orphan:

Interaction with Objective-C

Authors: John McCall

I propose some elementary semantics and limitations when exposing Swift code to Objective-C and vice-versa.

Dynamism in Objective-C

Objective-C intentionally defines its semantics almost entirely around its implementation model. The implementation supports two basic operations that you can perform on any object:

Instance variable access

An access to an ivar `foo` is type-checked by looking for a declared ivar with the name `foo` in the static type of the base object. To succeed, that search must find an ivar from some class `C`.

It is undefined behavior if, at runtime, the base expression does not evaluate to a valid object of type `C` or some subclass thereof. This restriction is necessary to allow the compiler to compute the address of the ivar at (relatively) minimal expense.

Because of that restriction, ivar accesses are generally seen by users as "primitive", and we probably have some additional flexibility about optimizing them. For example, we could probably impose a rule insisting that the entire static type of the base be accurate, thus forbidding a user from accessing an ivar using a pointer to an incorrect subclass of `C`. This would theoretically allow stronger alias analysis: for example, if we have `D *d;` and `E* e;` where those types are subclasses of `C`, we would be able to prove that `d->foo` does not alias `e->foo`. However, in practice, ObjC ivars tend to be directly accessed only within the implementation of the class which defines them, which means that the stronger rule would basically never kick in.

Message sends

A message send is type-checked by looking for a declared method with that selector in the static type of the receiver, or, if the receiver has type `id` or `Class`, in any known class in the translation unit. The arguments and expression result are then determined by that method declaration.

At runtime, if the receiver is actually `nil`, the message send returns immediately with a zero-initialized result; otherwise the runtime searches for a method implementation (and, sometimes, an alternate receiver) via a somewhat elaborate algorithm:

- The runtime first searches the method tables of the object for a method with that selector, starting from the object's dynamic class and proceeding up the hierarchy.
- If that search fails, the runtime gives the class an opportunity to dynamically resolve the method by adding it to the method lists of one of the classes in the hierarchy.
- If dynamic resolution fails, the runtime invokes the forwarding handler; the standard handler installed by CoreFoundation follows a complex rule that needn't be described here other than to note that it can involve actually sending the message to a completely different object if the receiver agrees.

It is undefined behavior if the signature of the method implementation actually found is not compatible with the signature of the method against which the message send was type-checked. This restriction is necessary in order to avoid the need to perform dynamic signature checking and automatic implicit conversion on arguments, which would substantially slow down the method-call mechanism.

Note that common practice requires this sense of "compatible" to be much looser than C's. For example, `performSelector` is documented as expecting the target method to return an object reference, but it is regularly used to invoke methods that actually return `void`.

Otherwise, the language provides no guarantees which would allow the compiler to reason accurately about what the invoked method will actually do. Moreover, this is not just a formal oversight; there is quite a bit of code relying on the ability to break each of these non-guarantees. We can classify them into five categories:

- The language does not guarantee that the static type of an object reference will be accurate. Just because a value is typed `C*` does not mean it is dynamically a reference to a `C` object. Proxy objects are frequently passed around as if they were values of their target class. (Some people would argue that users should only proxy protocols, not entire concrete classes; but that's not universally followed.)
- The language does not guarantee that the complete class hierarchy is statically knowable. Even ignoring the limitations of the C compilation model, it is possible to dynamically construct classes with new methods that may override the declared behavior of the class. Core Data relies on this kind of dynamic class generation.
- The language does not guarantee that an object's dynamic class will be the type that a user actually appeared to allocate. Even ignoring proxies, it is relatively common for a factory method on a class to return an instance of a subclass.
- The language does not guarantee that an object's dynamic class will not change. KVO works by changing an object's dynamic class to a dynamically-generated subclass with new methods that replace the observed accessors.

  However, it is probably reasonable to call it undefined behavior to change a dynamic class in a way that would add

or remove ivars.

- The language does not guarantee that a class will remain constant. Loading a new dynamic library may introduce categories that add and replace methods on existing classes, and the runtime provides public functions to do the same. These features are often used to introduce dynamic instrumentation, for example when debugging.

All of these constraints combined --- actually, many of them individually --- make devirtualization completely impossible in Objective-C [1].

[1] Not completely. We could optimistically apply techniques typically used in dynamic language implementations. For example, we could directly call an expected method body, then guard that call with a check of the actual dispatched implementation against the expected method pointer. But since the inlined code would necessarily be one side of a "diamond" in the CFG, and the branches in that diamond would overwhelmingly be unthreadable, it is not clear that the optimization would gain much, and it would significantly bloat the call.

Devirtualization in Swift

Method devirtualization [2] is likely to be a critically important optimization in Swift.

[2] In contrast to generic or existential devirtualization, which are also important, but which aren't affected by the Objective-C interoperation model.

A Missing Optimization

For one, it is an important missing optimization even in Objective-C. Any program that tries to separate its concerns will usually introduce some extra abstraction in its formal model. For example:

- A class might provide multiple convenience initializers that all delegate to each other so that all initialization will flow through a single point.
- A large operation might be simpler to reason about when split into several smaller methods.
- A property might be abstracted behind a getter/setter to make it easier to change the representation (or do additional work on set) later.

In each of the examples, the user has made a totally reasonable decision about code organization and reserved flexibility, and Objective-C proceeds to introduce unnecessary runtime costs which might force a performance-sensitive programmer to choose a different path.

Swift-Specific Concerns

The lack of devirtualization would hit Swift much harder because of its property model. With a synthesized property, Objective-C provides a way to either call the getter/setter (with dot syntax) or directly access the underlying ivar (with arrow syntax). By design, Swift hides that difference, and the abstract language model is that all accesses go through a getter or setter.

Using a getter or setter instead of a direct access is a major regression for several reasons. The first is the direct one: the generated code must call a function, which prevents the compiler from keep values live in the most efficient way, and which inhibits most compiler analyses. The second is a by-product of value types: if a value is read, modified, and then written back, the modification will take place on the temporary copy, forcing a copy-on-write. Any proposal to improve on that relies on having a richer API for the access than merely a getter/setter pair, which cannot be guaranteed.

For properties of a value type, this isn't a performance problem, because we can simply look at the implementation (ignoring resilience for now) and determine whether we can access the property directly. But for properties of a class type, polymorphism requires us to defensively handle the possibility that a subclass might add arbitrary logic to either the getter or setter. If our implementation model is as unrestricted as Objective-C's, that's a serious problem.

I think that this is such a massive regression from Objective-C that we have to address it.

Requirements for Devirtualization

There are several different ways to achieve devirtualization, each with its own specific requirements. But they all rely on a common guarantee: we must remove or constrain the ability to dynamically add and replace method implementations.

Restricting Method Replacement

There are two supported ways to add or replace methods in Objective-C.

The first is via the runtime API. If we do have to support doing this to replace Swift methods --- and we should try to avoid that --- then I think restricting it to require a `@dynamic` annotation on the replaceable method (or its lexical context) is reasonable. We should try to get the Objective-C runtime to complain about attempts to replace non-dynamic methods.

The second is via categories. It's generally understood that a category replacing an existing method implementation is "rude"

Point of Allocation

If we can see the true point of allocation of an object, then we know its dynamic class at that point. However:

Note that the true point of allocation is not the point at which we call some factory method on a specific type; it has to be an actual allocation: an `alloc_ref` SIL instruction. And it's questionable whether an ObjC allocation counts, because those sometimes don't return what you might expect.

Once you know the dynamic class at a particular point, you can devirtualize calls if:

- There is no supported way to replace a function implementation.

`+[NSManagedObject alloc]` does this).

We can reason forward from the point of allocation.

If we can see that an object was allocated with `alloc_object`, then we know the dynamic class at that point. That's relatively easy to deal with.

-

If we can restrict the ability to
change the dynamic class, or at least restrict

Access Control

Swift does give us one big tool for devirtualization that Objective-C lacks: access control. In Swift, access control determines visibility, and it doesn't make sense to override something that you can't see. Therefore:

- A declaration which is private to a file can only be overridden within that file.
- A declaration which is private to a module can only be overridden with that module.
- A public declaration can be overridden anywhere [3].

[3] We've talked about having two levels of public access control for classes, so that you have to opt in to subclassability. I still think this is a good idea.

This means that a private stored property can always be "devirtualized" into a direct access [4]. Unfortunately, `private` is not the default access control: module-private is. And if the current module can contain Objective-C code, then even that raises the question of what ObjC interop actually means.

[4] Assuming we don't introduce a supported way of dynamically replacing the implementation of a private Swift method!

Using Swift Classes from Objective-C

open the question of

Because we intentionally hide the difference between a stored property and its underlying storage,

For another example, code class might access

In both cases, t makes sense to organize the code that way, but Objective-C punishes the performance of that code in order to reserve the language's

provide some abstractions which are unnecessary in the current implementation. For example, a class might have multiple initializers, each chaining to another, so that it can be conveniently constructed with any set of arguments but any special initialization logic can go in one place.

Reserving that flexibility in the code is often good sense, and reserving it across API boundaries is good language design, but it's silly not actually

Well-factored object-oriented code often contains a large number of abstractions that improve the organization of the code or make it easier to later extend or maintain, but serve no current purpose.

In typical object-oriented code, many operations are split into several small methods in order to improve code organization and reserve the ability to

Conscientious developers

runtime calls cached-offset calculation for the ivar location.

restriction, there's general acceptance that the

is necessary to make ivar accesses not ridiculously expensive. Because of that, there's general acceptance that

If the compiler team cared, we could *probably* convince people to accept a language rule that requires the entire static type to be accurate, so that e.g.

Otherwise Objective-C provides no restrictions beyond those imposed by the source language.

ivar is accessed on an object reference which is `nil` or otherwise not a valid object of the class

- You can send an object reference a message.

- If the reference is actually `nil`, the message send returns immediately with a zero-initialized result.
- Otherwise, the runtime searches the class hierarchy of the object, starting from the object's dynamic class and proceeding to superclasses, looking for a concrete method matching the selector of the message. If a concrete method is found, it is called.
- If the class hierarchy contains no such method, a different method is invoked on the class to give it an opportunity to dynamically resolve the method; if taken, the process repeats, but skipping this step the second time.
- Otherwise, the global forwarding handler is invoked. On Darwin, this is set up by CoreFoundation, and it follows a complicated protocol which relies on NSInvocation.

It is undefined behavior if the type signature of the method implementation ultimately found is not compatible with the type signature of the method that was used to statically type-check the message send. This includes semantic annotations like ARC ownership conventions and the `noreturn` attribute. Otherwise, there are no semantic restrictions on what any particular method can do.

signature of the method implementation's
pr signature is not compatible with the signature at which the method was invoked.

, in which case the runtime searches
the class hierarchy of the object, from most to least derived, and calls the method

In Objective-C, every object has a class and every class has a collection of methods. The high-level semantics are essentially those

Basic Semantics

`@public` makes a declaration visible in code where the enclosing module is imported. So, given this declaration in the `Satchel` module:

```
@public struct Bag<T> : ... {
 ...
}
```

We could write, in any other module,

```
import Satchel
typealias SwingMe = Bag<Cat>
```

The difference from the status quo being that without `@public` on the declaration of `Bag`, the use of `Bag` above would be ill-formed.

Type-Checking

The types of all parameters and the return type of a func marked `@public` (including the implicit `self` of methods) must also be `@public`.

All parameters to a `func` marked `@public` (including the implicit `self` of methods) must also be `@public`:

```
struct X {}                    // not @public
@public struct Y {}
func f(_: X) {}                // OK; also not @public
@public func g(_: Y) {}        // OK; uses only @public types
@public func h(_: X, _: Y) {} // Ill-formed; non-public X in public signature
```

A `typealias` marked `@public` must refer to a type marked `@public`:

```
typealias XX = X               // OK; not @public
@public typealias YY = Y       // OK; Y is @public
@public typealias XXX = X      // Ill-formed; public typealias refers to non-public type
```

There is a straightforward and obvious rule for composing the `@public`-ness of any compound type, including function types, tuple types and instances of generic types: The compound type is public if and only if all of the component types, are `@public` and either defined in this module or re-exported from this module.

Enums

The cases of an `enum` are `@public` if and only if the `enum` is declared `@public`.

Derived Classes

A method that overrides an `@public` method must be declared `@public`, even if the enclosing class is non-`@public`.

Protocols

A `@public` protocol can have `@public` and non-`@public` requirements. `@public` requirements can only be satisfied by `@public` declarations. Non-`@public` requirements can be satisfied by `@public` or non-`@public` declarations.

Conformances

The conformance of a type to a protocol is `@public` if that conformance is part of an `@public` declaration. The program is ill-formed if any declaration required to satisfy a `@public` conformance is not also declared `@public`.:

```
@public protocol P {
  @public func f() { g() }
  func g()
}

struct X : P { // OK, X is not @public, so neither is its
  func f() {}  // conformance to P, and therefore f
  func g() {}  // can be non-@public
}

protocol P1 {}

@public struct Y : P1 {} // Y is @public so its
                         // conformance to P1 is, too.

@public
extension Y : P {     // This extension is @public, so
  @public func f() {} // Y's conformance to P is also, and
  func g() {}         // thus f must be @public too
}

protocol P2 {}

extension Y : P2 {}   // Y's conformance to P2 is non-@public
```

> **Note**
>
> It's our expectation that in the near term, and probably for v1.0, non-`@public` conformances on `@public` types will be diagnosed as ill-formed/unsupported.

A Related Naming Change

The existing `@exported` attribute for imports should be renamed `@public` with no change in functionality.

Future Directions

Some obvious directions to go in this feature space, which we are not proposing today, but with which we tried to make this proposal compatible:

- non-`@public` conformances
- file-private accessibility
- explicit non-`@public` overrides, e.g. `@!public`