orphan:

Bridging Swift Arrays to/from Cocoa

Authors: Chris Lattner, Joe Groff, Dave Abrahams

Summary: Unifying a fast C-style array with a Cocoa class cluster that can represent arbitrarily complex data structures is challenging. In a space where no approach satisfies all desires, we believe we've found a good compromise.

Basic Requirements

A successfully-bridged array type would be both "great for Cocoa" and "great for C."

Being "great for Cocoa" means this must work and be efficient:

```
var a = [cocoaObject1, cocoaObject2]
someCocoaObject.takesAnNSArray(a)

func processViews(_ views: [AnyObject]) { ... }
var b = someNSWindow.views // views is an NSArray
processViews(b)

var c: [AnyObject] = someNSWindow.views
```

Being "great For C" means that an array created in Swift must have C-like performance and be representable as a base pointer and length, for interaction with C APIs, at zero cost.

Proposed Solution

`Array<T>`, a.k.a. `[T]`, is notionally an `enum` with two cases; call them `Native` and `Cocoa`. The `Native` case stores a `ContiguousArray`, which has a known, contiguous buffer representation and O(1) access to the address of any element. The `Cocoa` case stores an `NSArray`.

`NSArray` bridges bidirectionally in O(1) [1] to `[AnyObject]`. It also implicitly converts in to `[T]`, where T is any class declared to be `@objc`. No dynamic check of element types is ever performed for arrays of `@objc` elements; instead we simply let `objc_msgSend` fail when `T`'s API turns out to be unsupported by the object. Any `[T]`, where T is an `@objc` class, converts implicitly to NSArray.

Optimization

Any type with more than one representation naturally penalizes fine-grained operations such as indexing, because the cost of repeatedly branching to handle each representation becomes significant. For example, the design above would pose significant performance problems for arrays of integers, because every subscript operation would have to check to see if the representation is an NSArray, realize it is not, then do the constant time index into the native representation. Beyond requiring an extra check, this check would disable optimizations that can provide a significant performance win (like auto-vectorization).

However, the inherent limitations of `NSArray` mean that we can often know at compile-time which representation is in play. So the plan is to teach the compiler to optimize for the `Native` case unless the element type is an `@objc` class or AnyObject. When `T` is statically known not to be an `@objc` class or AnyObject, it will be possible to eliminate the `Cocoa` case entirely. When generating code for generic algorithms, we can favor the `Native` case, perhaps going so far as to specialize for the case where all parameters are non-`@objc` classes. This will give us C-like performance for array operations on `Int`, `Float`, and other `struct` types [2].

To implement this, we'll need to implement a new generic builtin, something along the lines of "`Builtin.couldBeObjCType<T>()`", which returns a `Builtin.Int1` value. SILCombine and IRGen should eagerly fold this to "0" iff `T` is known to be a protocol other than AnyObject, if it is known to be a non-`@objc` class, or if it is known to be any struct, enum or tuple. Otherwise, the builtin is left alone, and if it reaches IRGen, IRGen should conservatively fold it to "1". In the common case where `Array<Element>` is inlined and specialized, this will allow us to eliminate all of the overhead in the important C cases.

Opportunity Feature

For hardcore systems programming, we can expose `ContiguousArray` as a user-consumable type. That will allow programmers who don't care about Cocoa interoperability to avoid ever paying the cost of branching on representation. This type would not bridge transparently to Array, but could be useful if you need an array of Objective-C type, don't care about NSArray compatibility, and care deeply about performance.

Other Approaches Considered

We considered an approach where conversions between `NSArray` and native Swift `Array` were entirely manual and quickly ruled it out as failing to satisfy the requirements.

We considered another promising proposal that would make `[T]` a (hand-rolled) existential wrapper type. Among other things, we felt this approach would expose multiple array types too prominently and would tend to "bless" an inappropriately-specific protocol as the generic collection interface (for example, a generic collection should not be indexable with `Int`).

We also considered several variants of the approach we've proposed here, tuning the criteria by which we'd decide to optimize for a `Native` representation.

---

[1] Value semantics dictates that when bridging an `NSArray` into Swift, we invoke its `copy` method. Calling `copy` on an immutable `NSArray` can be almost cost-free, but a mutable `NSArray` *will* be physically copied. We accept that copy as the cost of doing business.

[2] Of course, by default, array bounds checking is enabled. C does not include array bounds checks, so to get true C performance in all cases, these will have to be disabled.