orphan:

> **Warning**
>
> This proposal was rejected. We ultimately decided to keep Array as a dual-representation struct.

Bridging Container Protocols to Class Clusters

I think that attempting to bridge `NSArray` to a concrete type like `Array<T>` will be a poor compromise, losing both the flexibility of NSArray's polymorphism and the performance afforded by `Array<T>`'s simplicity. Here's what I propose instead:

- Rename our current `Array` back to `Vector` or perhaps something like `ContiguousArray`.
- Redefine `Array` as a refinement of the `Collection` protocol that has integer indices.
- Implement an `ArrayOf<T>` generic container, like `AnyIterator` and `AnySequence`, that can hold an arbitrary type conforming to `Array`.
- Bridge `NSArray` from ObjC to Swift `ArrayOf<AnyObject>` with value semantics.
- Bridge `Array`-conforming types with class element types in Swift to ObjC as `NSArray`.

Although I'll be talking about arrays in this proposal, I think the same approach would work for `NSDictionary` and `NSSet` as well, mapping them to generic containers for associative map and unordered container protocols respectively.

NSArray vs Array

Despite their similar names, `NSArray` and Swift's `Array` have fundamentally incompatible design goals. As the root of a class cluster, `NSArray` provides abstraction over many underlying data structures, trading weaker algorithmic guarantees for better representational flexibility and implementation encapsulation. Swift's `Array`, on the other hand, is intended to be a direct representation of a contiguous region of memory, more like a C array or C++'s `vector`, minimizing abstraction in order to provide tight algorithmic guarantees. Many `NSArray` implementations are lazy, such as those over KVO properties or Core Data aggregates, and transforming them to concrete `Array`s would have unintended semantic effects. And on the other side, the overhead of having to accommodate an arbitrary `NSArray` implementation inside `Array` destroys `Array` as a simple, high-performance container. Attempting to bridge these two types will result in an unattractive compromise to both sides, weakening the algorithmic guarantees of Array while forgoing the full flexibility of `NSArray`.

"Array" as a Refinement of the Collection Protocol

Swift's answer to container polymorphism is its generics system. The `Collection` protocol provides a common interface to indexable containers that can be used generically, which is exactly what `NSArray` provides in Cocoa for integer-indexable container implementations. `Array` could be described as a refinement of `Collection` with integer indices:

```
protocol Array : Collection {
  where IndexType == Int
}
protocol MutableArray : MutableCollection {
  where IndexType == Int
}
```

The familiar `NSArray` API can then be exposed using default implementations in the `Array` protocol, or perhaps even on the more abstract `Collection` and `Sequence` protocols, and we can bridge `NSArray` in a way that plays nicely with generic containers.

This naming scheme would of course require us to rename the concrete `Array<T>` container yet again. `Vector` is an obvious candidate, albeit one with a C++-ish bent. Something more descriptive like `ContiguousArray` might feel more Cocoa-ish.

The ArrayOf<T> Type

Although the language as implemented does not yet support protocol types for protocols with associated types, DaveA devised a technique for implementing types that provide the same effect in the library, such as his `AnyIterator<T>` and `AnySequence<T>` containers for arbitrary `Stream` and `Sequence` types. This technique can be extended to the `Array` protocol, using class inheritance to hide the concrete implementing type behind an abstract base:

```
// Abstract base class that forwards the Array protocol
class ArrayOfImplBase<T> {
  var startIndex: Int { fatal() }
  var endIndex: Int { fatal() }

  func __getitem__(_ i: Int) -> T { fatal() }

  // For COW
  func _clone() -> Self { fatal() }
}

// Concrete derived class containing a specific Array implementation
class ArrayOfImpl<T, ArrayT: Array where ArrayT.Element == T>
  : ArrayOfImplBase<T>
{
```

```
  var value: ArrayT
  var startIndex: Int { return value.startIndex }
  var endIndex: Int { return value.endIndex }
  func __getitem__(_ i: Int) -> T { return __getitem__(i) }

  // For COW
  func _clone() -> Self { return self(value) }
}

// Wrapper type that uses the base class to erase the concrete type of
// an Array
struct ArrayOf<T> : Array {
  var value: ArrayOfImplBase<T>

  var startIndex: Int { return value.startIndex }
  var endIndex: Int { return value.endIndex }
  func __getitem__(_ i: Int) -> T { return value.__getitem__(i) }

  init<ArrayT : Array where ArrayT.Element == T>(arr: ArrayT) {
    value = ArrayOfImpl<T, ArrayT>(arr)
  }
}
```

The mutable variant can use COW optimization to preserve value semantics:

```
struct MutableArrayOf<T> : MutableArray {
  /* ...other forwarding methods... */

  func __setitem__(_ i: Int, x: T) {
    if !isUniquelyReferenced(value) {
      value = value._clone()
    }
    value.__setitem__(i, x)
  }
}
```

Bridging `NSArray` into Swift

We could simply make `NSArray` conform to `Array`, which would be sufficient to allow it to be stored in an `ArrayOf<AnyObject>` container. However, a good experience for `NSArray` still requires special-case behavior. In particular, `NSArray` in Cocoa is considered a value class, and best practice dictates that it be defensively `copy`-ed when used. In Swift, we should give bridged NSArrays COW value semantics by default, like `NSString`. One way to handle this is by adding a case to the `ArrayOf` implementation, allowing it to either contain a generic value or an `NSArray` with COW semantics.

Bridging Swift Containers to `NSArray`

We could have an implicit conversion to `NSArray` from an arbitrary type conforming to `Array` with a class element type, allowing ObjC APIs to work naturally with generic Swift containers. Assuming we had support for `conversion_to` functions, it could look like this:

```
class NSArrayOf<ArrayT: Array where ArrayT.Element : class> : NSArray {
  /* ...implement NSArray methods... */
}

extension NSArray {
  @conversion_to
  func __conversion_to<
    ArrayT: Array where ArrayT.Element : class
  >(arr: ArrayT) -> NSArray {
    return NSArrayOf<ArrayT>(arr)
  }
}
```

`NSArray` has reference semantics in ObjC, which is a mismatch with Swift's value semantics, but because `NSArray` is a value class, this is probably not a problem in practice, because it will be `copy`-ed as necessary as a best practice. There also needs to be a special case for bridging an `ArrayOf<T>` that contains an `NSArray`; such a container should be bridged directly back to the underlying unchanged `NSArray`.