Ownership

Introduction

Adding "ownership" to Swift is a major feature with many benefits for programmers. This document is both a "manifesto" and a "meta-proposal" for ownership: it lays out the basic goals of the work, describes a general approach for achieving those goals, and proposes a number of specific changes and features, each of which will need to be separately discussed in a smaller and more targeted proposal. This document is intended to provide a framework for understanding the contributions of each of those changes.

Problem statement

The widespread use of copy-on-write value types in Swift has generally been a success. It does, however, come with some drawbacks:

- Reference counting and uniqueness testing do impose some overhead.
- Reference counting provides deterministic performance in most cases, but that performance can still be complex to analyze and predict.
- The ability to copy a value at any time and thus "escape" it forces the underlying buffers to generally be heap-allocated. Stack allocation is far more efficient but does require some ability to prevent, or at least recognize, attempts to escape the value.

Certain kinds of low-level programming require stricter performance guarantees. Often these guarantees are less about absolute performance than *predictable* performance. For example, keeping up with an audio stream is not a taxing job for a modern processor, even with significant per-sample overheads, but any sort of unexpected hiccup is immediately noticeable by users.

Another common programming task is to optimize existing code when something about it falls short of a performance target. Often this means finding "hot spots" in execution time or memory use and trying to fix them in some way. When those hot spots are due to implicit copies, Swift's current tools for fixing the problem are relatively poor; for example, a programmer can fall back on using unsafe pointers, but this loses a lot of the safety benefits and expressivity advantages of the library collection types.

We believe that these problems can be addressed with an opt-in set of features that we collectively call *ownership*.

What is ownership?

Ownership is the responsibility of some piece of code to eventually cause a value to be destroyed. An *ownership system* is a set of rules or conventions for managing and transferring ownership.

Any language with a concept of destruction has a concept of ownership. In some languages, like C and non-ARC Objective-C, ownership is managed explicitly by programmers. In other languages, like C++ (in part), ownership is managed by the language. Even languages with implicit memory management still have libraries with concepts of ownership, because there are other program resources besides memory, and it is important to understand what code has the responsibility to release those resources.

Swift already has an ownership system, but it's "under the covers": it's an implementation detail that programmers have little ability to influence. What we are proposing here is easy to summarize:

- We should add a core rule to the ownership system, called the Law of Exclusivity, which requires the implementation to prevent variables from being simultaneously accessed in conflicting ways. (For example,

being passed `inout` to two different functions.) This will not be an opt-in change, but we believe that most programs will not be adversely affected.

- We should add features to give programmers more control over the ownership system, chiefly by allowing the propagation of "shared" values. This will be an opt-in change; it will largely consist of annotations and language features which programmers can simply not use.
- We should add features to allow programmers to express types with unique ownership, which is to say, types that cannot be implicitly copied. This will be an opt-in feature intended for experienced programmers who desire this level of control; we do not intend for ordinary Swift programming to require working with such types.

These three tentpoles together have the effect of raising the ownership system from an implementation detail to a more visible aspect of the language. They are also somewhat inseparable, for reasons we'll explain, although of course they can be prioritized differently. For these reasons, we will talk about them as a cohesive feature called "ownership".

A bit more detail

The basic problem with Swift's current ownership system is copies, and all three tentpoles of ownership are about avoiding copies.

A value may be used in many different places in a program. The implementation has to ensure that some copy of the value survives and is usable at each of these places. As long as a type is copyable, it's always possible to satisfy that by making more copies of the value. However, most uses don't actually require ownership of their own copy. Some do: a variable that didn't own its current value would only be able to store values that were known to be owned by something else, which isn't very useful in general. But a simple thing like reading a value out of a class instance only requires that the instance still be valid, not that the code doing the read actually own a reference to it itself. Sometimes the difference is obvious, but often it's impossible to know. For example, the compiler generally doesn't know how an arbitrary function will use its arguments; it just falls back on a default rule for whether to pass ownership of the value. When that default rule is wrong, the program will end up making extra copies at runtime. So one simple thing we can do is to allow programs to be more explicit at certain points about whether they need ownership or not.

That closely dovetails with the desire to support non-copyable types. Most resources require a unique point of destruction: a memory allocation can only be freed once, a file can only be closed once, a lock can only be released once, and so on. An owning reference to such a resource is therefore naturally unique and thus non-copyable. Of course, we can artificially allow ownership to be shared by, say, adding a reference count and only destroying the resource when the count reaches zero. But this can substantially increase the overhead of working with the resource; and worse, it introduces problems with concurrency and re-entrancy. If ownership is unique, and the language can enforce that certain operations on a resource can only be performed by the code that owns the resource, then by construction only one piece of code can perform those operations at a time. As soon as ownership is shareable, that property disappears. So it is interesting for the language to directly support non-copyable types because they allow the expression of optimally-efficient abstractions over resources. However, working with such types requires all of the abstraction points like function arguments to be correctly annotated about whether they transfer ownership, because the compiler can no longer just make things work behind the scenes by adding copies.

Solving either of these problems well will require us to also solve the problem of non-exclusive access to variables. Swift today allows nested accesses to the same variable; for example, a single variable can be passed as two different `inout` arguments, or a method can be passed a callback that somehow accesses the same variable that the method was called on. Doing this is mostly discouraged, but it's not forbidden, and both the compiler and the standard library have to bend over backwards to ensure that the program won't misbehave too badly if it happens.

For example, `Array` has to retain its buffer during an in-place element modification; otherwise, if that modification somehow reassigned the array variable, the buffer would be freed while the element was still being changed. Similarly, the compiler generally finds it difficult to prove that values in memory are the same at different points in a function, because it has to assume that any opaque function call might rewrite all memory; as a result, it often has to insert copies or preserve redundant loads out of paranoia. Worse, non-exclusive access greatly limits the usefulness of explicit annotations. For example, a "shared" argument is only useful if it's really guaranteed to stay valid for the entire call, but the only way to reliably satisfy that for the current value of a variable that can be re-entrantly modified is to make a copy and pass that instead. It also makes certain important patterns impossible, like stealing the current value of a variable in order to build something new; this is unsound if the variable can be accessed by other code in the middle. The only solution to this is to establish a rule that prevents multiple contexts from accessing the same variable at the same time. This is what we propose to do with the Law of Exclusivity.

All three of these goals are closely linked and mutually reinforcing. The Law of Exclusivity allows explicit annotations to actually optimize code by default and enables mandatory idioms for non-copyable types. Explicit annotations create more optimization opportunities under the Law and enable non-copyable types to function. Non-copyable types validate that annotations are optimal even for copyable types and create more situations where the Law can be satisfied statically.

Criteria for success

As discussed above, it is the core team's expectation that ownership can be delivered as an opt-in enhancement to Swift. Programmers should be able to largely ignore ownership and not suffer for it. If this expectation proves to not be satisfiable, we will reject ownership rather than imposing substantial burdens on regular programs.

The Law of Exclusivity will impose some new static and dynamic restrictions. It is our belief that these restrictions will only affect a small amount of code, and only code that does things that we already document as producing unspecified results. These restrictions, when enforced dynamically, will also hurt performance. It is our hope that this will be "paid for" by the improved optimization potential. We will also provide tools for programmers to eliminate these safety checks where necessary. We will discuss these restrictions in greater detail later in this document.

Core definitions

Values

Any discussion of ownership systems is bound to be at a lower level of abstraction. We will be talking a lot about implementation topics. In this context, when we say "value", we mean a specific instance of a semantic, user-language value.

For example, consider the following Swift code:

```
var x = [1,2,3]
var y = x
```

People would often say that `x` and `y` have the same value at this point. Let's call this a *semantic value*. But at the level of the implementation, because the variables `x` and `y` can be independently modified, the value in `y` must be a copy of the value in `x`. Let's call this a *value instance*. A value instance can be moved around in memory and remain the same value instance, but a copy always yields a new value instance. For the remainder of this document, when we use "value" without any qualification, we mean it in this low-level sense of value instance.

What it means to copy or destroy a value instance depends on the type:

- Some types do not require extra work besides copying their byte-representation; we call these *trivial*. For example, `Int` and `Float` are trivial types, as are ordinary `struct`s and `enum`s containing only such

values. Most of what we have to say about ownership in this document doesn't apply to the values of such types. However, the Law of Exclusivity will still apply to them.

- For reference types, the value instance is a reference to an object. Copying the value instance means making a new reference, which increases the reference count. Destroying the value instance means destroying a reference, which decreases the reference count. Decreasing the reference count can, of course, drop it to zero and thus destroy the object, but it's important to remember that all this talk about copying and destroying values means manipulating reference counts, not copying the object or (necessarily) destroying it.

- For copy-on-write types, the value instance includes a reference to a buffer, which then works basically like a reference type. Again, it is important to remember that copying the value doesn't mean copying the contents of the buffer into a new buffer.

There are similar rules for every kind of type.

Memory

In general, a value can be owned in one of two ways: it can be "in flight", a temporary value owned by a specific execution context which computed the value as an operand, or it can be "at rest", stored in some sort of memory.

We don't need to focus much on temporary values because their ownership rules are straightforward. Temporary values are created as the result of some expression; that expression is used in some specific place; the value is needed in that place and not thereafter; so the implementation should clearly take all possible steps to forward the value directly to that place instead of forcing it to be copied. Users already expect all of this to happen, and there's nothing really to improve here.

Therefore, most of our discussion of ownership will center around values stored in memory. There are five closely related concepts in Swift's treatment of memory.

A *storage declaration* is the language-syntax concept of a declaration that can be treated in the language like memory. Currently, these are always introduced with `let`, `var`, and `subscript`. A storage declaration has a type. It also has an implementation which defines what it means to read or write the storage. The default implementation of a `var` or `let` just creates a new variable to store the value, but storage declarations can also be computed, and so there needn't be any variables at all behind one.

A *storage reference expression* is the syntax concept of an expression that refers to storage. This is similar to the concept from other languages of an "l-value", except that it isn't necessarily usable on the left side of an assignment because the storage doesn't have to be mutable.

A *storage reference* is the language-semantics concept of a fully filled-in reference to a specific storage declaration. In other words, it is the result of evaluating a storage reference expression in the abstract, without actually accessing the storage. If the storage is a member, this includes a value or storage reference for the base. If the storage is a subscript, this includes a value for the index. For example, a storage reference expression like `widgets[i].weight` might abstractly evaluate to this storage reference:

- the storage for the property `var weight: Double` of
- the storage for the subscript `subscript(index: Int)` at index value `19: Int` of
- the storage for the local variable `var widgets: [Widget]`

A *variable* is the semantics concept of a unique place in memory that stores a value. It's not necessarily mutable, at least as we're using it in this document. Variables are usually created for storage declarations, but they can also be created dynamically in raw memory, e.g. using `UnsafeRawPointer`. A variable always has a specific type. It also

has a *lifetime*, i.e. a point in the language semantics where it comes into existence and a point (or several) where it is destroyed.

A *memory location* is a contiguous range of addressable memory. In Swift, this is mostly an implementation concept. Swift does not guarantee that any particular variable will have a consistent memory location throughout its lifetime, or in fact be stored in a memory location at all. But a variable can sometimes be temporarily forced to exist at a specific, consistent location: e.g. it can be passed `inout` to `withUnsafeMutablePointer`.

Accesses

A particular evaluation of a storage reference expression is called an access. Accesses come in three kinds: *reads*, *assignments*, and *modifications*. Assignments and modifications are both *writes*, with the difference being that an assignment completely replaces the old value without reading it, while a modification does rely on the old value.

All storage reference expressions are classified into one of these three kinds of access based on the context in which the expression appears. It is important to note that this decision is superficial: it relies only on the semantic rules of the immediate context, not on a deeper analysis of the program or its dynamic behavior. For example, a storage reference passed as an `inout` argument is always evaluated as a modification in the caller, regardless of whether the callee actually uses the current value, performs any writes to it, or even refers to it at all.

The evaluation of a storage reference expression is divided into two phases: it is first formally evaluated to a storage reference, and then a formal access to that storage reference occurs for some duration. The two phases are often evaluated in immediate succession, but they can be separated in complex cases, such as when an `inout` argument is not the last argument to a call. The purpose of this phase division is to minimize the duration of the formal access while still preserving, to the greatest extent possible, Swift's left-to-right evaluation rules.

The Law of Exclusivity

With all of that established, we can succinctly state the first part of this proposal, the Law of Exclusivity:

> *If a storage reference expression evaluates to a storage reference that is implemented by a variable, then the formal access duration of that access may not overlap the formal access duration of any other access to the same variable unless both accesses are reads.*

This is intentionally vague: it merely says that accesses "may not" overlap, without specifying how that will be enforced. This is because we will use different enforcement mechanisms for different kinds of storage. We will discuss those mechanisms in the next major section. First, however, we need to talk in general about some of the implications of this rule and our approach to satisfying it.

Duration of exclusivity

The Law says that accesses must be exclusive for their entire formal access duration. This duration is determined by the immediate context which causes the access; that is, it's a *static* property of the program, whereas the safety problems we laid out in the introduction are *dynamic*. It is a general truth that static approaches to dynamic problems can only be conservatively correct: there will be dynamically-reasonable programs that are nonetheless rejected. It is fair to ask how that general principle applies here.

For example, when storage is passed as an `inout` argument, the access lasts for the duration of the call. This demands caller-side enforcement that no other accesses can occur to that storage during the call. Is it possible that this is too coarse-grained? After all, there may be many points within the called function where it isn't obviously using its `inout` argument. Perhaps we should track accesses to `inout` arguments at a finer-grained level, within the callee, instead of attempting to enforce the Law of Exclusivity in the caller. The problem is that that idea is simply too dynamic to be efficiently implemented.

A caller-side rule for `inout` has one key advantage: the caller has an enormous amount of information about what storage is being passed. This means that a caller-side rule can often be enforced purely statically, without adding dynamic checks or making paranoid assumptions. For example, suppose that a function calls a `mutating` method on a local variable. (Recall that `mutating` methods are passed `self` as an `inout` argument.) Unless the variable has been captured in an escaping closure, the function can easily examine every access to the variable to see that none of them overlap the call, thus proving that the rule is satisfied. Moreover, that guarantee is then passed down to the callee, which can use that information to prove the safety of its own accesses.

In contrast, a callee-side rule for `inout` cannot take advantage of that kind of information: the information is simply discarded at the point of the call. This leads to the widespread optimization problems that we see today, as discussed in the introduction. For example, suppose that the callee loads a value from its argument, then calls a function which the optimizer cannot reason about:

```
extension Array {
  mutating func organize(_ predicate: (Element) -> Bool) {
    let first = self[0]
    if !predicate(first) { return }
    ...
    // something here uses first
  }
}
```

Under a callee-side rule, the optimizer must copy `self[0]` into `first` because it must assume (paranoidly) that `predicate` might somehow turn around and modify the variable that `self` was bound to. Under a caller-side rule, the optimizer can use the copy of value held in the array element for as long as it can continue to prove that the array hasn't been modified.

Moreover, as the example above suggests, what sort of code would we actually be enabling by embracing a callee-side rule? A higher-order operation like this should not have to worry about the caller passing in a predicate that re-entrantly modifies the array. Simple implementation choices, like making the local variable `first` instead of re-accessing `self[0]` in the example above, would become semantically important; maintaining any sort of invariant would be almost inconceivable. It is no surprise that Swift's libraries generally forbid this kind of re-entrant access. But, since the library can't completely prevent programmers from doing it, the implementation must nonetheless do extra work at runtime to prevent such code from running into undefined behavior and corrupting the process. Because it exists solely to work around the possibility of code that should never occur in a well-written program, we see this as no real loss.

Therefore, this proposal generally proposes access-duration rules like caller-side `inout` enforcement, which allow substantial optimization opportunities at little semantic cost.

Components of value and reference types

We've been talking about *variables* a lot. A reader might reasonably wonder what all this means for *properties*.

Under the definition we laid out above, a property is a storage declaration, and a stored property creates a corresponding variable in its container. Accesses to that variable obviously need to obey the Law of Exclusivity, but are there any additional restrictions in play due to the fact that the properties are organized together into a container? In particular, should the Law of Exclusivity prevent accesses to different properties of the same variable or value from overlapping?

Properties can be classified into three groups:

- instance properties of value types,

- instance properties of reference types, and
- `static` and `class` properties on any kind of type.

We propose to always treat reference-type and `static` properties as independent from one another, but to treat value-type properties as generally non-independent outside of a specific (but important) special case. That's a potentially significant restriction, and it's reasonable to wonder both why it's necessary and why we need to draw this distinction between different kinds of property. There are three reasons.

Independence and containers

The first relates to the container.

For value types, it is possible to access both an individual property and the entire aggregate value. It is clear that an access to a property can conflict with an access to the aggregate, because an access to the aggregate is essentially an access to all of the properties at once. For example, consider a variable (not necessarily a local one) `p: Point` with stored properties `x`, `y`, and `z`. If it were possible to simultaneously and independently modify `p` and `p.x`, that would be an enormous hole in the Law of Exclusivity. So we do need to enforce the Law somehow here. We have three options.

(This may make more sense after reading the main section about enforcing the Law.)

The first option is to simply treat `p.x` as also an access to `p`. This neatly eliminates the hole because whatever enforcement we're using for `p` will naturally prevent conflicting accesses to it. But this will also prevent accesses to different properties from overlapping, because each will cause an access to `p`, triggering the enforcement.

The other two options involve reversing that relationship. We could split enforcement out for all the individual stored properties, not for the aggregate: an access to `p` would be treated as an access to `p.x`, `p.y`, and `p.z`. Or we could parameterize enforcement and teach it to record the specific path of properties being accessed: "", ".x", and so on. Unfortunately, there are two problems with these schemes. The first is that we don't always know the full set of properties, or which properties are stored; the implementation of a type might be opaque to us due to e.g. generics or resilience. An access to a computed property must be treated as an access to the whole value because it involves passing the variable to a getter or setter either `inout` or `shared`; thus it does actually conflict with all other properties. Attempting to make things work despite that by using dynamic information would introduce ubiquitous bookkeeping into value-type accessors, endangering the core design goal of value types that they serve as a low-cost abstraction tool. The second is that, while these schemes can be applied to static enforcement relatively easily, applying them to dynamic enforcement would require a fiendish amount of bookkeeping to be carried out dynamically; this is simply not compatible with our performance goals.

Thus, while there is a scheme which allows independent access to different properties of the same aggregate value, it requires us to be using static enforcement for the aggregate access and to know that both properties are stored. This is an important special case, but it is just a special case. In all other cases, we must fall back on the general rule that an access to a property is also an access to the aggregate.

These considerations do not apply to `static` properties and properties of reference types. There are no language constructs in Swift which access every property of a class simultaneously, and it doesn't even make sense to talk about "every" `static` property of a type because an arbitrary module can add a new one at any time.

Idioms of independent access

The second relates to user expectations.

Preventing overlapping accesses to different properties of a value type is at most a minor inconvenience. The Law of Exclusivity prevents "spooky action at a distance" with value types anyway: for example, calling a method on a

variable cannot kick off a non-obvious sequence of events which eventually reach back and modify the original variable, because that would involve two conflicting and overlapping accesses to the same variable.

In contrast, many established patterns with reference types depend on exactly that kind of notification-based update. In fact, it's not uncommon in UI code for different properties of the same object to be modified concurrently: one by the UI and the other by some background operation. Preventing independent access would break those idioms, which is not acceptable.

As for `static` properties, programmers expect them to be independent global variables; it would make no sense for an access to one global to prevent access to another.

Independence and the optimizer

The third relates to the optimization potential of properties.

Part of the purpose of the Law of Exclusivity is that it allows a large class of optimizations on values. For example, a non-`mutating` method on a value type can assume that `self` remains exactly the same for the duration of the method. It does not have to worry that an unknown function it calls in the middle will somehow reach back and modify `self`, because that modification would violate the Law. Even in a `mutating` method, no code can access `self` unless the method knows about it. Those assumptions are extremely important for optimizing Swift code.

However, these assumptions simply cannot be done in general for the contents of global variables or reference-type properties. Class references can be shared arbitrarily, and the optimizer must assume that an unknown function might have access to the same instance. And any code in the system can potentially access a global variable (ignoring access control). So the language implementation would gain little to nothing from treating accesses to different properties as non-independent.

Subscripts

Much of this discussion also applies to subscripts, though in the language today subscripts are never technically stored. Accessing a component of a value type through a subscript is treated as accessing the entire value, and so is considered to overlap any other access to the value. The most important consequence of this is that two different array elements cannot be simultaneously accessed. This will interfere with certain common idioms for working with arrays, although some cases (like concurrently modifying different slices of an array) are already quite problematic in Swift. We believe that we can mitigate the majority of the impact here with targeted improvements to the collection APIs.

Enforcing the Law of Exclusivity

There are three available mechanisms for enforcing the Law of Exclusivity: static, dynamic, and undefined. The choice of mechanism must be decidable by a simple inspection of the storage declaration, because the definition and all of its direct accessors must agree on how it is done. Generally, it will be decided by the kind of storage being declared, its container (if any), and any attributes that might be present.

Static enforcement

Under static enforcement, the compiler detects that the law is being violated and reports an error. This is the preferred mechanism where possible because it is safe, reliable, and imposes no runtime costs.

This mechanism can only be used when it is perfectly decidable. For example, it can be used for value-type properties because the Law, recursively applied, ensures that the base storage is being exclusively accessed. It cannot be used for ordinary reference-type properties because there is no way to prove in general that a particular object reference is the unique reference to an object. However, if we supported uniquely-referenced class types, it could be used for their properties.

In some cases, where desired, the compiler may be able to preserve source compatibility and avoid an error by implicitly inserting a copy instead. This is likely something we would only do in a source-compatibility mode.

Static enforcement will be used for:

- immutable variables of all kinds,
- local variables, except as affected by the use of closures (see below),
- `inout` arguments, and
- instance properties of value types.

Dynamic enforcement

Under dynamic enforcement, the implementation will maintain a record of whether each variable is currently being accessed. If a conflict is detected, it will trigger a dynamic failure. The compiler may emit an error statically if it detects that dynamic enforcement will always detect a conflict.

The bookkeeping requires two bits per variable, using a tri-state of "unaccessed", "read", and "modified". Although multiple simultaneous reads can be active at once, a full count can be avoided by saving the old state across the access, with a little cleverness.

The bookkeeping is intended to be best-effort. It should reliably detect deterministic violations. It is not required to detect race conditions; it often will, and that's good, but it's not required to. It *is* required to successfully handle the case of concurrent reads without e.g. leaving the bookkeeping record permanently in the "read" state. But it is acceptable for the concurrent-reads case to e.g. leave the bookkeeping record in the "unaccessed" case even if there are still active readers; this permits the bookkeeping to use non-atomic operations. However, atomic operations would have to be used if the bookkeeping records were packed into a single byte for, say, different properties of a class, because concurrent accesses are allowed on different variables within a class.

When the compiler detects that an access is "instantaneous", in the sense that none of the code executed during the access can possibly cause a re-entrant access to the same variable, it can avoid updating the bookkeeping record and instead just check that it has an appropriate value. This is common for reads, which will often simply copy the value during the access. When the compiler detects that all possible accesses are instantaneous, e.g. if the variable is `private` or `internal`, it can eliminate all bookkeeping. We expect this to be fairly common.

Dynamic enforcement will be used for:

- local variables, when necessary due to the use of closures (see below),
- instance properties of class types,
- `static` and `class` properties, and
- global variables.

We should provide an attribute to allow dynamic enforcement to be downgraded to undefined enforcement for a specific property or class. It is likely that some clients will find the performance consequences of dynamic enforcement to be excessive, and it will be important to provide them an opt-out. This will be especially true in the early days of the feature, while we're still exploring implementation alternatives and haven't yet implemented any holistic optimizations.

Future work on isolating class instances may allow us to use static enforcement for some class instance properties.

Undefined enforcement

Undefined enforcement means that conflicts are not detected either statically or dynamically, and instead simply have undefined behavior. This is not a desirable mechanism for ordinary code given Swift's "safe by default" design, but it's the only real choice for things like unsafe pointers.

Undefined enforcement will be used for:

- the `memory` properties of unsafe pointers.

Enforcement for local variables captured by closures

Our ability to statically enforce the Law of Exclusivity relies on our ability to statically reason about where uses occur. This analysis is usually straightforward for a local variable, but it becomes complex when the variable is captured in a closure because the control flow leading to the use can be obscured. A closure can potentially be executed re-entrantly or concurrently, even if it's known not to escape. The following principles apply:

- If a closure `C` potentially escapes, then for any variable `V` captured by `C`, all accesses to `V` potentially executed after a potential escape (including the accesses within `C` itself) must use dynamic enforcement unless all such accesses are reads.
- If a closure `C` does not escape a function, then its use sites within the function are known; at each, the closure is either directly called or used as an argument to another call. Consider the set of non-escaping closures used at each such call. For each variable `V` captured by `C`, if any of those closures contains a write to `V`, all accesses within those closures must use dynamic enforcement, and the call is treated for purposes of static enforcement as if it were a write to `V`; otherwise, the accesses may use static enforcement, and the call is treated as if it were a read of `V`.

It is likely that these rules can be improved upon over time. For example, we should be able to improve on the rule for direct calls to closures.

Explicit tools for ownership

Shared values

A lot of the discussion in this section involves the new concept of a *shared value*. As the name suggests, a shared value is a value that has been shared with the current context by another part of the program that owns it. To be consistent with the Law of Exclusivity, because multiple parts of the program can use the value at once, it must be read-only in all of them (even the owning context). This concept allows programs to abstract over values without copying them, just like `inout` allows programs to abstract over variables.

(Readers familiar with Rust will see many similarities between shared values and Rust's concept of an immutable borrow.)

When the source of a shared value is a storage reference, the shared value acts essentially like an immutable reference to that storage. The storage is accessed as a read for the duration of the shared value, so the Law of Exclusivity guarantees that no other accesses will be able to modify the original variable during the access. Some kinds of shared value may also bind to temporary values (i.e. an r-value). Since temporary values are always owned by the current execution context and used in one place, this poses no additional semantic concerns.

A shared value can be used in the scope that binds it just like an ordinary parameter or `let` binding. If the shared value is used in a place that requires ownership, Swift will simply implicitly copy the value -- again, just like an ordinary parameter or `let` binding.

Limitations of shared values

This section of the document describes several ways to form and use shared values. However, our current design does not provide general, "first-class" mechanisms for working with them. A program cannot return a shared value, construct an array of shared values, store shared values into `struct` fields, and so on. These limitations are similar to the existing limitations on `inout` references. In fact, the similarities are so common that it will be useful to have a term that encompasses both: we will call them *ephemerals*.

The fact that our design does not attempt to provide first-class facilities for ephemerals is a well-considered decision, born from a trio of concerns:

- We have to scope this proposal to something that can conceivably be implemented in the coming months. We expect this proposal to yield major benefits to the language and its implementation, but it is already quite broad and aggressive. First-class ephemerals would add enough complexity to the implementation and design that they are clearly out of scope. Furthermore, the remaining language-design questions are quite large; several existing languages have experimented with first-class ephemerals, and the results haven't been totally satisfactory.

- Type systems trade complexity for expressivity. You can always accept more programs by making the type system more sophisticated, but that's not always a good trade-off. The lifetime-qualification systems behind first-class references in languages like Rust add a lot of complexity to the user model. That complexity has real costs for users. And it's still inevitably necessary to sometimes drop down to unsafe code to work around the limitations of the ownership system. Given that a line does have to be drawn somewhere, it's not completely settled that lifetime-qualification systems deserve to be on the Swift side of the line.

- A Rust-like lifetime system would not necessarily be as powerful in Swift as it is in Rust. Swift intentionally provides a language model which reserves a lot of implementation flexibility to both the authors of types and to the Swift compiler itself.

  For example, polymorphic storage is quite a bit more flexible in Swift than it is in Rust. A `MutableCollection` in Swift is required to implement a `subscript` that provides accessor to an element for an index, but the implementation can satisfy this pretty much any way it wants. If generic code accesses this `subscript`, and it happens to be implemented in a way that provides direct access to the underlying memory, then the access will happen in-place; but if the `subscript` is implemented with a computed getter and setter, then the access will happen in a temporary variable and the getter and setter will be called as necessary. This only works because Swift's access model is highly lexical and maintains the ability to run arbitrary code at the end of an access. Imagine what it would take to implement a loop that added these temporary mutable references to an array -- each iteration of the loop would have to be able to queue up arbitrary code to run as a clean-up when the function was finished with the array. This would hardly be a low-cost abstraction! A more Rust-like `MutableCollection` interface that worked within the lifetime rules would have to promise that the `subscript` returned a pointer to existing memory; and that wouldn't allow a computed implementation at all.

  A similar problem arises even with simple `struct` members. The Rust lifetime rules say that, if you have a pointer to a `struct`, you can make a pointer to a field of that `struct` and it'll have the same lifetime as the original pointer. But this assumes not only that the field is actually stored in memory, but that it is stored *simply*, such that you can form a simple pointer to it and that pointer will obey the standard ABI for pointers to that type. This means that Rust cannot use layout optimizations like packing boolean fields together in a byte or even just decreasing the alignment of a field. This is not a guarantee that we are willing to make in Swift.

For all of these reasons, while we remain theoretically interested in exploring the possibilities of a more sophisticated system that would allow broader uses of ephemerals, we are not proposing to take that on now. Since such a system would primarily consist of changes to the type system, we are not concerned that this will cause ABI-stability problems in the long term. Nor are we concerned that we will suffer from source incompatibilities; we believe that any enhancements here can be done as extensions and generalizations of the proposed features.

Local ephemeral bindings

It is already a somewhat silly limitation that Swift provides no way to abstract over storage besides passing it as an `inout` argument. It's an easy limitation to work around, since programmers who want a local `inout` binding can simply introduce a closure and immediately call it, but that's an awkward way of achieving something that ought to be fairly easy.

Shared values make this limitation even more apparent, because a local shared value is an interesting alternative to a local `let`: it avoids a copy at the cost of preventing other accesses to the original storage. We would not encourage programmers to use `shared` instead of `let` throughout their code, especially because the optimizer will often be able to eliminate the copy anyway. However, the optimizer cannot always remove the copy, and so the `shared` micro-optimization can be useful in select cases. Furthermore, eliminating the formal copy may also be semantically necessary when working with non-copyable types.

We propose to remove this limitation in a straightforward way:

```
inout root = &tree.root

shared elements = self.queue
```

The initializer is required and must be a storage reference expression. The access lasts for the remainder of the scope.

Function parameters

Function parameters are the most important way in which programs abstract over values. Swift currently provides three kinds of argument-passing:

- Pass-by-value, owned. This is the rule for ordinary arguments. There is no way to spell this explicitly.
- Pass-by-value, shared. This is the rule for the `self` arguments of `nonmutating` methods. There is no way to spell this explicitly.
- Pass-by-reference. This is the rule used for `inout` arguments and the `self` arguments of `mutating` methods.

Our proposal here is just to allow the non-standard cases to be spelled explicitly:

- A function argument can be explicitly declared `owned`:

  ```
  func append(_ values: owned [Element]) {
    ...
  }
  ```

  This cannot be combined with `shared` or `inout`.

  This is just an explicit way of writing the default, and we do not expect that users will write it often unless they're working with non-copyable types.

- A function argument can be explicitly declared `shared`.

  ```
  func ==(left: shared String, right: shared String) -> Bool {
    ...
  }
  ```

  This cannot be combined with `owned` or `inout`.

  If the function argument is a storage reference expression, that storage is accessed as a read for the duration of the call. Otherwise, the argument expression is evaluated as an r-value and that temporary value is shared for the call. It's important to allow temporary values to be shared for function arguments because many function parameters will be marked as `shared` simply because the functions don't actually benefit from owning that parameter, not because it's in any way semantically important that they be passed a reference to an existing variable. For example, we expect to change things like comparison operators to take their parameters `shared`, because it needs to be possible to compare non-copyable values without claiming them, but this should not prevent programmers from comparing things to literal values.

  Like `inout`, this is part of the function type. Unlike `inout`, most function compatibility checks (such as override and function conversion checking) should succeed with a `shared` / `owned` mismatch. If a function with an `owned` parameter is converted to (or overrides) a function with a `shared` parameter, the argument type must actually be copyable.

- A method can be explicitly declared `consuming`.

  ```
  consuming func moveElements(into collection: inout [Element]) {
    ...
  }
  ```

  This causes `self` to be passed as an owned value and therefore cannot be combined with `mutating` or `nonmutating`.

  `self` is still an immutable binding within the method.

Function results

As discussed at the start of this section, Swift's lexical access model does not extend well to allowing ephemerals to be returned from functions. Performing an access requires executing storage-specific code at both the beginning and the end of the access. After a function returns, it has no further ability to execute code.

We could, conceivably, return a callback along with the ephemeral, with the expectation that the callback will be executed when the caller is done with the ephemeral. However, this alone would not be enough, because the callee might be relying on guarantees from its caller. For example, considering a `mutating` method on a `struct` which wants to returns an `inout` reference to a stored property. The correctness of this depends not only on the method being able to clean up after the access to the property, but on the continued validity of the variable to which `self` was bound. What we really want is to maintain the current context in the callee, along with all the active scopes in the caller, and simply enter a new nested scope in the caller with the ephemeral as a sort of argument. But this is a well-understood situation in programming languages: it is just a kind of co-routine. (Because of the scoping restrictions, it can also be thought of as sugar for a callback function in which `return`, `break`, etc. actually work as expected.)

In fact, co-routines are useful for solving a number of problems relating to ephemerals. We will explore this idea in the next few sub-sections.

`for` loops

In the same sense that there are three interesting ways of passing an argument, we can identify three interesting styles of iterating over a sequence. Each of these can be expressed with a `for` loop.

Consuming iteration

The first iteration style is what we're already familiar with in Swift: a consuming iteration, where each step is presented with an owned value. This is the only way we can iterate over an arbitrary sequence where the values might be created on demand. It is also important for working with collections of non-copyable types because it allows the collection to be destructured and the loop to take ownership of the elements. Because it takes ownership of the values produced by a sequence, and because an arbitrary sequence cannot be iterated multiple times, this is a `consuming` operation on `Sequence`.

This can be explicitly requested by declaring the iteration variable `owned`:

```
for owned employee in company.employees {
  newCompany.employees.append(employee)
}
```

It is also used implicitly when the requirements for a non-mutating iteration are not met. (Among other things, this is necessary for source compatibility.)

The next two styles make sense only for collections.

Non-mutating iteration

A non-mutating iteration simply visits each of the elements in the collection, leaving it intact and unmodified. We have no reason to copy the elements; the iteration variable can simply be bound to a shared value. This is a `nonmutating` operation on `Collection`.

This can be explicitly requested by declaring the iteration variable `shared`:

```
for shared employee in company.employees {
  if !employee.respected { throw CatastrophicHRFailure() }
}
```

It is also used by default when the sequence type is known to conform to `Collection`, since this is the optimal way of iterating over a collection.

```
for employee in company.employees {
  if !employee.respected { throw CatastrophicHRFailure() }
}
```

If the sequence operand is a storage reference expression, the storage is accessed for the duration of the loop. Note that this means that the Law of Exclusivity will implicitly prevent the collection from being modified during the iteration. Programs can explicitly request an iteration over a copy of the value in that storage by using the `copy` intrinsic function on the operand.

Mutating iteration

A mutating iteration visits each of the elements and potentially changes it. The iteration variable is an `inout` reference to the element. This is a `mutating` operation on `MutableCollection`.

This must be explicitly requested by declaring the iteration variable `inout`:

```
for inout employee in company.employees {
  employee.respected = true
}
```

The sequence operand must be a storage reference expression. The storage will be accessed for the duration of the loop, which (as above) will prevent any other overlapping accesses to the collection. (But this rule does not apply if the collection type defines the operation as a non-mutating operation, which e.g. a reference-semantics collection might.)

Expressing mutating and non-mutating iteration

Mutating and non-mutating iteration require the collection to produce ephemeral values at each step. There are several ways we could express this in the language, but one reasonable approach would be to use co-routines. Since a co-routine does not abandon its execution context when yielding a value to its caller, it is reasonable to allow a co-routine to yield multiple times, which corresponds very well to the basic code pattern of a loop. This produces a kind of co-routine often called a generator, which is used in several major languages to conveniently implement iteration. In Swift, to follow this pattern, we would need to allow the definition of generator functions, e.g.:

```
mutating generator iterateMutable() -> inout Element {
  var i = startIndex, e = endIndex
  while i != e {
    yield &self[i]
    self.formIndex(after: &i)
  }
}
```

On the client side, it is clear how this could be used to implement `for` loops; what is less clear is the right way to allow generators to be used directly by code. There are interesting constraints on how the co-routine can be used here because, as mentioned above, the entire co-routine must logically execute within the scope of an access to the base value. If, as is common for generators, the generator function actually returns some sort of generator object, the compiler must ensure that that object does not escape that enclosing access. This is a significant source of complexity.

Generalized accessors

Swift today provides very coarse tools for implementing properties and subscripts: essentially, just `get` and `set` methods. These tools are inadequate for tasks where performance is critical because they don't allow direct access to values without copies. The standard library has access to a slightly broader set of tools which can provide such direct access in limited cases, but they're still quite weak, and we've been reluctant to expose them to users for a variety of reasons.

Ownership offers us an opportunity to revisit this problem because `get` doesn't work for collections of non-copyable types because it returns a value, which must therefore be owned. The accessor really needs to be able to yield a shared value instead of returning an owned one. Again, one reasonable approach for allowing this is to use a special kind of co-routine. Unlike a generator, this co-routine would be required to yield exactly once. And there is no need to design an explicit way for programmers to invoke one because these would only be used in accessors.

The idea is that, instead of defining `get` and `set`, a storage declaration could define `read` and `modify`:

```
var x: String
var y: String
```

```
var first: String {
  read {
    if x < y { yield x }
    else { yield y }
  }
  modify {
    if x < y { yield &x }
    else { yield &y }
  }
}
```

A storage declaration must define either a `get` or a `read` (or be a stored property), but not both.

To be mutable, a storage declaration must also define either a `set` or a `modify`. It may also choose to define *both*, in which case `set` will be used for assignments and `modify` will be used for modifications. This is useful for optimizing certain complex computed properties because it allows modifications to be done in-place without forcing simple assignments to first read the old value; however, care must be taken to ensure that the `modify` is consistent in behavior with the `get` and the `set`.

Intrinsic functions

`move`

The Swift optimizer will generally try to move values around instead of copying them, but it can be useful to force its hand. For this reason, we propose the `move` function. Conceptually, `move` is simply a top-level function in the Swift standard library:

```
func move<T>(_ value: T) -> T {
  return value
}
```

However, it is blessed with some special semantics. It cannot be used indirectly. The argument expression must be a reference to some kind of local owning storage: either a `let`, a `var`, or an `inout` binding. A call to `move` is evaluated by semantically moving the current value out of the argument variable and returning it as the type of the expression, leaving the variable uninitialized for the purposes of the definitive-initialization analysis. What happens with the variable next depends on the kind of variable:

- A `var` simply transitions back to being uninitialized. Uses of it are illegal until it is assigned a new value and thus reinitialized.
- An `inout` binding is just like a `var`, except that it is illegal for it to go out of scope uninitialized. That is, if a program moves out of an `inout` binding, the program must assign a new value to it before it can exit the scope in any way (including by throwing an error). Note that the safety of leaving an `inout` temporarily uninitialized depends on the Law of Exclusivity.
- A `let` cannot be reinitialized and so cannot be used again at all.

This should be a straightforward addition to the existing definitive-initialization analysis, which proves that local variables are initialized before use.

`copy`

`copy` is a top-level function in the Swift standard library:

```
func copy<T>(_ value: T) -> T {
  return value
}
```

The argument must be a storage reference expression. The semantics are exactly as given in the above code: the argument value is returned. This is useful for several reasons:

- It suppresses syntactic special-casing. For example, as discussed above, if a `shared` argument is a storage reference, that storage is normally accessed for the duration of the call. The programmer can suppress this and force the copy to complete before the call by calling `copy` on the storage reference before.
- It is necessary for types that have suppressed implicit copies. See the section below on non-copyable types.

`endScope`

`endScope` is a top-level function in the Swift standard library:

```
func endScope<T>(_ value: T) -> () {}
```

The argument must be a reference to a local `let`, `var`, or standalone (non-parameter, non-loop) `inout` or `shared` declaration. If it's a `let` or `var`, the variable is immediately destroyed. If it's an `inout` or `shared`, the access immediately ends.

The definitive-initialization analysis must prove that the declaration is not used after this call. It's an error if the storage is a `var` that's been captured in an escaping closure.

This is useful for ending an access before control reaches the end of a scope, as well as for micro-optimizing the destruction of values.

`endScope` provides a guarantee that the given variable has been destroyed, or the given access has ended, by the time of the execution of the call. It does not promise that these things happen exactly at that point: the implementation is still free to end them earlier.

Lenses

Currently, all storage reference expressions in Swift are *concrete*: every component is statically resolvable to a storage declaration. There is some recurring interest in the community in allowing programs to abstract over storage, so that you might say:

```
let prop = Widget.weight
```

and then `prop` would be an abstract reference to the `weight` property, and its type would be something like `(Widget) -> Double`.

This feature is relevant to the ownership model because an ordinary function result must be an owned value: not shared, and not mutable. This means lenses could only be used to abstract *reads*, not *writes*, and could only be created for copyable properties. It also means code using lenses would involve more copies than the equivalent code using concrete storage references.

Suppose that, instead of being simple functions, lenses were their own type of value. An application of a lens would be a storage reference expression, but an *abstract* one which accessed statically-unknown members. This would require the language implementation to be able to perform that sort of access dynamically. However, the problem of

accessing an unknown property is very much like the problem of accessing a known property whose implementation is unknown; that is, the language already has to do very similar things in order to implement generics and resilience.

Overall, such a feature would fit in very neatly with the ownership model laid out here.

Non-copyable types

Non-copyable types are useful in a variety of expert situations. For example, they can be used to efficiently express unique ownership. They are also interesting for expressing values that have some sort of independent identity, such as atomic types. They can also be used as a formal mechanism for encouraging code to work more efficiently with types that might be expensive to copy, such as large struct types. The unifying theme is that we do not want to allow the type to be copied implicitly.

The complexity of handling non-copyable types in Swift comes from two main sources:

- The language must provide tools for moving values around and sharing them without forcing copies. We've already proposed these tools in this document because they're equally important for optimizing the use of copyable types.
- The generics system has to be able to express generics over non-copyable types without massively breaking source compatibility and forcing non-copyable types on everybody.

Otherwise, the feature itself is pretty small. The compiler implicitly emits moves instead of copies, just like we discussed above for the `move` intrinsic, and then diagnoses anything that that didn't work for.

`moveonly` contexts

The generics problem is real, though. The most obvious way to model copyability in Swift is to have a `Copyable` protocol which types can conform to. An unconstrained type parameter `T` would then not be assumed to be copyable. Unfortunately, this would be a disaster for both source compatibility and usability, because almost all the existing generic code written in Swift assumes copyability, and we really don't want programmers to have to worry about non-copyable types in their first introduction to generic code.

Furthermore, we don't want types to have to explicitly declare conformance to `Copyable`. That should be the default.

The logical solution is to maintain the default assumption that all types are copyable, and then allow select contexts to turn that assumption off. We will call these contexts `moveonly` contexts. All contexts lexically nested within a `moveonly` context are also implicitly `moveonly`.

A type can be a `moveonly` context:

```
moveonly struct Array<Element> {
  // Element and Array<Element> are not assumed to be copyable here
}
```

This suppresses the `Copyable` assumption for the type declared, its generic arguments (if any), and their hierarchies of associated types.

An extension can be a `moveonly` context:

```
moveonly extension Array {
  // Element and Array<Element> are not assumed to be copyable here
```

```
}
```

A type can declare conditional copyability using a conditional conformance:

```
moveonly extension Array: Copyable where Element: Copyable {
  ...
}
```

Conformance to `Copyable`, conditional or not, is an inherent property of a type and must be declared in the same module that defines the type. (Or possibly even the same file.)

A non-`moveonly` extension of a type reintroduces the copyability assumption for the type and its generic arguments. This is necessary in order to allow standard library types to support non-copyable elements without breaking compatibility with existing extensions. If the type doesn't declare any conformance to `Copyable`, giving it a non-`moveonly` extension is an error.

A function can be a `moveonly` context:

```
extension Array {
  moveonly func report<U>(_ u: U)
}
```

This suppresses the copyability assumption for any new generic arguments and their hierarchies of associated types.

A lot of the details of `moveonly` contexts are still up in the air. It is likely that we will need substantial implementation experience before we can really settle on the right design here.

One possibility we're considering is that `moveonly` contexts will also suppress the implicit copyability assumption for values of copyable types. This would provide an important optimization tool for code that needs to be very careful about copies.

`deinit` for non-copyable types

A value type declared `moveonly` which does not conform to `Copyable` (even conditionally) may define a `deinit` method. `deinit` must be defined in the primary type definition, not an extension.

`deinit` will be called to destroy the value when it is no longer required. This permits non-copyable types to be used to express the unique ownership of resources. For example, here is a simple file-handle type that ensures that the handle is closed when the value is destroyed:

```
moveonly struct File {
  var descriptor: Int32

  init(filename: String) throws {
    descriptor = Darwin.open(filename, O_RDONLY)

    // Abnormally exiting 'init' at any point prevents deinit
    // from being called.
    if descriptor == -1 { throw ... }
  }

  deinit {
    _ = Darwin.close(descriptor)
```

```
  }

  consuming func close() throws {
    if Darwin.fsync(descriptor) != 0 { throw ... }

    // This is a consuming function, so it has ownership of self.
    // It doesn't consume self in any other way, so it will
    // destroy it when it exits by calling deinit.  deinit
    // will then handle actually closing the descriptor.
  }
}
```

Swift is permitted to destroy a value (and thus call `deinit`) at any time between its last use and its formal point of destruction. The exact definition of "use" for the purposes of this definition is not yet fully decided.

If the value type is a `struct`, `self` can only be used in `deinit` in order to refer to the stored properties of the type. The stored properties of `self` are treated like local `let` constants for the purposes of the definitive-initialization analysis; that is, they are owned by the `deinit` and can be moved out of.

If the value type is an `enum`, `self` can only be used in `deinit` as the operand of a `switch`. Within this `switch`, any associated values are used to initialize the corresponding bindings, which take ownership of those values. Such a `switch` leaves `self` uninitialized.

Explicitly-copyable types

Another idea in the area of non-copyable types that we're exploring is the ability to declare that a type cannot be implicitly copied. For example, a very large struct can formally be copied, but it might be an outsized impact on performance if it is copied unnecessarily. Such a type should conform to `Copyable`, and it should be possible to request a copy with the `copy` function, but the compiler should diagnose any implicit copies the same way that it would diagnose copies of a non-copyable type.

Implementation priorities

This document has laid out a large amount of work. We can summarize it as follows:

- Enforcing the Law of Exclusivity:
  - Static enforcement
  - Dynamic enforcement
  - Optimization of dynamic enforcement
- New annotations and declarations:
  - `shared` parameters
  - `consuming` methods
  - Local `shared` and `inout` declarations
- New intrinsics affecting DI:
  - The `move` function and its DI implications
  - The `endScope` function and its DI implications
- Co-routine features:

    - Generalized accessors
    - Generators
- Non-copyable types
    - Further design work
    - DI enforcement
    - `moveonly` contexts

Priorities for ABI stability

The single most important goal for the upcoming releases is ABI stability. The prioritization and analysis of these features must center around their impact on the ABI. With that in mind, here are the primary ABI considerations:

The Law of Exclusivity affects the ABI because it changes the guarantees made for parameters. We must adopt this rule before locking down on the ABI, or else we will get stuck making conservative assumptions forever. However, the details of how it is enforced do not affect the ABI unless we choose to offload some of the work to the runtime, which is not necessary and which can be changed in future releases. (As a technical note, the Law of Exclusivity is likely to have a major impact on the optimizer; but this is an ordinary project-scheduling consideration, not an ABI-affecting one.)

The standard library is likely to enthusiastically adopt ownership annotations on parameters. Those annotations will affect the ABI of those library routines. Library developers will need time in order to do this adoption, but more importantly, they will need some way to validate that their annotations are useful. Unfortunately, the best way to do that validation is to implement non-copyable types, which are otherwise very low on the priority list.

The generalized accessors work includes changing the standard set of "most general" accessors for properties and subscripts from `get` / `set` / `materializeForSet` to (essentially) `read` / `set` / `modify`. This affects the basic ABI of all polymorphic property and subscript accesses, so it needs to happen. However, this ABI change can be done without actually taking the step of allowing co-routine-style accessors to be defined in Swift. The important step is just ensuring that the ABI we've settled on is good enough for co-routines in the future.

The generators work may involve changing the core collections protocols. That will certainly affect the ABI. In contrast with the generalized accessors, we will absolutely need to implement generators in order to carry this out.

Non-copyable types and algorithms only affect the ABI inasmuch as they are adopted in the standard library. If the library is going to extensively adopt them for standard collections, that needs to happen before we stabilize the ABI.

The new local declarations and intrinsics do not affect the ABI. (As is often the case, the work with the fewest implications is also some of the easiest.)

Adopting ownership and non-copyable types in the standard library is likely to be a lot of work, but will be important for the usability of non-copyable types. It would be very limiting if it was not possible to create an `Array` of non-copyable types.