orphan:

Initializer Inheritance

Authors: Doug Gregor, John McCall

Contents

Introduction

This proposal introduces the notion of initializer inheritance into the Swift initialization model. The intent is to more closely model Objective-C's initializer inheritance model while maintaining memory safety.

Background

An initializer is a definition, and it belongs to the class that defines it. However, it also has a signature, and it makes sense to talk about other initializers in related classes that share that signature.

Initializers come in two basic kinds: complete object and subobject. That is, an initializer either takes responsibility for initializing the complete object, potentially including derived class subobjects, or it takes responsibility for initializing only the subobjects of its class and its superclasses.

There are three kinds of delegation:

- **super**: runs an initializer belonging to a superclass (in ObjC, `[super init...]`; in Swift, `super.init(...)`).
- **peer**: runs an initializer belonging to the current class (ObjC does not have syntax for this, although it is supported by the runtime; in Swift, the current meaning of `self.init(...)`)
- **dispatched**: given a signature, runs the initializer with that signature that is either defined or inherited by the most-derived class (in ObjC, `[self init...]`; not currently supported by Swift)

We can also distinguish two ways to originally invoke an initializer:

- **direct**: the most-derived class is statically known
- **indirect**: the most-derived class is not statically known and an initialization must be dispatched

Either kind of dispatched initialization poses a soundness problem because there may not be a sound initializer with any given signature in the most-derived class. In ObjC, initializers are normal instance methods and are therefore inherited like normal, but this isn't really quite right; initialization is different from a normal method in that it's not inherently sensible to require subclasses to provide initializers at all the signatures that their superclasses provide.

Subobject initializers

The defining class of a subobject initializer is central to its behavior. It can be soundly inherited by a class C only if is trivial to initialize the ivars of C, but it's convenient to ignore that and assume that subobjects will always trivially wrap and delegate to superclass subobject initializers.

A subobject initializer must either (1) delegate to a peer subobject initializer or (2) take responsibility for initializing all ivars of its defining class and delegate to a subobject initializer of its superclass. It cannot initialize any ivars of its defining class if it then delegates to a peer subobject initializer, although it can re-assign ivars after initialization completes.

A subobject initializer soundly acts like a complete object initializer of a class C if and only if it is defined by C.

Complete object initializers

The defining class of a complete object initializer doesn't really matter. In principle, complete object initializers could just as well be freestanding functions to which a metatype is passed. It can make sense to inherit a complete object initializer.

A complete object initializer must either (1) delegate (in any way) to another complete object initializer or (2) delegate (dispatched) to a subobject initializer of the most-derived class. It may not initialize any ivars, although it can re-assign them after initialization completes.

A complete object initializer soundly acts like a complete object initializer of a class C if and only if it delegates to an initializer which

soundly acts like a complete object initializer of C.

These rules are not obvious to check statically because they're dependent on the dynamic value of the most-derived class C. Therefore any ability to check them depends on restricting C somehow relative to the defining class of the initializer. Since, statically, we only know the defining class of the initializer, we can't establish soundness solely at the definition site; instead we have to prevent unsound initializers from being called.

Guaranteed initializers

The chief soundness question comes back to dispatched initialization: when can we reasonably assume that the most-derived class provides an initializer with a given signature?

Only a complete object initializer can perform dispatched delegation. A dispatched delegation which invokes another complete object initializer poses no direct soundness issues. The dynamic requirement for soundness is that, eventually, the chain of dispatches leads to a subobject initializer that soundly acts like a complete object initializer of the most-derived class.

Virtual initializers

The above condition is not sufficient to make indirect initialization sound, because it relies on the ability to simply not use an initializer in cases where its delegation behavior isn't known to be sound, and we can't do that to arbitrary code. For that, we would need true virtual initializers.

A virtual initializer is a contract much more like that of a standard virtual method: it guarantees that every subclass will either define or inherit a constructor with a given signature, which is easy to check at the time of definition of the subclass.

Proposal

Currently, all Swift initializers are subobject initializers, and there is no way to express the notion of a complete subobject initializer. We propose to introduce complete subobject initializers into Swift and to make them inheritable when we can guarantee that doing so is safe.

Complete object initializers

Introduce the notion of a complete object initializer, which is written as an initializer with `Self` as its return type [1], e.g.:

```
init() -> Self {
  // ...
}
```

The use of `Self` here fits well with dynamic `Self`, because a complete object initializer returns an instance of the dynamic type being initialized (rather than the type that defines the initializer).

A complete object initializer must delegate to another initializer via `self.init`, which may itself be either a subobject initializer or a complete object initializer. The delegation itself is dispatched. For example:

```
class A {
  var title: String

  init() -> Self { // complete object initializer
    self.init(withTitle:"The Next Great American Novel")
  }

  init withTitle(title: String) { // subobject initializer
    self.title = title
  }
}
```

Subobject initializers become more restricted. They must initialize A's instance variables and then perform super delegation to a subobject initializer of the superclass (if any).

Initializer inheritance

A class inherits the complete object initializers of its direct superclass when it overrides all of the subobject initializers of its direct superclass. Subobject initializers are never inherited. Some examples:

```
class B1 : A {
  var counter: Int

  init withTitle(title: String) { // subobject initializer
    counter = 0
    super.init(withTitle:title)
  }

  // inherits A's init()
}

class B2 : A {
```

```
  var counter: Int

  init withTitle(title: String) -> Self { // complete object initializer
    self.init(withTitle: title, initialCount: 0)
  }

  init withTitle(title: String) initialCount(Int) { // subobject initializer
    counter = initialCount
    super.init(withTitle:title)
  }

  // inherits A's init()
}

class B3 : A {
  var counter: Int

  init withInitialCount(initialCount: Int) { // subobject initializer
    counter = initialCount
    super.init(withTitle: "Unnamed")
  }

  init withStringCount(str: String) -> Self { // complete object initializer
    var initialCount = 0
    if let count = str.toInt() { initialCount = count }
    self.init(withInitialCount: initialCount)
  }

  // does not inherit A's init(), because init withTitle(String) is not
  // overridden.
}
```

`B3` does not override `A`'s subobject initializer, so it does not inherit `init()`. Classes `B1` and `B2`, however, both inherit the initializer `init()` from `A`, because both override its only subobject initializer, `init withTitle(String)`. This means that one can construct either a `B1` or a `B2` with no arguments:

```
B1() // okay
B2() // okay
B3() // error
```

That `B1` uses a subobject initializer to override it's superclass's subobject initializer while `B2` uses a complete object initializer has an effect on future subclasses. A few more examples:

```
class C1 : B1 {
  init withTitle(title: String) { // subobject initializer
    super.init(withTitle:title)
  }

  init withTitle(title: String) initialCount(Int) { // subobject initializer
    counter = initialCount
    super.init(withTitle:title)
  }
}

class C2 : B2 {
  init withTitle(title: String) initialCount(Int) { // subobject initializer
    super.init(withTitle: title, initialCount:initialCount)
  }

  // inherits A's init(), B2's init withTitle(String)
}

class C3 : B3 {
  init withInitialCount(initialCount: Int) { // subobject initializer
    super.init(withInitialCount: initialCount)
  }

  // inherits B3's init withStringCount(str: String)
  // does not inherit A's init()
}
```

Virtual initializers

With the initializer inheritance rules described above, there is no guarantee that one can dynamically dispatch to an initializer via a metatype of the class. For example:

```
class D {
  init() { }
}
```

```
func f(_ meta: D.Type) {
  meta() // error: no guarantee that an arbitrary of subclass D has an init()
}
```

Virtual initializers, which are initializers that have the `virtual` attribute, are guaranteed to be available in every subclass of `D`. For example, if `D` was written as:

```
class D {
  @virtual init() { }
}

func f(_ meta: D.Type) {
  meta() // okay: every subclass of D guaranteed to have an init()
}
```

Note that `@virtual` places a requirement on all subclasses to ensure that an initializer with the same signature is available in every subclass. For example:

```
class E1 : D {
  var title: String

  // error: E1 must provide init()
}

class E2 : D {
  var title: String

  @virtual init() {
    title = "Unnamed"
    super.init()
  }

  // okay, init() is available here
}

class E3 : D {
  var title: String

  @virtual init() -> Self {
    self.init(withTitle: "Unnamed")
  }

  init withTitle(title: String) {
    self.title = title
    super.init()
  }
}
```

Whether an initializer is virtual is orthogonal to whether it is a complete object or subobject initializer. However, an inherited complete object initializer can be used to satisfy the requirement for a virtual requirement. For example, `E3`'s subclasses need not provide an `init()` if they override `init withTitle(String)`:

```
class F3A : E3 {
  init withTitle(title: String) {
    super.init(withTitle: title)
  }

  // okay: inherited ``init()`` from E3 satisfies requirement for virtual init()
}

class F3B : E3 {
  // error: requirement for virtual init() not satisfied, because it is neither defined nor inherited
}

class F3C : E3 {
  @virtual init() {
    super.init(withTitle: "TSPL")
  }

  // okay: satisfies requirement for virtual init().
}
```

Objective-C interoperability

When an Objective-C class that contains at least one designated-initializer annotation (i.e., via `NS_DESIGNATED_INITIALIZER`) is imported into Swift, it's designated initializers are considered subobject initializers. Any non-designed initializers (i.e., secondary or convenience initializers) are considered to be complete object initializers. No other special-case behavior is warranted here.

When an Objective-C class with no designated-initializer annotations is imported into Swift, all initializers in the same module as the class definition are subobject initializers, while initializers in a different module are complete object initializers. This effectively means

that subclassing Objective-C classes without designated-initializer annotations will provide little or no initializer inheritance, because one would have to override nearly *all* of its initializers before getting the others inherited. This seems acceptable so long as we get designated-initializer annotations into enough of the SDK.

In Objective-C, initializers are always inherited, so an error of omission on the Swift side (failing to override a subobject initializer from a superclass) can result in runtime errors if an Objective-C framework messages that initializer. For example, consider a trivial `NSDocument`:

```
class MyDocument : NSDocument {
  var title: String
}
```

In Swift, there would be no way to create an object of type `MyDocument`. However, the frameworks will allocate an instance of `MyDocument` and then send a message such as `initWithContentsOfURL:ofType:error:` to the object. This will find `-[NSDocument initWithContentsOfURL:ofType:error:]`, which delegates to `-[NSDocument init]`, leaving `MyDocument`'s stored properties uninitialized.

We can improve the experience slightly by producing a diagnostic when there are no initializers for a given class. However, a more comprehensive approach is to emit Objective-C entry points for each of the subobject initializers of the direct superclass that have not been implemented. These entry points would immediately abort with some diagnostic indicating that the initializer needs to be implemented.

[1] Syntax suggestion from Joe Groff.