orphan:

Optimizing Accessors in Swift

Definitions

An abstract storage declaration is a language construct that declares a means of accessing some sort of abstract entity. I'll just say "storage" hereafter.

Swift provides three storage declarations:

- a single named entity, called a *variable* and declared with `var`
- a single named entity which can never be reassigned, called a *constant* and declared with `let`
- a compound unnamed entity accessed with an index, called a *subscript* and declared with `subscript`

These features are similar to those in other languages. Swift notably lacks compound named entities, such as C#'s indexed properties; the design team intentionally chose to favor the use of named single entities of subscriptable type.

It's useful to lump the two kinds of single named entities together; I'll just call them both "variables" hereafter.

Subscripts must always be instance type members. When a variable is a type member, it's called a *property*.

When I say that these entities are *abstract*, I mean that they're not directly tied to any particular implementation. All of them may be backed directly by memory storing values of the entity's type, or they may simply provide a way of invoking arbitrary code to access a logical memory, or they may be a little of both.

Full-value accesses

All accesses to storage, no matter the implementation, can be performed with two primitive operations:

- a full-value load, which creates a new copy of the current value
- a full-value store, which overwrites the current value with a different, independent value

A function which implements a full-value load is called a *getter*; a full-value store, a *setter*.

An operation which calls for a full-value load into a temporary, then a modification of the temporary, then a full-value store of the temporary into the original entity, is called a *write-back*.

Implementing accesses with full-value accesses introduces two problems: subobject clobbering and performance.

Subobject clobbering

Subobject clobbering is a semantic issue. It occurs when there are two changes to an entity in flight at once, as in:

```
swap(&point.x, &point.y)
```

(By "at once", I mean synchronously. Unlike Java, Swift is willing to say that an *asynchronous* simultaneous access (mutating or not) to an entity that's being modified is completely undefined behavior, with any sort of failure permitted, up to and including memory corruption and crashes. As Swift transitions towards being a systems language, it can selectively choose to define some of this behavior, e.g. allowing racing accesses to different elements of an array.)

Subobject clobbering is a problem with user expectation. Users are unlikely to write code that intentionally modifies two obviously overlapping entities, but they might very reasonably write code that modifies two "separate" sub-entities. For example, they might write code to swap two properties of a struct, or two elements of an array. The natural expansion of these subobject accesses using whole-object accesses generates code that "loses" the changes made to one of the objects:

```
var point0 = point
var x = point0.x
var point1 = point
var y = point1.y
swap(&x, &y)
point1.y = y
point = point1
point0.x = x
point = point0
```

Note that `point.y` is left unchanged.

Local analysis

There are two straightforward solutions to subobject clobbering, both reliant on doing local analysis to recognize two accesses which obviously alias (like the two references to `point` in the above example). Once you've done this, you can either:

- Emit an error, outlawing such code. This is what Swift currently does (but only when the aliasing access must be implemented with full-value loads and stores).
- Use a single temporary, potentially changing the semantics.

It's impossible to guarantee the absence of subobject clobbering with this analysis without extremely heavy-handed languages

changes. Fortunately, subobject clobbering is "only" a semantics issue, not a memory-safety issue, at least as long as it's implemented with full-value accesses.

Reprojection

A more general solution is to re-project the modified subobject from scratch before writing it back. That is, you first acquire an initial value like normal for the subobject you wish to modify, then modify that temporary copy in-place. But instead of then recursively consuming all the intermediate temporaries when writing them back, you drop them all and recompute the current value, then write the modified subobject to it, then write back all the way up. That is:

```
var point0 = point
var x = point0.x
var point1 = point
var y = point1.y
swap(&x, &y)
point1 = point      // reload point1
point1.y = y
point = point1
point0 = point      // reload point0
point0.x = x
point = point0
```

In this example, I've applied the solution consistently to all accesses, which protects against unseen modifications (e.g. during the call to `swap`) at the cost of performing two extra full-value loads.

You can heuristically lower this by combining it with a simple local analysis and only re-projecting when writing back to other l-values besides the last. In other words, generate code that will work as long as the entity is not modified behind abstraction barriers or through unexpected aliases:

```
var point0 = point
var x = point0.x
var point1 = point
var y = point1.y
swap(&x, &y)
point1.y = y        // do not reload point1
point = point1
point0 = point      // reload point0
point0.x = x
point = point0
```

Note that, in either solution, you've introduced extra full-value loads. This may be quite expensive, and it's not guaranteed to be semantically equivalent.

Performance

There are three major reasons why full-value accesses are inefficient.

Unnecessary subobject accesses

The first is that they may load or store more than is necessary.

As an obvious example, imagine a variable of type `(Int, Int)`; even if my code only accesses the first element of the tuple, full-value accesses force me to read or write the second element as well. That means that, even if I'm purely overwriting the first element, I actually have to perform a full-value load first so that I know what value to use for the second element when performing the full-value store.

Additionally, while unnecessarily loading the second element of an `(Int, Int)` pair might seem trivial, consider that the tuple could actually have twenty elements, or that the second element might be non-trivial to copy (e.g. if it's a retainable pointer).

Abstraction barriers

A full-value load or store which you can completely reason about is one thing, but if it has to be performed as a call, it can be a major performance drag.

For one, calls do carry a significant amount of low-level overhead.

For another, optimizers must be extremely conservative about what a call might do. A retainable pointer might have to be retained and later released purely to protect against the possibility that a getter might, somehow, cause the pointer to otherwise be deallocated.

Furthermore, the conventions of the call might restrict performance. One way or another, a getter for a retainable pointer generally returns at +1, meaning that as part of the return, it is retained, forcing the caller to later release. If the access were instead direct to memory, this retain might be avoidable, depending on what the caller does with the pointer.

Copy-on-write

These problems are compounded by copy-on-write (COW) types. In Swift, a copy-on-write value embeds an object reference. Copying the value has low immediate cost, because it simply retains the existing reference. However, modifying a value requires the

reference to be made unique, generally by copying the data held by the value into a fresh object. I'll call this operation a *structural copy* in an effort to avoid the more treacherous term "deep copy".

COW types are problematic with full-value accesses for several reasons.

First, COW types are often used to implement aggregates and thus often have several distinguishable subobjects which users are likely to think of as independent. This heightens the dangers of subobject clobbering.

Second, a full-value load of a COW type implies making the object reference non-unique. Changing the value at this point will force a structural copy. This means that modifying a temporary copy has dramatically worse performance compared to modifying the original entity in-place. For example:

```
window.name += " (closing)"
```

If `&window.name` can be passed directly to the operator, and the string buffer is uniquely referenced by that string, then this operation may be as cheap as copying a few characters into the tail of the buffer. But if this must be done with a write-back, then the temporary will never have a unique reference, and there will always be an unneeded structural copy.

Conservative access patterns

When you know how storage is implemented, it's straightforward to generate an optimal access to it. There are several major reasons why you might not know how a storage declaration is implemented, though:

- It might be an abstract declaration, not a concrete declaration. Currently this means a protocol member, but Swift may someday add abstract class members.
- It might be a non-final class member, where the implementation you can see is potentially overridable by a subclass.
- It might be a resilient declaration, where you know only that the entity exists and know nothing statically about its implementation.

In all of these cases, you must generate code that will handle the worst possible case, which is that the entity is implemented with a getter and a setter. Therefore, the conservative access pattern includes opaque getter and setter functions.

However, for all the reasons discussed above, using unnecessary full-value accesses can be terrible for performance. It's really bad if a little conservatism --- e.g. because Swift failed to devirtualize a property access --- causes asymptotic inefficiencies. Therefore, Swift's native conservative access pattern also includes a third accessor which permits direct access to storage when possible. This accessor is called `materializeForSet`.

`materializeForSet` receives an extra argument, which is an uninitialized buffer of the value type, and it returns a pointer and a flag. When it can provide direct access to storage for the entity, it constructs a pointer to the storage and returns false. When it can't, it performs a full-value load into the buffer and returns true. The caller performs the modification in-place on the returned pointer and then, if the flag is true, passes the value to the setter.

The overall effect is to enable direct storage access as a dynamic optimization when it's impossible as a static optimization.

For now, `materializeForSet` is always automatically generated based on whether the entity is implemented with a computed setter. It is possible to imagine data structures that would benefit from having this lifted to a user-definable feature; for example, a data structure which sometimes holds its elements in memory but sometimes does not.

`materializeForSet` can provide direct access whenever an address for the storage can be derived. This includes when the storage is implemented with a `mutableAddress` accessor, as covered below. Observing accessors currently prevent `materializeForSet` from offering direct access; that's fixable for `didSet` using a slightly different code pattern, but `willSet` is an inherent obstacle.

Independent of any of the other optimizations discussed in this whitepaper, `materializeForSet` had the potential to immediately optimize the extremely important case of mutations to COW values in un-devirtualized class properties, with fairly minimal risk. Therefore, `materializeForSet` was implemented first, and it shipped in Xcode 6.1.

Direct access at computed addresses

What entities can be directly accessed in memory? Non-computed variables make up an extremely important set of cases; Swift has enough built-in knowledge to know that it can provide direct access to them. But there are a number of other important cases where the address of an entity is not built-in to the compiler, but where direct access is nonetheless possible. For example, elements of a simple array always have independent storage in memory. Most benchmarks on arrays would profit from being able to modify array elements in-place.

There's a long chain of proposals in this area, many of which are refinement on previous proposals. None of these proposals has yet shipped in Xcode.

Addressors

For something like a simple array (or any similar structure, like a deque) which is always backed by a buffer, it makes sense for the implementor to simply define accessors which return the address of the element. Such accessors are called *addressors*, and there are two: `address` and `mutableAddress`.

The conservative access pattern can be generated very easily from this: the getter calls `address` and loads from it, the setter calls `mutableAddress` and stores to it, and `materializeForSet` provides direct access to the address returned from

`mutableAddress`.

If the entity has type `T`, then `address` returns an `UnsafePointer<T>` and `mutableAddress` returns an `UnsafeMutablePointer<T>`. This means that the formal type of the entity must exactly match the formal type of the storage. Thus, the standard subscript on `Dictionary<K,V>` cannot be implemented using addressors, because the formal type of the entity is `V?`, but the backing storage holds a `V`. (And this is in keeping with user expectations about the data structure: assigning `nil` at a key is supposed to erase any existing entry there, not create a new entry to hold `nil`.)

This simple addressor proposal was the first prong of our efforts to optimize array element access. Unfortunately, while it is useful for several other types (such as `ContiguousArray` and `UnsafeMutablePointer`), it is not flexible enough for the `Array` type.

Mixed addressors

Swift's chief `Array` type is only a simple array when it is not interacting with Objective-C. Type bridging requires `Array` to be able to store an immutable `NSArray` instance, and the `NSArray` interface does not expose the details of how it stores elements. An `NSArray` is even permitted to dynamically generate its values in its `objectAtIndex:` method. And it would be absurd for `Array` to perform a structural copy during a load just to make non-mutating accesses more efficient! So the load access pattern for `Array`'s subscript declaration must use a getter.

Fortunately, this requirement does not preclude using an addressor for mutating accesses. Mutations to `Array` always transition the array to a unique contiguous buffer representation as their first step. This means that the subscript operator can sensibly return an address when it's used for the purposes of mutation: in other words, exactly when `mutableAddress` would be invoked.

Therefore, the second prong of our efforts to optimize array element access was to allow entities to be implemented with the combination of a `get` accessor and a `mutableAddress` accessor. This is straightforward in the user model, where it simply means lifting a restriction. It's more complex behind the scenes because it broke what was previously a clean conceptual division between "physical" and "logical" l-values.

Mixed addressors have now been adopted by `Array` to great success. As expected, they substantially improved performance mutating COW array elements. But they also fix an important instance of subobject clobbering, because modifications to different subobjects (notably, different elements of the same array) can occur simultaneously by simply projecting out their addresses in the unique buffer. For example, this means that it's possible to simply swap two elements of an array directly:

```
swap(&array[i], &array[j])

// Expanded:
array.transitionToUniquelyReferenced()
let address_i = array.buffer.storage + i
array.transitionToUniquelyReferenced()
let address_j = array.buffer.storage + j
swap(address_i, address_j)
```

Mixed addressors weren't completely implemented until very close to the Xcode 6.1 deadline, and they changed code-generation patterns enough to break a number of important array-specific optimizations. Therefore, the team sensibly decided that they were too risky for that release, and that there wasn't enough benefit from other applications to justify including any of the addressor work.

In a way, that was a fortunate decision, because the naive version of addressors implemented so far in Swift creates a safety hole which would otherwise have been exposed to users.

Memory unsafety of addressors

The semantics and memory safety of operations on COW types rely on a pair of simple rules:

- A non-mutating operation must own a reference to the buffer for the full course of the read.
- A mutating operation must own a unique reference to the buffer for the full course of the mutation.

Both rules tend to be naturally satisfied by the way that operations are organized into methods. A value must own a reference to its buffer at the moment that a method is invoked on it. A mutating operation immediately transitions the buffer to a unique reference, performing a structural copy if necessary. This reference will remain valid for the rest of the method as long as the method is *atomic*: as long as it does not synchronously invoke arbitrary user code.

(This is a single-threaded notion of atomicity. A second thread which modifies the value simultaneously can clearly invalidate the assumption. But that would necessarily be a data race, and the language design team is willing to say that such races have fully undefined behavior, and arbitrary consequences like memory corruption and crashes are acceptable in their wake.)

However, addressors are not atomic in this way: they return an address to the caller, which may then interleave arbitrary code before completing the operation. This can present the opportunity for corruption if the interleaved code modifies the original value. Consider the following code:

```
func operate(value: inout Int, count: Int) { ... }

var array: [Int] = [1,2,3,4]
operate(&array[0], { array = []; return 0 }())
```

The dynamic sequence of operations performed here will expand like so:

```
var array: [Int] = [1,2,3,4]
let address = array.subscript.mutableAddress(0)
array = []
operate(address, 0)
```

The assignment to `array` within the closure will release the buffer containing `address`, thus passing `operate` a dangling pointer.

Nor can this be fixed with a purely local analysis; consider:

```
class C { var array: [Int] }
let global_C = C()

func assign(value: inout Int) {
  C.array = []
  value = 0
}

assign(&global_C.array[0])
```

Fixing the memory safety hole

Conceptually, the correct fix is to guarantee that the rules are satisfied by ensuring that the buffer is retained for the duration of the operation. Any interleaving modifications will then see a non-uniquely-referenced buffer and perform a structural copy:

```
// Project the array element.
let address = array.subscript.mutableAddress(0)

// Remember the new buffer value and keep it retained.
let newArrayBuffer = array.buffer
retain(newArrayBuffer)

// Reassign the variable.
release(array.buffer)
array.buffer = ...

// Perform the mutation.  These changes will be silently lost, but
// they at least won't be using deallocated memory.
operate(address, 0)

// Release the "new" buffer.
release(newArrayBuffer)
```

Note that this still leaves a semantic hole if the original value is copied in interleaving code before the modification, because the subsequent modification will be reflected in the copy:

```
// Project the array element.
let address = array.subscript.mutableAddress(0)

// Remember the new buffer value and keep it retained.
let newArrayBuffer = array.buffer
retain(newArrayBuffer)

// Copy the value.  Note that arrayCopy uses the same buffer that
// 'address' points into.
let arrayCopy = array
retain(arrayCopy.buffer)

// Perform the mutation.
operate(address, 0)

// Release the "new" buffer.
release(newArrayBuffer)
```

This might be unexpected behavior, but the language team is willing to accept unexpected behavior for this code. What's non-negotiable is breaking memory safety.

Unfortunately, applying this fix naively reintroduces the problem of subobject clobbering: since a modification of one subobject immediately retains a buffer that's global to the entire value, an interleaved modification of a different subobject will see a non-unique buffer reference and therefore perform a structural copy. The modifications to the first subobject will therefore be silently lost.

Unlike the interleaving copy case, this is seen as unacceptable. Notably, it breaks swapping two array elements:

```
// Original:
swap(&array[i], &array[j])

// Expanded:

// Project array[i].
array.transitionToUniquelyReferenced()
let address_i = array.buffer.storage + i
let newArrayBuffer_i = array.buffer
```

```
retain(newArrayBuffer_i)

// Project array[j].  Note that this transition is guaranteed
// to have to do a structural copy.
array.transitionToUniquelyReferenced()
let address_j = array.buffer.storage + j
let newArrayBuffer_j = array.buffer
retain(newArrayBuffer_j)

// Perform the mutations.
swap(address_i, address_j)

// Balance out the retains.
release(newArrayBuffer_j)
release(newArrayBuffer_i)
```

get- and setForMutation

Some collections need finer-grained control over the entire mutation process. For instance, to support divide-and-conquer algorithms using slices, sliceable collections must "pin" and "unpin" their buffers while a slice is being mutated to grant permission for the slice to mutate the collection in-place while sharing ownership. This flexibility can be exposed by a pair of accessors that are called before and after a mutation. The "get" stage produces both the value to mutate and a state value (whose type must be declared) to forward to the "set" stage. A pinning accessor can then look something like this:

```
extension Array {
  subscript(range: Range<Int>) -> Slice<Element> {
    // `getForMutation` must declare its return value, a pair of both
    // the value to mutate and a state value that is passed to
    // `setForMutation`.
    getForMutation() -> (Slice<Element>, PinToken) {
      let slice = _makeSlice(range)
      let pinToken = _pin()
      return (slice, pinToken)
    }

    // `setForMutation` receives two arguments--the result of the
    // mutation to write back, and the state value returned by
    // `getForMutation`.
    setForMutation(slice, pinToken) {
      _unpin(pinToken)
      _writeSlice(slice, backToRange: range)
    }
  }
}
```

`getForMutation` and `setForMutation` must appear as a pair; neither one is valid on its own. A `get` and `set` accessor should also still be provided for simple read and write operations. When the compiler has visibility that storage is implemented in terms of `getForMutation` and `setForMutation`, it lowers a mutable projection using those accessors as follows:

```
// A mutation like this (assume `reverse` is a mutating method):
array[0...99].reverse()
// Decomposes to:
let index = 0...99
(var slice, let state) = array.`subscript.getForMutation`(index)
slice.reverse()
array.`subscript.setForMutation`(index, slice, state)
```

To support the conservative access pattern, a *materializeForSet* accessor can be generated from *getForMutation* and *setForMutation* in an obvious fashion: perform *getForMutation* and store the state result in its scratch space, and return a callback that loads the state and hands it off to *setForMutation*.

The beacon of hope for a user-friendly future: Inversion of control

Addressors and `{get,set}ForMutation` expose important optimizations to the standard library, but are undeniably fiddly and unsafe constructs to expose to users. A more natural model would be to recognize that a compound mutation is a composition of nested scopes, and express it in the language that way. A strawman model might look something like this:

```
var foo: T {
  get { return getValue() }
  set { setValue(newValue) }

  // Perform a full in-out mutation. The `next` continuation is of
  // type `(inout T) -> ()` and must be called exactly once
  // with the value to hand off to the nested mutation operation.
  mutate(next) {
    var value = getValue()
    next(&value)
    setValue(value)
  }
```

```
}
```

This presents a natural model for expressing the lifetime extension concerns of addressors, and the state maintenance necessary for pinning `getForMutation` accessors:

```
// An addressing mutator
mutate(next) {
  withUnsafePointer(&resource) {
    next(&$0.memory)
  }
}

// A pinning mutator
mutate(next) {
  var slice = makeSlice()
  let token = pin()
  next(&slice)
  unpin(token)
  writeBackSlice(slice)
}
```

For various semantic and implementation efficiency reasons, we don't want to literally implement every access as a nesting of closures like this. Doing so would allow for semantic surprises (a mutate() operation never invoking its continuation, or doing so multiple times would be disastrous), and would interfere with the ability for *inout* and *mutating* functions to throw or otherwise nonlocally exit. However, we could present this model using *inversion of control*, similar to Python generators or async-await. A *mutate* operation could *yield* the *inout* reference to its inner value, and the compiler could enforce that a *yield* occurs exactly once on every control flow path:

```
// An addressing, yielding mutator
mutate {
  withUnsafePointer(&resource) {
    yield &$0.memory
  }
}

// A pinning mutator
mutate {
  var slice = makeSlice()
  let token = pin()
  yield &slice
  unpin(token)
  writeBackSlice(slice)
}
```

This obviously requires more implementation infrastructure than we currently have, and raises language and library design issues (in particular, lifetime-extending combinators like `withUnsafePointer` would need either a `reyields` kind of decoration, or to become macros), but represents a promising path toward exposing the full power of the accessor model to users in an elegant way.

Acceptability

This whitepaper has mentioned several times that the language team is prepared to accept such-and-such behavior but not prepared to accept some other kind of behavior. Clearly, there is a policy at work. What is it?

General philosophy

For any given language problem, a perfect solution would be one which:

- guarantees that all operations complete without crashing or corrupting the program state,
- guarantees that all operations produce results according to consistent, reliable, and intuitive rules,
- does not limit or require complex interactions with the remainder of the language, and
- imposes no performance cost.

These goals are, however, not all simultaneously achievable, and different languages reach different balances. Swift's particular philosophy is as follows:

- The language should be as dynamically safe as possible. Straightforward uses of ordinary language features may cause dynamic failure, but the should never corrupt the program state. Any unsafe language or library features (other than simply calling into C code) should be explicitly labeled as unsafe.

  A dynamic failure should mean that the program reliably halts, ideally with a message clearly describing the source of the failure. In the future, the language may allow for emergency recovery from such failures.

- The language should sit on top of C, relying only on a relatively unobtrusive runtime. Accordingly, the language's interactions with C-based technologies should be efficient and obvious.

- The language should allow a static compiler to produce efficient code without dynamic instrumentation. Accordingly, static analysis should only be blocked by incomplete information when the code uses an obviously abstract language feature (such as calling a class method or an unknown function), and the language should provide tools to allow programmers to limit such

cases.

(Dynamic instrumentation can, of course, still help, but it shouldn't be required for excellent performance.)

General solutions

A language generally has six tools for dealing with code it considers undesirable. Some of this terminology is taken from existing standards, others not.

- The language may nonetheless take steps to ensure that the code executes with a reliable result. Such code is said to have *guaranteed behavior*.
- The language may report the code as erroneous before it executes. Such code is said to be *ill formed*.
- The language may reliably report the code as having performed an illegal operation when it executes. Such code is said to be *asserting* or *aborting*.
- The language may allow the code to produce an arbitrary-but-sound result. Such code is said to have *unspecified behavior* or to have produced an *unspecified value*.
- The language may allow the code to produce an unsound result which will result in another of these behaviors, but only if used. Such code is said to have produced a *trap value*.
- The language may declare the code to be completely outside of the guarantees of the language. Such code is said to have *undefined behavior*.

In keeping with its design philosophy, Swift has generally limited itself to the first four solutions, with two significant exceptions.

The first exception is that Swift provides several explicitly unsafe language and library features, such as `UnsafePointer<T>` and `unowned(unsafe)`. The use of these features is generally subject to undefined behavior rules.

The second exception is that Swift does not make any guarantees about programs in the presence of race conditions. It is extremely difficult to make even weak statements about the behavior of a program with a race condition without either:

- heavily restricting shared mutable state on a language level, which would require invasive changes to how the language interacts with C;
- forcing implicit synchronization when making any change to potentially shared memory, which would cripple performance and greatly complicate library implementation; or
- using a garbage collector to manage all accessible memory, which would impose a very large burden on almost all of Swift's language goals.

Therefore, Swift does surrender safety in the presence of races.

Acceptability conditions for storage accesses

Storage access involves a tension between four goals:

- Preserving all changes when making simultaneous modifications to distinct subobjects; in other words, avoiding subobject clobbering
- Performing a predictable and intuitive sequence of operations when modifying storage that's implemented with a getter and setter
- Avoiding unnecessary copies of a value during a modification, especially when this forces a structural copy of a COW value
- Avoiding memory safety holes when accessing storage that's been implemented with memory.

Reprojection is good at preserving changes, but it introduces extra copies, and it's less intuitive about how many times getters and setters will be called. There may be a place for it anyway, if we're willing to accept the extra conceptual complexity for computed storage, but it's not a reasonable primary basis for optimizing the performance of storage backed by memory.

Solutions permitting in-place modification are more efficient, but they do have the inherent disadvantage of having to vend the address of a value before arbitrary interleaving code. Even if the address remains valid, and the solution to that avoids subobject clobbering, there's an unavoidable issue that the write can be lost because the address became dissociated from the storage. For example, if your code passes `&array[i]` to a function, you might plausibly argue that changes to that argument should show up in the `i`th element of `array` even if you completely reassign `array`. Reprojection could make this work, but in-place solutions cannot efficiently do so. So, for any in-place solution to be acceptable, there does need to be some rule specifying when it's okay to "lose track" of a change.

Furthermore, the basic behavior of COW means that it's possible to copy an array with an element under modification and end up sharing the same buffer, so that the modification will be reflected in a value that was technically copied beforehand. For example:

```
var array = [1,2,3]
var oldArray : [Int] = []

// This function copies array before modifying it, but because that
// copy is of a value undergoing modification, the copy will use
// the same buffer and therefore observe updates to the element.
func foo(element: inout Int) {
  oldArray = array
  element = 4
}

// Therefore, oldArray[2] will be 4 after this call.
foo(&array[2])
```

Nor can this be fixed by temporarily moving the modified array aside, because that would prevent simultaneous modifications to different elements (and, in fact, likely cause them to assert). So the rule will also have to allow this.

However, both of these possibilities already come up in the design of both the library and the optimizer. The optimizer makes a number of assumptions about aliasing; for example, the general rule is that storage bound to an `inout` parameter cannot be accessed through other paths, and while the optimizer is not permitted to compromise memory safety, it is permitted to introduce exactly this kind of unexpected behavior where aliasing accesses may or may not the storage as a consistent entity.

Formal accesses

That rule leads to an interesting generalization. Every modification of storage occurs during a *formal access* (FA) to that storage. An FA is also associated with zero or more *designated storage names* (DSNs), which are `inout` arguments in particular execution records. An FA arises from an l-value expression, and its duration and DSN set depend on how the l-value is used:

- An l-value which is simply loaded from creates an instantaneous FA at the time of the load. The DSN set is empty.

  Example:

  ```
  foo(array)
  // instantaneous FA reading array
  ```

- An l-value which is assigned to with = creates an instantaneous FA at the time of the primitive assignment. The DSN set is empty.

  Example:

  ```
  array = []
  // instantaneous FA assigning array
  ```

  Note that the primitive assignment strictly follows the evaluation of both the l-value and r-value expressions of the assignment. For example, the following code:

  ```
  // object is a variable of class type
  object.array = object.array + [1,2,3]
  ```

  produces this sequence of formal accesses:

  ```
  // instantaneous FA reading object (in the left-hand side)
  // instantaneous FA reading object (in the right-hand side)
  // instantaneous FA reading object.array (in the right-hand side)
  // evaluation of [1,2,3]
  // evaluation of +
  // instantaneous FA assigning object.array
  ```

- An l-value which is passed as an `inout` argument to a call creates an FA beginning immediately before the call and ending immediately after the call. (This includes calls where an argument is implicitly passed `inout`, such as to mutating methods or user-defined assignment operators such as += or ++.) The DSN set contains the `inout` argument within the call.

  Example:

  ```
  func swap<T>(lhs: inout T, rhs: inout T) {}

  // object is a variable of class type
  swap(&leftObject.array, &rightObject.array)
  ```

  This results in the following sequence of formal accesses:

  ```
  // instantaneous FA reading leftObject
  // instantaneous FA reading rightObject
  // begin FA for inout argument leftObject.array (DSN={lhs})
  // begin FA for inout argument rightObject.array (DSN={rhs})
  // evaluation of swap
  // end FA for inout argument rightObject.array
  // end FA for inout argument leftObject.array
  ```

- An l-value which is used as the base of a member storage access begins an FA whose duration is the same as the duration of the FA for the subobject l-value. The DSN set is empty.

  Example:

  ```
  swap(&leftObject.array[i], &rightObject.array[j])
  ```

  This results in the following sequence of formal accesses:

  ```
  // instantaneous FA reading leftObject
  // instantaneous FA reading i
  // instantaneous FA reading rightObject
  // instantaneous FA reading j
  // begin FA for subobject base leftObject.array (DSN={})
  // begin FA for inout argument leftObject.array[i] (DSN={lhs})
  // begin FA for subobject base rightObject.array (DSN={})
  ```

```
// begin FA for inout argument rightObject.array[j] (DSN={rhs})
// evaluation of swap
// end FA for subobject base rightObject.array[j]
// end FA for inout argument rightObject.array
// end FA for subobject base leftObject.array[i]
// end FA for inout argument leftObject.array
```

- An l-value which is the base of an ! operator begins an FA whose duration is the same the duration of the FA for the resulting l-value. The DSN set is empty.

  Example:

  ```
  // left is a variable of type T
  // right is a variable of type T?
  swap(&left, &right!)
  ```

  This results in the following sequence of formal accesses:

  ```
  // begin FA for inout argument left (DSN={lhs})
  // begin FA for ! operand right (DSN={})
  // begin FA for inout argument right! (DSN={rhs})
  // evaluation of swap
  // end FA for inout argument right!
  // end FA for ! operand right
  // end FA for inout argument left
  ```

- An l-value which is the base of a ? operator begins an FA whose duration begins during the formal evaluation of the l-value and ends either immediately (if the operand was nil) or at the end of the duration of the FA for the resulting l-value. In either case, the DSN set is empty.

  Example:

  ```
  // left is a variable of optional struct type
  // right is a variable of type Int
  left?.member += right
  ```

  This results in the following sequence of formal accesses, assuming that `left` contains a value:

  ```
  // begin FA for ? operand left (DSN={})
  // instantaneous FA reading right (DSN={})
  // begin FA for inout argument left?.member (DSN={lhs})
  // evaluation of +=
  // end FA for inout argument left?.member
  // end FA for ? operand left
  ```

The FAs for all `inout` arguments to a call begin simultaneously at a point strictly following the evaluation of all the argument expressions. For example, in the call `foo(&array, array)`, the evaluation of the second argument produces a defined value, because the FA for the first argument does not begin until after all the arguments are formally evaluated. No code should actually be emitted during the formal evaluation of `&array`, but for an expression like `someClassReference.someArray[i]`, the class r-value and index expressions would be fully evaluated at that time, and then the l-value would be kept abstract until the FA begins. Note that this requires changes in SILGen's current code generation patterns.

The FA rule for the chaining operator `?` is an exception to the otherwise-simple intuition that formal accesses begin immediately before the modification begins. This is necessary because the evaluation rules for `?` may cause arbitrary computation to be short-circuited, and therefore the operand must be accessed during the formal evaluation of the l-value. There were three options here:

- Abandon short-circuiting for assignments to optional l-values. This is a very high price; short-circuiting fits into user intuitions about the behavior of the chaining operator, and it can actually be quite awkward to replicate with explicit accesses.
- Short-circuit using an instantaneous formal access, then start a separate formal access before the actual modification. In other words, evaluation of `X += Y` would proceed by first determining whether `X` exists (capturing the results of any r-value components), discarding any projection information derived from that, evaluating `Y`, reprojecting `X` again (using the saved r-value components and checking again for whether the l-value exists), and finally calling the `+=` operator.

  If `X` involves any sort of computed storage, the steps required to evaluate this might be... counter-intuitive.
- Allow the formal access to begin during the formal evaluation of the l-value. This means that code like the following will have unspecified behavior:

  ```
  array[i]?.member = deriveNewValueFrom(array)
  ```

In the end, I've gone with the third option. The intuitive explanation is that `array` has to be accessed early in order to continue the evaluation of the l-value. I think that's comprehensible to users, even if it's not immediately obvious.

Disjoint and non-disjoint formal accesses

I'm almost ready to state the core rule about formal accesses, but first I need to build up a few more definitions.

An *abstract storage location* (ASL) is:

- a global variable declaration;
- an `inout` parameter declaration, along with a reference to a specific execution record for that function;
- a local variable declaration, along with a reference to a specific execution record for that declaration statement;
- a static/class property declaration, along with a type having that property;
- a struct/enum instance property declaration, along with an ASL for the base;
- a struct/enum subscript declaration, along with a concrete index value and an ASL for the base;
- a class instance property declaration, along with an instance of that class; or
- a class instance subscript declaration, along with a concrete index value and an instance of that class.

Two abstract storage locations may be said to *overlap*. Overlap corresponds to the imprecise intuition that a modification of one location directly alters the value of another location. Overlap is an "open" property of the language: every new declaration may introduce its own overlap behavior. However, the language and library make certain assertions about the overlap of some locations:

- An `inout` parameter declaration overlaps exactly the set of ASLs overlapped by the ASL which was passed as an argument.
- If two ASLs are both implemented with memory, then they overlap only if they have the same kind in the above list and the corresponding data match:
    - execution records must represent the same execution
    - types must be the same
    - class instances must be the same
    - ASLs must overlap
- For the purposes of the above rule, the subscript of a standard library array type is implemented with memory, and the two indexes match if they have the same integer value.
- For the purposes of the above rule, the subscript of a standard library dictionary type is implemented with memory, and the two indexes match if they compare equal with `==`.

Because this definition is open, it is impossible to completely statically or dynamically decided it. However, it would still be possible to write a dynamic analysis which decided it for common location kinds. Such a tool would be useful as part of, say, an ASan-like dynamic tool to diagnose violations of the unspecified-behavior rule below.

The overlap rule is vague about computed storage partly because computed storage can have non-obvious aliasing behavior and partly because the subobject clobbering caused by the full-value accesses required by computed storage can introduce unexpected results that can be reasonably glossed as unspecified values.

This notion of abstract storage location overlap can be applied to formal accesses as well. Two FAs `x` and `y` are said to be *disjoint* if:

- they refer to non-overlapping abstract storage locations or
- they are the base FAs of two disjoint member storage accesses `x.a` and `y.b`.

Given these definitions, the core unspecified-behavior rule is:

> If two non-disjoint FAs have intersecting durations, and neither FA is derived from a DSN for the other, then the program has unspecified behavior in the following way: if the second FA is a load, it yields an unspecified value; otherwise, both FAs store an unspecified value in the storage.

Note that you cannot have two loads with intersecting durations, because the FAs for loads are instantaneous.

Non-overlapping subobject accesses make the base accesses disjoint because otherwise code like `swap(&a[0], &a[1])` would have unspecified behavior, because the two base FAs are to clearly overlapping locations and have intersecting durations.

Note that the optimizer's aliasing rule falls out from this rule. If storage has been bound as an `inout` argument, accesses to it through any path not derived from the `inout` argument will start a new FA for overlapping storage, the duration of which will necessarily intersect duration with that of the FA through which the `inout` argument was bound, causing unspecified behavior. If the `inout` argument is forwarded to another call, that will start a new FA which is validly based on a DSN of the first; but an attempt to modify the storage through the first `inout` argument while the second call is active will create a third FA not based on the DSN from the second `inout` call, causing a conflict there. Therefore a function may assume that it can see all accesses to the storage bound to an `inout` argument.

If you didn't catch all that...

That may have been a somewhat intense description, so here's a simple summary of the rule being proposed.

If storage is passed to an `inout` argument, then any other simultaneous attempt to read or write to that storage, including to the storage containing it, will have unspecified behavior. Reads from it may see partially-updated values, or even values which will change as modifications are made to the original storage; and writes may be clobbered or simply disappear.

But this only applies during the call with the `inout` argument: the evaluation of other arguments to the call will not be interfered with, and as soon as the call ends, all these modifications will resolve back to a quiescent state.

And this unspecified behavior has limits. The storage may end up with an unexpected value, with only a subset of the writes made to it, and copies from it may unexpectedly reflect modifications made after they were copied. However, the program will otherwise

remain in a consistent and uncorrupted state. This means that execution will be able to continue apace as long as these unexpected values don't trip up some higher-level invariant.

Tracking formal accesses

Okay, now that I've analyzed this to death, it's time to make a concrete proposal about the implementation.

As discussed above, the safety hole with addressors can be fixed by always retaining the buffer which keeps the address valid. Assuming that other uses of the buffer follow the general copy-on-write pattern, this retain will prevent structural changes to the buffer while the address is in use.

But, as I also discussed above, this introduces two problems:

Copies during modification

Copying a COW aggregate value always shares the same buffer that was stored there at the time of the copy; there is no uniqueness check done as part of the copy. Changes to subobjects will then be instantly reflected in the "copy" as they are made to the original. The structure of the copy will stay the same, but the values of its subobjects will appear to spontaneously change.

I want to say that this behavior is acceptable according to the formal-access rule I laid out above. How does that reasoning work?

First, I need to establish what kind of behavior is at work here. It clearly isn't guaranteed behavior: copies of COW values are normally expected to be independent. The code wasn't rejected by the compiler, nor did it dynamically assert; it simply seems to misbehave. But there are limits to the misbehavior:

- By general COW rules, there's no way to change the structure of an existing buffer unless the retain count is 1. For the purposes of this analysis, that means that, as long as the retain count is above 1, there's no way to invalidate the address returned by the addressor.
- The buffer will be retained for as long as the returned address is being modified. This retain is independent of any storage which might hold the aggregate value (and thus also retain the buffer).
- Because of this retain, the only way for the retain count to drop to 1 is for no storage to continue to refer to the buffer.
- But if no storage refers to the buffer, there is no way to initiate an operation which would change the buffer structure.

Thus the address will remain valid, and there's no danger of memory corruption. The only thing is that the program no longer makes useful guarantees about the value of the copied aggregate. In other words, the copy yielded an unspecified value.

The formal-access rule allows loads from storage to yield an unspecified value if there's another formal access to that storage in play and the load is (1) not from an l-value derived from a name in the other FA's DSN set and (2) not from a non-overlapping subobject. Are these conditions true?

Recall that an addressor is invoked for an l-value of the form:

```
base.pointee
```

or:

```
base[i]
```

Both cases involve a formal access to the storage `base` as the base of a subobject formal access. This kind of formal access always has an empty DSN set, regardless of how the subobject is used. A COW mutable addressor will always ensure that the buffer is uniquely referenced before returning, so the only way that a value containing that buffer can be copied is if the load is a non-subobject access to `base`. Therefore, there are two simultaneous formal accesses to the same storage, and the load is not from an l-value derived from the modification's DSN set (which is empty), nor is it for a non-overlapping subobject. So the formal-access rule applies, and an unspecified value is an acceptable result.

The implementation requirement here, then, is simply that the addressor must be called, and the buffer retained, within the duration of the formal access. In other words, the addressor must only be called immediately prior to the call, rather than at the time of the formal evaluation of the l-value expression.

What would happen if there *were* a simultaneous load from a non-overlapping subobject? Accessing the subobject might cause a brief copy of `base`, but only for the duration of copying the subobject. If the subobject does not overlap the subobject which was projected out for the addressor, then this is harmless, because the addressor will not allow modifications to those subobjects; there might be other simultaneous formal accesses which do conflict, but these two do not. If the subobject does overlap, then a recursive analysis must be applied; but note that the exception to the formal-access rule will only apply if non-overlapping subobjects were projected out from *both* formal accesses. Otherwise, it will be acceptable for the access to the overlapping subobject to yield an unspecified value.

Avoiding subobject clobbering during parallel modification

The other problem is that the retain will prevent simultaneous changes to the same buffer. The second change will cause a structural copy, and the first address will end up modifying a buffer which is no longer referenced: in other words, the program will observe subobject clobbering. A similar analysis to the one from the last section suggests that this can be described as unspecified behavior.

Unfortunately, this unspecified behavior is unwanted: it violates the guarantees of the formal-access rule as I laid it out above, because it occurs even if you have formal accesses to two non-overlapping subobjects. So something does need to be done here.

One simple answer is to dynamically track whether a COW buffer is currently undergoing a non-structural mutation. I'll call this *NSM tracking*, and I'll call buffers which are undergoing non-structural mutations *NSM-active*.

The general rules of COW say that mutating operations must ensure that their buffer is uniquely referenced before performing the modification. NSM tracking works by having non-structural mutations perform a weaker check: the buffer must be either uniquely referenced or be NSM-active. If the non-structural mutation allows arbitrary code to run between the start of the mutation and the end --- as an addressor does --- it must both retain the buffer and flag it as NSM-active for the entire duration.

Because the retain still occurs, and because any *structural* changes to the buffer that might invalidate the addresses of subobjects are still blocked by that retain, all of the earlier analysis about the memory safety of simultaneous accesses still applies. The only change is that simultaneous non-structural modifications, as would be created by simultaneous formal accesses to subobjects, will now be able to occur on a single buffer.

A set of simultaneous formal accesses on a single thread follows a natural stack protocol, or can be made to do so with straightforward SILGen and SIL optimizer consideration. Therefore, the runtime can track whether a buffer is NSM-active on a thread using a single bit, which nested modifications can be told not to clear. Call this the *NSM bit*. Ignoring multithreading considerations for a moment, since the NSM bit is only ever set at the same as a retain and only ever cleared at the same time as a release, it makes sense to pack this into the strong reference count. There is no need to support this operation on non-Swift objects. The runtime should provide three new functions:

- A function to test whether an object is either uniquely referenced or NSM-active. Call this `swift_isUniquelyReferencedForNSM`.
- A function to perform the above test and, if the test passes and the NSM bit is not set, atomically retain the object and set the NSM bit. It should return both the result of the test and an object to later set as NSM-inactive. That object will be nil if the test failed or the NSM bit was already set. Call this `swift_tryRetainForNSM`.
- A function to atomically clear the NSM bit and release the object. Call this `swift_releaseForNSM`.

These operations should also be reflected in SIL.

Concurrent modifications and the non-structural modification bit

What about concurrency? Two concurrent non-structural modifications could race to set the NSM bit, and then the winning thread could clear it before the other thread's modification is complete. This could cause memory-unsafe behavior, since the losing thread would be modifying the object through an address while not retaining the value.

The major question here is whether this is a significant objection. It's accepted that race conditions have undefined behavior. Is such code inherently racy?

The answer appears to be "no", and that it is possible to write code which concurrently writes to existing non-overlapping elements of a COW aggregate without causing races; but that such code is extremely fraught, and moreover it is extremely fraught regardless of whether NSM-activeness is tracked with a single bit or a wider count. Consider:

- If the shared aggregate value is ever non-uniquely referenced, two threads concurrently modifying it will race to unique the array. This unavoidably has undefined behavior, because uniquing the array requires the previous value to eventually be released, and a race may cause an over-release.
- Assume that it's possible to guarantee that the aggregate value's buffer is uniquely referenced before any threads concurrently access it. Now, all of the threads are performing different concurrent accesses.
    - If any of the accesses is a structural modification, there will be a race to re-unique the buffer.
    - If all of the accesses are non-structural modifications, then there will be no races as long as the retain-and-set and release-and-clear operations are atomic: when starting any particular operation, the buffer will always either be uniquely referenced or have the bit set.
    - If any of the accesses is a read, and that read does not occur during a non-structural modification, then the buffer may briefly become non-uniquely referenced and there will be a race from concurrent modifications to re-unique it.
    - If any of the accesses is a read, and that read occurs during a non-structural modification, and the optimizer does not re-order the read's retain/release around the retainForNSM/releaseForNSM operations, then it matters how NSM-activeness is tracked.

      If there is complete tracking (i.e. a count, not just a single bit), the retain for the read will only occur while the buffer is flagged as NSM-active, and so it will have no effect.

      If there is incomplete tracking (i.e. just a single NSM bit), then there is a potential for undefined behavior. Suppose two threads race to set the NSM bit. The loser then initiates a read and retains the buffer. Before the loser releases the buffer, the winner clears the NSM bit. Now another thread might see that the buffer is non-uniquely referenced and not NSM-active, and so it will attempt to unique the buffer.

      It is probably unreasonable to require the optimizer to never reorder ordinary retains and releases past retainForNSM and releaseForNSM operations.

More importantly, the use case here (many threads concurrently accessing different elements of a shared data structure) just inherently doesn't really work well with a COW data structure. Even if the library were able to make enough guarantees to ensure that, with the right pattern of accesses, there would never be a structural copy of the aggregate, it would still be extremely inefficient,

because all of the threads would be competing for atomic access to the strong reference count.

In short, I think it's reasonable for the library to say that programs which want to do this should always use a type with reference semantics. Therefore, it's reasonable to ignore concurrent accesses when deciding how to best track whether an aggregate is undergoing non-structural modification. This removes the only objection I can see to tracking this with a single NSM bit.

Code generation patterns

The signatures and access patterns for addressors will need to change in order to ensure memory-safety.

`mutableAddress` currently returns an `UnsafeMutablePointer`; it will need to return `(Builtin.NativeObject?, UnsafeMutablePointer)`. The owner pointer must be a native object; we cannot efficiently support either uniqueness checking or the NSM bit on non-Swift objects. SILGen will mark that the address depends on the owner reference and push a cleanup to `releaseForNSM` it.

`address` currently returns an `UnsafePointer`; it will need to return `(Builtin.NativeObject?, UnsafePointer)`. I do not currently see a reason to allow non-Swift owners, but the model doesn't depend on that. SILGen will mark that the address depends on the owner reference and push a cleanup to `release` it.

In order to support ultimately calling an addressor in the conservative access path, `materializeForSet` must also return an owner reference. Since `materializeForSet` calls `mutableAddress` in this case, SILGen will follow that pattern for calls. SILGen will also assume that the need to perform a `releaseForNSM` is exclusive with the need to call the setter.

Mutating operations on COW types will now have two different paths for making a buffer mutable and unique: one for structural mutations and another for non-structural mutations. I expect that this will require separate semantics annotations, and the optimizer will have to recognize both.

`releaseForNSM` operations will not be reorderable unless the optimizer can prove that the objects are distinct.

Summary of proposal and plan

Let me summarize what I'm proposing:

- Swift's core approach to optimizing accesses should be based around providing direct access to memory, either statically or dynamically. In other words, Swift should adopt addressors on core data structures as much as possible.
- Swift should fix the current memory hole with addressors by retaining for the duration of the access and, for modifications, flagging the buffer as NSM-active. The implementation plan follows:
    - The runtime implements the NSM-bit and its entrypoints.
    - SIL provides operations for manipulating and querying the NSM bit. IRGen implements these operations using the runtime functions. Builtins are exposed.
    - The standard library changes data structures to do different uniquing for structural and non-structural modifications. This patch is not yet committed.
    - The optimizer reacts to the above. When both are settled, they can be committed.
    - SILGen changes the emission patterns for l-values so that addresses and writebacks are live only during the formal access.
    - Sema changes the signature of `address`, `mutableAddress`, and `materializeForSet` to return an optional owner reference. Sema changes `materializeForSet` synthesis to return the owner correctly. SILGen implements the desired code patterns.

      The standard library changes its addressor implementations to continue to compile, but for staging purposes, it only uses nil owners.
    - The standard library changes addressor implementations to use meaningful owners. This patch is not yet committed.
    - The optimizer reacts to the above. When both are settled, they can be committed.