**orphan:**

**System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\swift-main\docs\proposals\(swift-main)(docs)(proposals)InoutCOWOptimization.rst`, line 3)**

Unknown directive type "highlight".

```
.. highlight:: sil
```

# Copy-On-Write Optimization of `inout` Values

**Authors:** Dave Abrahams, Joe Groff

**Summary:** Our writeback model interacts with Copy-On-Write (COW) to cause some surprising inefficiencies, such as O(N) performance for `x[0][0] = 1`. We propose a modified COW optimization that recovers O(1) performance for these cases and supports the efficient use of slices in algorithm implementation.

## Whence the Problem?

The problem is caused as follows:

- COW depends on the programmer being able to mediate all writes (so she can copy if necessary)
- Writes to container elements and slices are mediated through subscript setters, so in

  ```
  x[0].mutate()
  ```

  we "`subscript get`" `x[0]` into a temporary, mutate the temporary, and "`subscript set`" it back into `x[0]`.
- When the element itself is a COW type, that temporary implies a retain count of at least 2 on the element's buffer.
- Therefore, mutating such an element causes an expensive copy, *even when the element's buffer isn't otherwise shared.*

Naturally, this problem generalizes to any COW value backed by a getter/setter pair, such as a computed or resilient `String` property:

```
anObject.title.append('.') // O(N)
```

## Interaction With Slices

Consider the classic divide-and-conquer algorithm QuickSort, which could be written as follows:

```
protocol Sliceable {
  ...
  @mutating
  func quickSort(_ compare: (StreamType.Element, StreamType.Element) -> Bool) {
    let (start, end) = (startIndex, endIndex)
    if start != end && start.succ() != end {
      let pivot = self[start]
      let mid = partition(by: {!compare($0, pivot)})
      self[start...mid].quickSort(compare)
      self[mid...end].quickSort(compare)
    }
  }
}
```

The implicit `inout` on the target of the recursive `quickSort` calls currently forces two allocations and O(N) copies in each layer of the QuickSort implementation. Note that this problem applies to simple containers such as `Int[]`, not just containers of COW elements.

Without solving this problem, mutating algorithms must operate on `MutableCollection`s and pairs of their `Index` types, and we must hope the ARC optimizer is able to eliminate the additional reference at the top-level call. However, that does nothing for the cases mentioned in the previous section.

## Our Solution

We need to prevent lvalues created in an `inout` context from forcing a copy-on-write. To accomplish that:

- In the class instance header, we reserve a bit `INOUT`.
- When a unique reference to a COW buffer `b` is copied into an `inout` lvalue, we save the value of the `b.INOUT` bit and set it.
- When a reference to `b` is taken that is not part of an `inout` value, `b.INOUT` is cleared.
- When `b` is written-back into `r`, `b.INOUT` is restored to the saved value.
- A COW buffer can be modified in-place when it is uniquely referenced *or* when `INOUT` is set.

We believe this can be done with little user-facing change; the author of a COW type would add an attribute to the property that stores the buffer, and we would use a slightly different check for in-place writability.

#### Other Considered Solutions

Move optimization seemed like a potential solution when we first considered this problem--given that it is already unspecified to reference a property while an active `inout` reference can modify it, it seems natural to move ownership of the value to the `inout` when entering writeback and move it back to the original value when exiting writeback. We do not think it is viable for the following reasons:

- In general, relying on optimizations to provide performance semantics is brittle.
- Move optimization would not be memory safe if either the original value or `inout` slice were modified to give up ownership of the original backing store. Although observing a value while it has inout aliases is unspecified, it should remain memory-safe to do so. This should remain memory safe, albeit unspecified:

  ```
  var arr = [1,2,3]
  func mutate(_ x: inout Int[]) -> Int[] {
    x = [3...4]
    return arr[0...2]
  }
  mutate(&arr[0...2])
  ```

  Inout slices thus require strong ownership of the backing store independent of the original object, which must also keep strong ownership of the backing store.
- Move optimization requires unique referencing and would fail when there are multiple concurrent, non-overlapping `inout` slices. `swap(&x.a, &x.b)` should be well-defined if `x.a` and `x.b` do not access overlapping state, and so should be `swap(&x[0...50], &x[50...100])`. More generally, we would like to use inout slicing to implement divide-and-conquer parallel algorithms, as in:

  ```
  async { mutate(&arr[0...50]) }
  async { mutate(&arr[50...100]) }
  ```

## Language Changes

#### Builtin.isUniquelyReferenced

A mechanism needs to be exposed to library writers to allow them to check whether a buffer is uniquely referenced. This check requires primitive access to the layout of the heap object, and can also potentially be reasoned about by optimizations, so it makes sense to expose it as a `Builtin` which lowers to a SIL `is_uniquely_referenced` instruction.

#### The `@cow` attribute

A type may declare a stored property as being `@cow`:

```
class ArrayBuffer { /* ... */ }

struct Array {
  @cow var buffer : ArrayBuffer
}
```

The property must meet the following criteria:

- It must be a stored property.
- It must be of a pure Swift class type. (More specifically, at the implementation level, it must have a Swift refcount.)
- It must be mutable. A `@cow val` property would not be useful.

Values with `@cow` properties have special implicit behavior when they are used in `inout` contexts, described below.

## Implementation of @cow properties

#### inout SIL operations

To maintain the `INOUT` bit of a class instance, we need new SIL operations that update the `INOUT` bit. Because the state of the bit needs to be saved and restored through every writeback scope, we can have:

```
%former = inout_retain %b : $ClassType
```

increase the retain count, save the current value of `INOUT`, set `INOUT`, and produce the `%former` value as its `Int1` result. To release, we have:

```
inout_release %b : $ClassType, %former : $Builtin.Int1
```

both reduce the retain count and change the value of `INOUT` back to the value saved in `%former`. Furthermore:

```
strong_retain %b : $ClassType
```

must always clear the `INOUT` bit.

To work with opaque types, `copy_addr` must also be able to perform an `inout` initialization of a writeback buffer as well as reassignment to an original value. This can be an additional attribute on the source, mutually exclusive with `[take]`:

```
copy_addr [inout] %a to [initialization] %b
```

This implies that value witness tables will need witnesses for inout-initialization and inout-reassignment.

### Copying of @cow properties for writeback

When a value is copied into a writeback buffer, its `@cow` properties must be retained for the new value using `inout_retain` instead of `strong_retain` (or `copy_addr [inout] [initialization]` instead of plain `copy_addr [initialization]`). When the value is written back, the property values should be `inout_release`d, or the value should be written back using `copy_addr [inout]` reassignment.