orphan:

Values and References

Author: Dave Abrahams
Author: Joe Groff
Date: 2013-03-15

Abstract: We propose a system that offers first-class support for both value and reference semantics. By allowing--but not requiring--(instance) variables, function parameters, and generic constraints to be declared as `val` or `ref`, we offer users the ability to nail down semantics to the desired degree without compromising ease of use.

> **Note**
>
> We are aware of some issues with naming of these new keywords; to avoid chaos we discuss alternative spelling schemes in a Bikeshed section at the end of this document.

Introduction

Until recently, Swift's support for value semantics outside trivial types like scalars and immutable strings has been weak. While the recent `Clonable` proposal makes new things possible in the "safe" zone, it leaves the language syntactically and semantically lumpy, keeping interactions between value and reference types firmly outside the "easy" zone and failing to address the issue of generic programming.

This proposal builds on the `Clonable` proposal to create a more uniform, flexible, and interoperable type system while solving the generic programming problem and expanding the "easy" zone.

General Description

The general rule we propose is that most places where you can write `var` in today's swift, and also on function parameters, you can write `val` or `ref` to request value or reference semantics, respectively. Writing `var` requests the default semantics for a given type. Non-`class` types (`struct`s, tuples, arrays, `union`s) default to `val` semantics, while `class`es default to `ref` semantics. The types `val SomeClass` and `ref SomeStruct` also become part of the type system and can be used as generic parameters or as parts of tuple, array, and function types.

Because the current specification already describes the default behaviors, we will restrict ourselves to discussing the new combinations, such as `struct` variables declared with `ref` and `class` variables declared with `val`, and interactions between the two.

Terminology

When we use the term "copy" for non-`class` types, we are talking about what traditionally happens on assignment and pass-by-value. When applied to `class` types, "copy" means to call the `clone()` method, which is generated by the compiler when the user has explicitly declared conformance to the `Clonable` protocol.

When we refer to variables being "declared `val`" or "declared `ref`", we mean to include the case of equivalent declarations using `var` that request the default semantics for the type.

Unless otherwise specified, we discuss implementation details such as "allocated on the heap" as a way of describing operational semantics, with the understanding that semantics-preserving optimizations are always allowed.

When we refer to the "value" of a class, we mean the combination of values of its `val` instance variables and the identities of its `ref` instance variables.

Variables

Variables can be explicitly declared `val` or `ref`:

```
var x: Int  // x is stored by value
val y: Int  // just like "var y: Int"
ref z: Int  // z is allocated on the heap.

var q: SomeClass          // a reference to SomeClass
ref r: SomeClass          // just like "var r: SomeClass"
val s: SomeClonableClass // a unique value of SomeClonableClass type
```

Assignments and initializations involving at least one `val` result in a copy. Creating a `ref` from a `val` copies into heap memory:

```
ref z2 = x         // z2 is a copy of x's value on the heap
y = z              // z2's value is copied into y

ref z2 = z         // z and z2 refer to the same Int value
ref z3 = z.clone() // z3 refers to a copy of z's value
```

```
val t = r          // Illegal unless SomeClass is Clonable
ref u = s          // s's value is copied into u
val v = s          // s's value is copied into v
```

Standalone Types

`val`- or `ref`-ness is part of the type. When the type appears without a variable name, it can be written this way:

```
ref Int                 // an Int on the heap
val SomeClonableClass  // a value of SomeClonableClass type
```

Therefore, although it is not recommended style, we can also write:

```
var y: val Int              // just like "var y: Int"
var z: ref Int              // z is allocated on the heap.
var s: val SomeClonableClass // a unique value of type SomeClonableClass
```

Instance Variables

Instance variables can be explicitly declared `val` or `ref`:

```
struct Foo {
    var x: Int  // x is stored by-value
    val y: Int  // just like "var y: Int"
    ref z: Int  // allocate z on the heap

    var q: SomeClass          // q is a reference to SomeClass
    ref r: SomeClass          // just like "var r: SomeClass"
    val s: SomeClonableClass // clone() s when Foo is copied
}

class Bar : Clonable {
    var x: Int  // x is stored by-value
    val y: Int  // just like "var y: Int"
    ref z: Int  // allocate z on the heap

    var q: SomeClass          // q is stored by-reference
    ref r: SomeClass          // just like "var r: SomeClass"
    val s: SomeClonableClass // clone() s when Bar is clone()d
}
```

When a value is copied, all of its instance variables declared `val` (implicitly or explicitly) are copied. Instance variables declared `ref` merely have their reference counts incremented (i.e. the reference is copied). Therefore, when the defaults are in play, the semantic rules already defined for Swift are preserved.

The new rules are as follows:

- A non-`class` instance variable declared `ref` is allocated on the heap and can outlive its enclosing `struct`.
- A `class` instance variable declared `val` will be copied when its enclosing `struct` or `class` is copied. We discuss below what to do when the `class` is not `Clonable`.

Arrays

TODO: reconsider sugared array syntax. Maybe val<Int>[42] would be better

Array elements can be explicitly declared `val` or `ref`:

```
var x : Int[42]         // an array of 42 integers
var y : Int[val 42]     // an array of 42 integers
var z : Int[ref 42]     // an array of 42 integers-on-the-heap
var z : Int[ref 2][42]  // an array of 2 references to arrays
ref a : Int[42]         // a reference to an array of 42 integers
```

When a reference to an array appears without a variable name, it can be written using the usual syntax:

```
var f : () -> ref Int[42] // a closure returning a reference to an array
var b : ref Int[42]       // equivalent to "ref b : Int[42]"
```

Presumably there is also some fully-desugared syntax using angle brackets, that most users will never touch, e.g.:

```
var x : Array<Int,42>              // an array of 42 integers
var y : Array<val Int,42>          // an array of 42 integers
var z : Array<ref Int,42>          // an array of 42 integers-on-the-heap
var z : Array<ref Array<Int,42>, 2> // an array of 2 references to arrays
ref a : Array<Int,42>              // a reference to an array of 42 integers
var f : () -> ref Array<Int,42>    // a closure returning a reference to an array
var b : ref Array<Int,42>          // equivalent to "ref b : Int[42]"
```

Rules for copying array elements follow those of instance variables.

unions

Unions

Union types, like structs, have default value semantics. Constructors for the union can declare the `val`- or `ref`-ness of their associated values, using the same syntax as function parameters, described below:

```
union Foo {
  case Bar(ref bar:Int)
  case Bas(val bas:SomeClass)
}
```

Unions allow the definition of recursive types. A constructor for a union may recursively reference the union as a member; the necessary indirection and heap allocation of the recursive data structure is implicit and has value semantics:

```
// A list with value semantics--copying the list recursively copies the
// entire list
union List<T> {
  case Nil()
  case Cons(car:T, cdr:List<T>)
}

// A list node with reference semantics--copying the node creates a node
// that shares structure with the tail of the list
union Node<T> {
  case Nil()
  case Cons(car:T, ref cdr:Node<T>)
}
```

A special `union` type is the nullable type `T?`, which is sugar syntax for a generic union type `Nullable<T>`. Since both nullable refs and refs-that-are-nullable are useful, we could provide sugar syntax for both to avoid requiring parens:

```
ref? Int // Nullable reference to Int: Nullable<ref T>
ref Int? // Reference to nullable Int: ref Nullable<T>
val? SomeClass // Nullable SomeClass value: Nullable<val T>
val Int? // nullable Int: val Nullable<T> -- the default for Nullable<T>
```

Function Parameters

Function parameters can be explicitly declared `val`, or `ref`:

```
func baz(
    _ x: Int      // x is passed by-value
  , val y: Int  // just like "y: Int"
  , ref z: Int  // allocate z on the heap

  , q: SomeClass               // passing a reference
  , ref r: SomeClass           // just like "var r: SomeClass"
  , val s: SomeClonableClass) // Passing a copy of the argument
```

> **Note**
>
> We suggest allowing explicit `var` function parameters for uniformity.

Semantics of passing arguments to functions follow those of assignments and initializations: when a `val` is involved, the argument value is copied.

> **Note**
>
> We believe that `[inout]` is an independent concept and still very much needed, even with an explicit `ref` keyword. See also the Bikeshed discussion at the end of this document.

Generics

TODO: Why do we need these constraints? TODO: Consider generic classes/structs

As with an array's element type, a generic type parameter can also be bound to a `ref` or a `val` type.

> var rv = new Array<ref Int> // Create a vector of Ints-on-the-heap var vv = new Array<val SomeClass> // Create a vector that owns its SomeClasses

The rules for declarations in terms of `ref` or `val` types are that an explicit `val` or `ref` overrides any `val`- or `ref`-ness of the type parameter, as follows:

```
ref x : T // always declares a ref
val x : T // always declares a val
var x : T // declares a val iff T is a val
```

`ref` and `val` can be specified as protocol constraints for type parameters:

```
// Fill an array with independent copies of x
func fill<T:val>(_ array:[T], x:T) {
  for i in 0...array.length {
    array[i] = x
  }
}
```

Protocols similarly can inherit from `val` or `ref` constraints, to require conforming types to have the specified semantics:

```
protocol Disposable : ref {
  func dispose()
}
```

The ability to explicitly declare `val` and `ref` allow us to smooth out behavioral differences between value and reference types where they could affect the correctness of algorithms. The continued existence of `var` allows value-agnostic generic algorithms, such as `swap`, to go on working as before.

Non-Copyability

A non-`Clonable` `class` is not copyable. That leaves us with several options:

1. Make it illegal to declare a non-copyable `val`
2. Make non-copyable `val`s legal, but not copyable, thus infecting their enclosing object with non-copyability.
3. Like #2, but also formalize move semantics. All `val`s, including non-copyable ones, would be explicitly movable. Generic `var` parameters would probably be treated as movable but non-copyable.

We favor taking all three steps, but it's useful to know that there are valid stopping points along the way.

Default Initialization of ref

TODO

Array

TODO: Int[...], etc.

Equality and Identity

TODO

Why Expand the Type System?

TODO

Why do We Need `[inout]` if we have `ref`?

TODO

Why Does the Outer Qualifier Win?

TODO

Objective-C Interoperability

Clonable Objective-C classes

In Cocoa, a notion similar to cloneability is captured in the `NSCopying` and `NSMutableCopying` protocols, and a notion similar to `val` instance variables is captured by the behavior of `(copy)` properties. However, there are some behavioral and semantic differences that need to be taken into account. `NSCopying` and `NSMutableCopying` are entangled with Foundation's idiosyncratic management of container mutability: `-[NSMutableThing copy]` produces a freshly copied immutable `NSThing`, whereas `-[NSThing copy]` returns the same object back if the receiver is already immutable. `-[NSMutableThing mutableCopy]` and `-[NSThing mutableCopy]` both return a freshly copied `NSMutableThing`. In order to avoid requiring special case Foundation-specific knowledge of whether class types are notionally immutable or mutable, we propose this first-draft approach to mapping the Cocoa concepts to `Clonable`:

- If an Objective-C class conforms to `NSMutableCopying`, use the `-mutableCopyWithZone:` method to fulfill the Swift `Clonable` concept, casting the result of `-mutableCopyWithZone:` back to the original type.
- If an Objective-C class conforms to `NSCopying` but not `NSMutableCopying`, use `-copyWithZone:`, also casting the result back to the original type.

This is suboptimal for immutable types, but should work for any Cocoa class that fulfills the `NSMutableCopying` or `NSCopying` contracts without requiring knowledge of the intended semantics of the class beyond what the compiler can see.

Objective-C `(copy)` properties should behave closely enough to Swift `val` properties to be able to vend Objective-C `(copy)`

properties to Swift as `val` properties, and vice versa.

Objective-C protocols

In Objective-C, only classes can conform to protocols, and the `This` type is thus presumed to have references semantics. Swift protocols imported from Objective-C or declared as `[objc]` could be conformed to by `val` types, but doing so would need to incur an implicit copy to the heap to create a `ref` value to conform to the protocol.

How This Design Improves Swift

1. You can choose semantics at the point of use. The designer of a type doesn't know whether you will want to use it via a reference; she can only guess. You might *want* to share a reference to a struct, tuple, etc. You might *want* some class type to be a component of the value of some other type. We allow that, without requiring awkward explicit wrapping, and without discarding the obvious defaults for types that have them.
2. We provide a continuum of strictness in which to program. If you're writing a script, you can go with `var` everywhere: don't worry; be happy. If you're writing a large-scale program and want to be very sure of what you're getting, you can forbid `var` except in carefully-vetted generic functions. The choice is yours.
3. We allow generic programmers to avoid subtle semantic errors by explicitly specifying value or reference semantics where it matters.
4. We move the cases where values and references interact much closer to, and arguably into, the "easy" zone.

How This Design Beats Rust/C++/C#/etc.

- Simple programs stay simple. Rust has a great low-level memory safety story, but it comes at the expense of ease-of-use. You can't learn to use that system effectively without confronting two kinds of pointer, named lifetimes, borrowing managed boxes and rooting, etc. By contrast, there's a path to learning swift that postpones the `val`/`ref` distinction, and that's pretty much *all* one must learn to have a complete understanding of the object model in the "easy" and "safe" zones.
- Simple programs stay safe. C++ offers great control over everything, but the sharp edges are always exposed. This design allows programmers to accomplish most of what people want to with C++, but to do it safely and expressively. As with the rest of Swift, the sharp edges are still available as an opt-in feature, and without harming the rest of the language.
- Unlike C++, types meant to be reference types, supporting inheritance, aren't copyable by default. This prevents inadvertent slicing and wrong semantics.
- By retaining the `class` vs. `struct` distinction, we give type authors the ability to provide a default semantics for their types and avoid confronting their users with a constant `T*` vs. `T` choice like C/C++.
- C# also provides a `class` vs. `struct` distinction with a generics system, but it provides no facilities for nontrivial value semantics on struct types, and the only means for writing generic algorithms that rely on value or reference semantics is to apply a blunt `struct` or `class` constraint to type parameters and limit the type domain of the generic. By generalizing both value and reference semantics to all types, we allow both for structs with interesting value semantics and for generics that can reliably specify and use value or reference semantics without limiting the types they can be used with.

`structs` Really Should Have Value Semantics

It is *possible* to build a struct with reference semantics. For example,

..parsed-literal:

```
struct XPair
{
   constructor() {
       // These Xs are notionally **part of my value**
       first = new X
       second = new X
   }
   **ref** first : X
   **ref** second : X
}
```

However, the results can be surprising:

```
val a : XPair  // I want an independent value, please!
val b = a          // and a copy of that value
a.first.mutate()   // Oops, changes b.first!
```

If `XPair` had been declared a class,

```
val a : XPair      // I want an independent value, please!
```

would only compile if `XPair` was also `Clonable`, thereby protecting the user's intention to create an independent value

Getting the `ref` out of a `class` instance declared `val`

A `class` instance is always accessed through a reference, but when an instance is declared `val`, that reference is effectively hidden

behind the `val` wrapper. However, because `this` is passed to `class` methods as a reference, we can unwrap the underlying `ref` as follows:

```
val x : SomeClass

extension SomeClass {
  func get_ref() { return this }
}

ref y : x.get_ref()
y.mutate()          // mutates x
```

Teachability

By expanding the type system we have added complexity to the language. To what degree will these changes make Swift harder to learn?

We believe the costs can be mitigated by teaching plain `var` programming first. The need to confront `val` and `ref` can be postponed until the point where students must see them in the interfaces of library functions. All the same standard library interfaces that could be expressed before the introduction of `val` and `ref` can still be expressed without them, so this discovery can happen arbitrarily late in the game. However, it's important to realize that having `val` and `ref` available will probably change the optimal way to express the standard library APIs, and choosing where to use the new capabilities may be an interesting balancing act.

(Im)Mutability

We have looked, but so far, we don't think this proposal closes (or, for that matter, opens) the door to anything fundamentally new with respect to declared (im)mutability. The issues that arise with explicit `val` and `ref` also arise without them.

Bikeshed

There are a number of naming issues we might want to discuss. For example:

- `var` is only one character different from `val`. Is that too confusable? Syntax highlighting can help, but it might not be enough.
  - What about `let` as a replacement for `var`? There's always the dreaded `auto`.
  - Should we drop `let`/`var`/`auto` for ivars, because it "just feels wrong" there?
- `ref` is spelled like `[inout]`, but they mean very different things
  - We don't think they can be collapsed into one keyword: `ref` requires shared ownership and is escapable and aliasable, unlike `[inout]`.
  - Should we spell `[inout]` differently? I think at a high level it means something like "`[rebind]` the name to a new value."
- Do we want to consider replacing `struct` and/or `class` with new names such as `valtype` and `reftype`? We don't love those particular suggestions. One argument in favor of a change: `struct` comes with a strong connotation of weakness or second-class-ness for some people.