orphan:

Value Semantics in Swift

Abstract: Swift is the first language to take Generic Programming seriously that also has both value and reference types. The (sometimes subtle) differences between the behaviors of value and reference types create unique challenges for generic programs that we have not yet addressed. This paper surveys related problems and explores some possible solutions.

Definitions

I propose the following definitions of "value semantics" and "reference semantics."

Value Semantics

For a type with value semantics, variable initialization, assignment, and argument-passing (hereafter, "the big three operations") each create an *independently modifiable copy* of the source value that is *interchangeable with the source*. [1]

If `T` has value semantics, the `f`s below are all equivalent:

```
func f1() -> T {
   var x : T
   return x
}

func f2() -> T {
   var x : T
   var y = x
   return y  // a copy of x is equivalent to x
}

func f2a() -> T {
   var x : T
   var y : T
   y = x
   return y  // a copy of x is equivalent to x
}

func f3() -> T {
   var x : T
   var y = x
   y.mutate() // a copy of x is modifiable
   return x   // without affecting x
}

func f3a() -> T {
   var x : T
   var y : T
   y = x;
   y.mutate() // a copy of x is modifiable
   return x   // without affecting x
}

func g(_ x : T) { x.mutate() }

func f4() -> T {
   var x : T
   g(x)          // when x is passed by-value the copy
   return x      // is modifiable without affecting x
}
```

Reference Semantics

Values of a type with reference semantics are only accessible indirectly, via a reference. Although swift tries to hide this fact for simplicity, for the purpose of this discussion it is important to note that there are always *two* values in play: the value of the reference itself and that of the object being referred to (a.k.a. the target).

The programmer thinks of the target's value as primary, but it is never used as a variable initializer, assigned, or passed as a function argument. Conversely, the reference itself has full value semantics, but the user never sees or names its type. The programmer expects that copies of a reference share a target object, so modifications made via one copy are visible as side-effects through the others.

If `T` has reference semantics, the `f`s below are all equivalent:

```
func f1(_ x: T) {
   x.mutate()
   return x
}

func f2(_ x: T) -> T {
```

```
   var y = x
   y.mutate()  // mutation through a copy of x
   return x    // is visible through x
}

func f2a(_ x: T) -> T {
   var y : T
   y = x
   y.mutate()  // mutation through a copy of x
   return x    // is visible through x
}

func g(_ x : T) { x.mutate() }

func f3(_ x: T) -> T {
   g(x)        // when x is passed to a function, mutation
   return x    // through the parameter is visible through x
}
```

The Role of Mutation

It's worth noting that in the absence of mutation, value semantics and reference semantics are indistinguishable. You can easily prove that to yourself by striking the calls to `mutate()` in each of the previous examples, and seeing that the equivalences hold for any type. In fact, the fundamental difference between reference and value semantics is that **value semantics never creates multiple paths to the same mutable state**. [2]

> **`struct` vs `class`**
>
> Although `struct`s were designed to support value semantics and `class`es were designed to support reference semantics, it would be wrong to assume that they are always used that way. As noted earlier, in the absence of mutation, value semantics and reference semantics are indistinguishable. Therefore, any immutable `class` trivially has value semantics (*and* reference semantics).
>
> Second, it's easy to implement a `struct` with reference semantics: simply keep the primary value in a `class` and refer to it through an instance variable. So, one cannot assume that a `struct` type has value semantics. `Array` could be seen (depending on how you view its value) as an example of a reference-semantics `struct` from the standard library.

The Problem With Generics

The classic Liskov principle says the semantics of operations on `Duck`'s subtypes need to be consistent with those on `Duck` itself, so that functions operating on `Duck`s still "work" when passed a `Mallard`. More generally, for a function to make meaningful guarantees, the semantics of its sub-operations need to be consistent regardless of the actual argument types passed.

The type of an argument passed by-value to an ordinary function is fully constrained, so the "big three" have knowable semantics. The type of an ordinary argument passed by-reference is constrained by subtype polymorphism, where a (usually implicit) contract between base- and sub-types can dictate consistency.

However, the situation is different for functions with arguments of protocol or parameterized type. In the absence of specific constraints to the contrary, the semantics of the big three can vary.

Example

For example, there's an algorithm called `cycle_length` that measures the length of a cycle of states (e.g. the states of a pseudo-random number generator). It needs to make one copy and do in-place mutation of the state, rather than wholesale value replacement via assignment, which might be expensive.

Here's a version of cycle_length that works when state is a mutable value type:

```
func cycle_length<State>(
  _ s : State, mutate : ([inout] State) -> ()
) -> Int
  requires State : EqualityComparable
{
    State x = s     // one copy                // 1
    mutate(&x)      // in-place mutation
    Int n = 1
    while x != s {                            // 2
         mutate(&x) // in-place mutation
         ++n
    }
    return n
}
```

The reason the above breaks when the state is in a class instance is that the intended copy in line 1 instead creates a new reference to the same state, and the comparison in line 2 (regardless of whether we decide `!=` does "identity" or "value" comparison) always

succeeds.

You can write a different implementation that only works on clonable classes:

```
// Various random number generators will implement this interface
abstract class RandomNumberGenerator
  : Clonable, Equalable
{
  func nextValue() -> Int
}

func cycle_length<State>(
  _ s : State, mutate : ([inout] State) -> ()
) -> Int
  requires State : EqualityComparable, Clonable
{
    State x = s.clone()
    Int n = 1
    while ! x.equal(s) {
        etc.
}

RandomNumberGenerator x = new MersenneTwister()
print(
   cycle_length(x, (x : [inout] RandomNumberGenerator) { x.nextValue() })
)
```

You could also redefine the interface so that it works on both values and clonable classes:

```
func cycle_length<State>(
  _ s : State,
  next : (x : State) -> State,
  equal : (x : [inout] State, y : [inout] State) -> Bool
) -> Int
  requires State : EqualityComparable
{
    State x = next(s)
    Int n = 1
    while !equal(x, s) {
         x = next(x)
         ++n
    }
    return n
}
```

However, this implementation makes O(N) separate copies of the state. I don't believe there's a reasonable way write this so it works on clonable classes, non-classes, and avoids the O(N) copies. [3]

Class Identities are Values

It's important to note that the first implementation of `cycle_length` works when the state is the *identity*, rather than the *contents* of a class instance. For example, imagine a circular linked list:

```
class Node {
    constructor(Int) { next = this; prev = this }

    // link two circular lists into one big cycle.
    func join(_ otherNode : Node) -> () { ... }

    var next : WeakRef<Node> // identity of next node
    var prev : WeakRef<Node> // identity of previous node
}
```

We can measure the length of a cycle in these nodes as follows:

```
cycle_length(someNode, (x: [inout] Node){ x = x.next })
```

This is why so many generic algorithms seem to work on both `class`es and non-`class`es: `class` *identities* work just fine as values.

The Role of Moves

Further complicating matters is the fact that the big three operations can be--and often are--combined in ways that mask the value/reference distinction. In fact both of the following must be present in order to observe a difference in behavior:

1. Use of (one of) the big three operations on an object `x`, creating shared mutable state iff `x` is a reference
2. In-place mutation of `x` *while a (reference) copy is extant* and thus can be observed through the copy iff `x` is a reference.

Take, for example, `swap`, which uses variable initialization and assignment to exchange two values:

```
func swap<T>(_ lhs : [inout] T, rhs : [inout] T)
{
```

```
    var tmp = lhs   // big 3: initialization - ref copy in tmp
    lhs = rhs       // big 3: assignment     - ref copy in lhs
    rhs = tmp       // big 3: assignment     - no ref copies remain
}
```

Whether `T` is a reference type makes no observable difference in the behavior of `swap`. Why? Because although `swap` makes reference copies to mutable state, the existence of those copies is encapsulated within the algorithm, and it makes no in-place mutations.

Any such algorithm can be implemented such that copy operations are replaced by destructive *moves*, where the source value is not (necessarily) preserved. Because movability is a weaker requirement than copyability, it's reasonable to say that `swap` is built on *moves*, rather than copies, in the same way that C++'s `std::find` is built on input iterators rather than on forward iterators.

We could imagine a hypothetical syntax for moving in swift, where (unlike assignment) the value of the right-hand-side of the `<-` is not necessarily preserved:

```
var tmp <- lhs
lhs <- rhs
rhs <- tmp
```

Such operations are safe to use in generic code without regard to the differences between value- and reference- semantics. If this syntax were extended to handle function arguments, it would cover the "big three" operations:

```
f(<-x)
```

How to Build an Interesting Type with Value Semantics

Suppose we want to build a variable-sized data structure `X` with (mutable) value semantics? How do we do it?

If we make `X` a ``class`, we automatically get reference semantics, so its value must be copied before each mutation, which is tedious and error-prone. Its public mutating interface must be in terms of free functions (not methods), so that the original reference value can be passed `[inout]` and overwritten. Since there's no user access to the reference count, we can't determine that we hold the only reference to the value, so we can't optimize copy-on-write, even in single-threaded programs. In multi-threaded programs, where each mutation implies synchronization on the reference count, the costs are even higher.

If we make the type a `struct`, you have only two ways to create variable-sized data:

1. Hold a type with reference semantics as an instance variable. Unfortunately, this is really nothing new; we must still implement copy-on-write. We can, however, use methods for mutation in lieu of free functions.
2. Use discriminated unions (`union`). Interestingly, a datatype built with `union` automatically has value semantics. However, there vocabulary of efficient data structures that can be built this way is extremely limited. For example, while a singly-linked list is trivial to implement, an efficient doubly-linked list is effectively impossible.

---

[1] Technically, copies of objects with value semantics are interchangeable until they're mutated. Thereafter, the copies are interchangeable except insofar as it matters what value type they are *aggregated into*.

[2] Note that this definition *does* allow for value semantics using copy-on-write

[3] I can think of a language extension that would allow this, but it requires creating a protocol for generic copying, adding compiler magic to get both classes and structs to conform to it, and telling generic algorithm and container authors to use that protocol instead of `=`, which IMO is really ugly and probably not worth the cost.