the *meaning* of a program, since the default implementation for a newly-added requirement should always be *correct*. However, it may have significantly different performance characteristics or side effects that would make the difference in behavior a surprise.

This is similar to adding a new overload to an existing set of functions, which can also change the meaning of client code just by recompiling. However, the difference here is that the before-recompilation behavior was never requested or acknowledged by the client; it's just the best the library can do.

A possible solution here is to require the client to acknowledge the added requirement in some way when it is recompiled.

(We do not want to perform overload resolution at run time to find the best possible default implementation for a given type.)

Evolving Attributes

A number of annotations (attributes and modifiers) would benefit from being able to change between releases of a library. For example:

- Making a `public` class `open`.
- Making a `public` class `final`.
- Making a struct or enum `@frozen`.
- Making a function `@inlinable`.

However, all of these changes have to be associated with a specific release to be correct:

- The newly-`open` class can't be subclassed if the client's deployment target is too old.
- Convenience initializers on a newly-`final` class can't satisfy protocol requirements if the client's deployment target is too old.
- A newly-`@frozen` struct may have had more private stored properties in the past.
- A newly-`@inlinable` function's body may not work correctly with older versions of the library.

While solutions can be tailored to each of these problems, they have a common need to record the OS version when an annotation is added or changed. Without that, we're locked in to the way a declaration is originally published, or having to take correctness on faith.

Limitations

For any annotation that affects the calling conventions of a function specifically, it becomes tricky to add or remove that annotation without breaking ABI. Consider the example of a struct becoming `@frozen`. Normally, functions that pass or return non-frozen structs have to do so indirectly (i.e. through a pointer), whereas frozen structs that are "small enough" (according to the ABI) can be passed in registers instead. So we get tables like this:

Frozen struct	used in a module's ABI	used in non-public ways
in its own module	direct	direct
in a client with library evolution	direct	direct
in a client without library evolution (like an app)	direct	direct

Non-frozen struct	used in a module's ABI	used in non-public ways
in its own module	indirect (no promises)	direct
in a client with library evolution	indirect (knows nothing)	indirect (knows nothing)
in a client without library evolution (like an app)	indirect (knows nothing)	indirect (knows nothing)

However, for any library *or client* with library evolution enabled, the ABI columns must not change. That implies that marking a struct as frozen *after its initial release* is different from marking the struct frozen from the start.

Frozen struct that used to be non-frozen	used in a module's ABI	used in non-public ways
in its own module	indirect (ABI compat)	direct
in a client with library evolution	indirect (ABI compat)	direct
in a client without library evolution (like an app)	direct	direct

There are a few ways to improve upon this, the most obvious being that any APIs with newer availability than when the struct became frozen can pass the struct around directly. It's worth noting that a library's minimum deployment target *cannot* be used for this purpose, since changing a minimum deployment target is not supposed to change a library's ABI.

This isn't a *terrible* performance cost; remember that all C++ methods take `this` indirectly, and clients with new enough deployment targets can still *manipulate* the values directly even if the calling convention can't change. But it's something to think about for every annotation that can change after a type is introduced.

Additional Annotations

Layout Constraints

Developers with performance-sensitive use cases may want to promise more specific things about various types and have the compiler enforce them.

- "trivial": Promises that assignment just requires a fixed-size bit-for-bit copy without any indirection or reference-counting operations. This may allow more efficient copying and destruction of values.
- "maximum size N/alignment A": Promises that the type's size and required alignment are at most N bits and A bits, respectively. (Both may be smaller.) This would make it easier for the compiler to work with values on the stack.

(Neither of these names / spellings are final. The name "trivial" comes from C++, though Swift's trivial is closer to C++'s "trivially copyable".)

Both of these annotations are things the compiler already knows when a type is declared, but writing them explicitly (a) allows the developer to check that they've made something properly optimizable, and (b) promises not to change that behavior between releases, even if, say, stored properties are added to or removed from a struct.

Frozen classes

The compiler actually has basic support for frozen classes, which allow stored properties to be accessed directly by offset from the class reference. There's no real reason why this can't be supported more generally, but confusion around *what's* being frozen and the rarity of wanting to make this promise anyway kept it out of SE-0260.

Generalized Availability Model

This section is focused on various ways to extend the OS-version-based availability model.

Per-library availability

As a placeholder, imagine replacing OS versions with library versions in the various parts of Swift's availability checking syntax:

```
// Client code
@available(Magician 1.5)
class CrystalBallView : MagicView { /*...*/ }

func scareMySiblings() {
  if #available(Magician 1.2) {
    summonDemons()
  } else {
    print("BOO!!")
  }
}
```

This sort of version checking is important for *backward*-deployment, where the developer has the interface available for a new version of a library (Magician 1.5), but might end up running the client against an old version (Magician 1.0). It's not interesting for *forward*-deployment, however—that's all up to the library to preserve ABI compatibility.

Possible implementations for version checking include generating a hidden symbol into a library, or putting the version number in some kind of metadata, like the Info.plist in a framework bundle on Apple platforms.

Publishing API with availability

A library's API is already marked with the `public` modifier, but if a client wants to work with multiple releases of the library, the API needs availability information as well. Declaring availability on an entity using the *current* library's name specifies the oldest version in which that entity can be used.

- Classes, structs, enums, and protocols may all have availability.
- Methods, properties, subscripts, and initializers may all have availability.
- Top-level functions, variables, and constants may have availability.
- Typealiases are treated as having availability for the purpose of availability checking, even though they have no run-time presence.

In a versioned library, any top-level public entity from the list above may not be made ABI-public (`public` , `open` , `@usableFromInline` , or `@inlinable`) without availability. A public entity declared within a type (or an extension of a type) will default to having the same availability as the type.

Code within a library may generally use all other entities declared within the library (barring their own availability checks), since the entire library is shipped as a unit. That is, even if a particular API was introduced in v1.0, its (non-public) implementation may refer to APIs introduced in later versions. Inlinable functions are the exception to this, since inlined code ends up outside the original module.

Declaring library version dependencies

Swift's current OS-based availability model includes the notion of a *minimum deployment target*, the version of an OS that must be present for the program being compiled to run at all. For example, a program compiled with a minimum deployment target of iOS 9.2 will not launch on iOS 9.0.

The generalized model suggests being able to make similar guarantees for individual libraries. For example, a client program may depend on version 1.1 of the "Magician" library; trying to run using version 1.0 will result in errors. By declaring this at compile-time, the client code can omit `@available` and `#available` checks that are satisfied by the minimum library version.

Resilience domains

In general, the features around library evolution for ABI stability's sake do not apply to clients that bundle their dependencies with them (such as an iOS app embedding third-party frameworks). In this case, a client can rely on all the current implementation details of its libraries when compiling, since the same version of the library is guaranteed to be present at run time. This allows more optimization than would otherwise be possible.

In some cases, a collection of libraries may be built and delivered together, even though their clients may be packaged separately. (For example, the ICU project is usually built into several library binaries, but these libraries are always distributed together.) While the *clients* cannot rely on a particular version of any library being present, the various libraries in the collection should be able to take advantage of the implementations of their dependencies also in the collection---that is, it should treat types as frozen (except where it would affect public-facing ABI). We've used the term *resilience domain* to describe modules in this sort of collection.

There's no design yet for how resilience domains should be specified. In today's compiler, every library with library evolution enabled is in its own resilience domain, and every library that *doesn't* have library evolution enabled is in the "app" resilience domain.

Deployments

Related to the concept of a resilience domain is a *deployment*. While a resilience domain allows related libraries to be compiled more efficiently, a deployment groups related libraries together to present semantic version information to clients. The simplest example of this might be an OS release: OS X 10.10.0 contains Foundation version 1151.16 and AppKit version 1343. A deployment thus acts as a "virtual dependency": clients that depend on OS X 10.10 can rely on the presence of both of the library versions above.

The use of deployments allows clients to only have to think about aggregate dependencies, instead of listing every library they might depend on. It also allows library authors to build [many versions of a library][Foundation version numbers] within a larger release cycle, as well as allowing a vendor to bundle together many libraries with uncoordinated release schedules and release them as a logical unit.

[Foundation version numbers]:
https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Miscellaneous/Foundation_Constants/index.html#//apple_ref/doc/constant_group/Found

There are lots of details to figure out here, including how to distribute this information. In particular, just like libraries publish the history of their own APIs, a deployment must publish the history of their included library versions, i.e. not just that OS X 10.10 contains Foundation 1151.16 and AppKit 1343, but also that OS X 10.9 contains Foundation 1056 and AppKit 1265, and that OS X 10.8 contains Foundation 945.0 and AppKit 1187, and so on, back to the earliest version of the deployment that is supported.

Obviously, formalizing a model here is probably most useful for people distributing ABI-stable Swift libraries as part of an OS, i.e. Apple.

Automated Tooling

ABI Checker

The Swift repository has a basic ABI checker in the form of swift-api-digester. This tool looks at two versions of a library and determines if there are any changes which are known to be unsafe (say, changing the type of a function parameter). It would be nice™ to integrate this into SwiftPM and Xcode in some way.

Automatic Versioning

A possible extension of the ABI checker would be a tool that *automatically* generates versioning information for entities in a library, given the previous public interface of the library. This would remove the need for versions on annotations, and declaring new public API would be as simple as marking an entity `public` . Obviously this would also remove the possibility of human error in managing library versions.

However, making this tool has a number of additional difficulties beyond the simple checker tool:

- The tool must be able to read past library interface formats. This is true for a validation tool as well, but the cost of failure is much higher. Similarly, the past version of a library *must* be available to correctly compile a new version.

- Because the information goes into a library's public interface, the versioning tool must either be part of the compilation process, modify the interface generated by compilation, or produce a sidecar file that can be loaded when compiling the client. In any case, it must *produce* information in addition to *consuming* it.
- Occasionally a library owner may want to override the inferred versions. This can be accomplished by providing explicit versioning information, as described above.
- Bugs in the tool manifest as bugs in client programs.