orphan:

Clonable

Author: Dave Abrahams
Author: Joe Groff
Date: 2013-03-21
Edition: 2

> **Warning**
>
> This proposal was rejected. We decided not to introduce a language-level copying mechanism for classes.

Abstract: to better support the creation of value types, we propose a "magic" `Clonable` protocol and an annotation for describing which instance variables should be cloned when a type is copied. This proposal **augments revision 1** of the Clonable proposal with our rationale for dropping our support for `val` and `ref`, a description of the programming model for generics, and a brief discussion of equality. It is **otherwise unchanged**.

Rationale

By eliminating `val`, we lose the easy creation of runtime-polymorphic value types. Instead of merely writing:

```
val x : MyClass
```

one has to engage in some kind of wrapping and forwarding:

```
struct MyClassVal {
  var [clone] value : MyClass

  constructor(x : A, y : B) {
     value = new MyClass(x, y)
  }

  func someFunction(_ z : C) -> D {
    return value.someFunction(z)
  }

  // ...etc...
}
```

Although such wrapping is awful, this is not the only place where one would want to do it. Therefore, some kind of ability to forward an entire interface wholesale could be added as a separate extension (getter/setter for `This`?), which would solve more problems. Then it would be easy enough to write the wrapper as a generic struct and `Val<T>` would be a reality.

By eliminating `ref`, we lose the easy creation of references to value types. However, among those who prefer programming with values, having an explicit step for dereferencing might make more sense, so we could use this generic class:

```
class Reference<T> { value : T }
```

If explicit dereferencing isn't desired, there's always manual (or automatic, if we add that feature) forwarding.

By dropping `val` we also lose some terseness aggregating `class` contents into `struct`s. However, since `ref` is being dropped there's less call for a symmetric `val`. The extra "cruft" that `[clone]` adds actually seems appropriate when viewed as a special bridge for `class` types, and less like a penalty against value types.

Generics

There is actually a straightforward programming model for generics. If you want to design a generic component where a type parameter `T` binds to both `class`es and non-`class` types, you can view `T` as a value type where--as with C pointers--the value is the reference rather than the object being referred to.

Of course, if `T` is only supposed to bind to `class`es, a different programming model may work just as well.

Implications for Equality

We think the programming model suggested for generics has some pretty strong implications for equality of `class`es: `a == b` must return true iff `a` and `b` refer to the same object.

Details *(unchanged from Revision 1)*

When a type with reference semantics `R` is to be used as a part of (versus merely being referred-to-by) a type with value semantics `V`, a new annotation, `[clone]` can be used to indicate that the `R` instance variable should be `clone()`d when `V` is copied.

A `class` can be `clone()`d when it implements the built-in `Clonable` protocol:

```
protocol Clonable {
   func clone() -> Self { /* see below */ }
}
```

The implementation of `clone()` (which will be generated by the compiler and typically never overridden) performs a primitive copy of all ordinary instance variables, and a `clone()` of all instance variables marked `[clone]`:

```
class FooValue : Clonable  {}

class Bar {}

class Foo : Clonable {
    var count : Int
    var [clone] myValue : FooValue
    var somethingIJustReferTo : Bar
}

struct Baz {
    var [clone] partOfMyValue : Foo
    var anotherPart : Int
    var somethingIJustReferTo : Bar
}
```

When a `Baz` is copied by any of the "big three" operations (variable initialization, assignment, or function argument passing), even as part of a larger `struct`, its `[clone]` member is `clone()`d. Because `Foo` itself has a `[clone]` member, that is `clone()`d also. Therefore copying a `Baz` object `clone()`s a `Foo` and `clone()`ing a `Foo` `clone()`s a `FooValue`.

All `struct`s are `Clonable` by default, with `clone()` delivering ordinary copy semantics. Therefore,

```
var x : Baz
var y = x.clone()
```

is equivalent to

```
var x : Baz
var y = x
```

Note that `Clonable` is the first protocol with a default implementation that can't currently be written in the standard library (though arguably we'd like to add the capability to write that implementation).