orphan:

Initialization

Contents

Superclass Delegation

The initializer for a class that has a superclass must ensure that its superclass subobject gets initialized. The typical way to do so is through the use of superclass delegation:

```
class A {
  var x: Int

  init(x: Int) {
    self.x = x
  }
}

class B : A {
  var value: String

  init() {
    value = "Hello"
    super.init(5) // superclass delegation
  }
}
```

Swift implements two-phase initialization, which requires that all of the instance variables of the subclass be initialized (either within the class or within the initializer) before delegating to the superclass initializer with `super.init`.

If the superclass is a Swift class, superclass delegation is a direct call to the named initializer in the superclass. If the superclass is an Objective-C class, superclass delegation uses dynamic dispatch via `objc_msgSendSuper` (and its variants).

Peer Delegation

An initializer can delegate to one of its peer initializers, which then takes responsibility for initializing this subobject and any superclass subobjects:

```
extension A {
  init fromString(s: String) {
    self.init(Int(s)) // peer delegation to init(Int)
  }
}
```

One cannot access any of the instance variables of `self` nor invoke any instance methods on `self` before delegating to the peer initializer, because the object has not yet been constructed. Additionally, one cannot initialize the instance variables of `self`, prior to delegating to the peer, because doing so would lead to double initializations.

Peer delegation is always a direct call to the named initializer, and always calls an initializer defined for the same type as the delegating initializer. Despite the syntactic similarities, this is very different from Objective-C's `[self init...]`, which can call methods in either the superclass or subclass.

Peer delegation is primarily useful when providing convenience initializers without having to duplicate initialization code. However, peer delegation is also the only viable way to implement an initializer for a type within an extension that resides in a different resilience domain than the definition of the type itself. For example, consider the following extension of `A`:

```
extension A {
  init(i: Int, j: Int) {
    x = i + j    // initialize x
  }
}
```

If this extension is in a different resilience domain than the definition of `A`, there is no way to ensure that this initializer is initializing all of the instance variables of `A`: new instance variables could be added to `A` in a future version (these would not be properly initialized) and

existing instance variables could become computed properties (these would be initialized when they shouldn't be).

Initializer Inheritance

Initializers are *not* inherited by default. Each subclass takes responsibility for its own initialization by declaring the initializers one can use to create it. To make a superclass's initializer available in a subclass, one can re-declare the initializer and then use superclass delegation to call it:

```
class C : A {
  var value = "Hello"

  init(x: Int) {
    super.init(x) // superclass delegation
  }
}
```

Although `C`'s initializer has the same parameters as `A`'s initializer, it does not "override" `A`'s initializer because there is no dynamic dispatch for initializers (but see below).

We could syntax-optimize initializer inheritance if this becomes onerous. DaveA provides a reasonable suggestion:

```
class C : A {
  var value = "Hello"

  @inherit init(Int)
}
```

Note: one can only inherit an initializer into a class `C` if all of the instance variables in that class have in-class initializers.

Virtual Initializers

The initializer model above only safely permits initialization when we statically know the type of the complete object being initialized. For example, this permits the construction `A(5)` but not the following:

```
func createAnA(_ aClass: A.metatype) -> A {
  return aClass(5) // error: no complete initializer accepting an ``Int``
}
```

The issue here is that, while `A` has an initializer accepting an `Int`, it's not guaranteed that an arbitrary subclass of `A` will have such an initializer. Even if we had that guarantee, there wouldn't necessarily be any way to call the initializer, because (as noted above), there is no dynamic dispatch for initializers.

This is an unacceptable limitation for a few reasons. The most obvious reason is that `NSCoding` depends on dynamic dispatch to `-initWithCoder:` to deserialize an object of a class type that is dynamically determined, and Swift classes must safely support this paradigm. To address this limitation, we can add the `virtual` attribute to turn an initializer into a virtual initializer:

```
class A {
  @virtual init(x: Int) { ... }
}
```

Virtual initializers can be invoked when constructing an object using an arbitrary value of metatype type (as in the `createAnA` example above), using dynamic dispatch. Therefore, we need to ensure that a virtual initializer is always a complete object initializer, which requires that every subclass provide a definition for each virtual initializer defined in its superclass. For example, the following class definition would be ill-formed:

```
class D : A {
  var floating: Double
}
```

because `D` does not provide an initializer accepting an `Int`. To address this issue, one would add:

```
class D : A {
  var floating: Double

  @virtual init(x: Int) {
    floating = 3.14159
    super.init(x)
  }
}
```

As a convenience, the compiler could synthesize virtual initializer definitions when all of the instance variables in the subclass have in-class initializers:

```
class D2 : A {
  var floating = 3.14159

  /* compiler-synthesized */
  @virtual init(x: Int) {
    super.init(x)
```

```
  }
}
```

This looks a lot like inherited initializers, and can eliminate some boilerplate for simple subclasses. The primary downside is that the synthesized implementation might not be the right one, e.g., it will almost surely be wrong for an inherited `-initWithCoder:`. I don't think this is worth doing.

Note: as a somewhat unfortunate side effect of the terminology, the initializers for structs and enums are considered to be virtual, because they are guaranteed to be complete object initializers. If this bothers us, we could use the term (and attribute) "complete" instead of "virtual". I'd prefer to stick with "virtual" and accept the corner case.

Initializers in Protocols

We currently ban initializers in protocols because we didn't have an implementation model for them. Protocols, whether used via generics or via existentials, use dynamic dispatch through the witness table. More importantly, one of the important aspects of protocols is that, when a given class `A` conforms to a protocol `P`, all of the subclasses of `A` also conform to `P`. This property interacts directly with initializers:

```
protocol NSCoding {
  init withCoder(coder: NSCoder)
}

class A : NSCoding {
  init withCoder(coder: NSCoder) { /* ... */ }
}

class B : A {
  // conforms to NSCoding?
}
```

Here, `A` appears to conform to `NSCoding` because it provides a matching initializer. `B` should conform to `NSCoding`, because it should inherit its conformance from `A`, but the lack of an `initWithCoder:` initializer causes problems. The fix here is to require that the witness be a virtual initializer, which guarantees that all of the subclasses will have the same initializer. Thus, the definition of `A` above will be ill-formed unless `initWithCoder:` is made virtual:

```
protocol NSCoding {
  init withCoder(coder: NSCoder)
}

class A : NSCoding {
  @virtual init withCoder(coder: NSCoder) { /* ... */ }
}

class B : A {
  // either error (due to missing initWithCoder) or synthesized initWithCoder:
}
```

As noted earlier, the initializers of structs and enums are considered virtual.

Objective-C Interoperability

The initialization model described above guarantees that objects are properly initialized before they are used, covering all of the major use cases for initialization while maintaining soundness. Objective-C has a very different initialization model with which Swift must interoperate.

Objective-C Entrypoints

Each Swift initializer definition produces a corresponding Objective-C init method. The existence of this init method allows object construction from Objective-C (both directly via `[[A alloc] init:5]` and indirectly via, e.g., `[obj initWithCoder:coder]`) and initialization of the superclass subobject when an Objective-C class inherits from a Swift class (e.g., `[super initWithCoder:coder]`).

Note that, while Swift's initializers are not inherited and cannot override, this is only true *in Swift code*. If a subclass defines an initializer with the same Objective-C selector as an initializer in its superclass, the Objective-C init method produced for the former will override the Objective-C init method produced for the latter.

Objective-C Restrictions

The emission of Objective-C init methods for Swift initializers open up a few soundness problems, illustrated here:

```
@interface A
@end

@implementation A
- init {
  return [self initWithInt:5];
```

```
}

- initWithInt:(int)x {
  // initialize me
}

- initWithString:(NSString *)s {
  // initialize me
}
@end

class B1 : A {
  var dict: NSDictionary

  init withInt(x: Int) {
    dict = []
    super.init() // loops forever, initializing dict repeatedly
  }
}

class B2 : A {
}

@interface C : B2
@end

@implementation C
@end

void getCFromString(NSString *str) {
  return [C initWithString:str]; // doesn't initialize B's dict ivar
}
```

The first problem, with `B1`, comes from `A`'s dispatched delegation to `-initWithInt:`, which is overridden by `B1`'s initializer with the same selector. We can address this problem by enforcing that superclass delegation to an Objective-C superclass initializer refer to a designated initializer of that superclass when that class has at least one initializer marked as a designated initializer.

The second problem, with `C`, comes from Objective-C's implicit inheritance of initializers. We can address this problem by specifying that init methods in Objective-C are never visible through Swift classes, making the message send `[C initWithString:str]` ill-formed. This is a relatively small Clang-side change.

Remaining Soundness Holes

Neither of the above "fixes" are complete. The first depends entirely on the adoption of a not-yet-implemented Clang attribute to mark the designated initializers for Objective-C classes, while the second is (almost trivially) defeated by passing the `-initWithString:` message to an object of type `id` or using some other dynamic reflection.

If we want to close these holes tighter, we could stop emitting Objective-C init methods for Swift initializers. Instead, we would fake the init method declarations when importing Swift modules into Clang, and teach Clang's CodeGen to emit calls directly to the Swift initializers. It would still not be perfect (e.g., some variant of the problem with `C` would persist), but it would be closer. I suspect that this is far more work than it is worth, and that the "fixes" described above are sufficient.